[illegible]

[illegible] the Art of

[illegible]

[illegible], social

[illegible]

[illegible] and [illegible] Barton

High-performance Computing and the Art of Parallel Programming

Developments in information technology and the computerisation of society have created a data-rich world in many developed countries. There is an increasing imperative for geographers, social scientists and engineers to discover new ways of coping with this information explosion in the context of an emerging new machine age. The obvious answer is to exploit high-performance computing to develop new tools and to solve both old and new problems. This computational paradigm is destined to grow in importance.

High-performance Computing and the Art of Parallel Programming provides a non-technical introduction to high-performance computing applications together with advice about how beginners can start to write parallel programs. The authors show what HPC can offer geographers and social scientists and how it can be used in GIS. They provide examples of where it has already been used and suggestions for other areas of application in geography, GIS and the social sciences. Case studies drawn from geography explain the key principles and help the reader to understand the logic and thought processes that lie behind parallel programming.

Concerned with the art rather than the science of parallel programming this volume provides a plain and practical introduction to a subject that has previously been heavily encoded in computer jargon and will be useful to readers with a general interest in the topic as well as those wishing to develop practical parallel programming skills.

Stan Openshaw is Professor of Human Geography and **Ian Turton** is a Senior Research Fellow in the geography department of the University of Leeds.

High-performance Computing and the Art of Parallel Programming

An introduction for geographers, social scientists and engineers

Stan Openshaw and Ian Turton

ROUTLEDGE
Taylor & Francis Group

London and New York

First published 2000
by Routledge
11 New Fetter Lane, London EC4P 4EE

Simultaneously published in the USA and Canada
by Routledge
29 West 35th Street, New York, NY 10001

Routledge is an imprint of the Taylor & Francis Group

Typeset in Galliard by
J&L Composition Ltd, Filey, North Yorkshire
Printed and bound in Great Britain by St Edmundsbury Press, Bury St Edmunds, Suffolk

British Library Cataloguing in Publication Data
A catalogue record for this book is available from the British Library

Library of Congress Cataloging in Publication Data
Openshaw, Stan.
High performance computing and the art of parallel programming: an introduction for geographers, social scientists, and engineers/ Stan Openshaw and Ian Turton.
p. cm
Includes bibliographical references.
1. High performance computing. 2. Parallel programming (Computer science) I. Turton, Ian, 1952– . II. Title.
QA76.88.064 1999
004'.36—dc21 99–18439
CIP

ISBN 0–415–15692–0

Contents

Figures

Tables

Appendices

Dedication and acknowledgements

This book is dedicated to the Economic and Social Science Research Council in the hope that it may help them to realise the tremendous future importance of high-performance computing (HPC) and especially parallel programming to geography and the social sciences more generally. The practical problems of doing parallel processing are based on several years of experience in a geographical context and on a far shorter period of teaching and explaining it to masters students in geography and GIS in the School of Geography at the University of Leeds. So it can be assumed that geographers and perhaps other social scientists represent our target market. However, we would hope that the task of explaining and teaching parallel processing suitable for a geographical readership using applications from the world of GIS and modelling will also be useful to other scientists interested in HPC and parallel computing. We were and remain fascinated by all of it and hope that our interest and enthusiasm for parallel programming and HPC will successfully infect at least some of the readers of our book. Make no mistake, high-performance computing using parallel processors creates many far-reaching new opportunities, particularly in the modelling of complex human systems and in geographical data mining. Our hope is that the principles and ideas described here will endure far longer than any of the hardware used here.

Finally, we wish to thank Maureen Rosindale for so patiently typing the original manuscript. We also most gratefully acknowledge the support of the EPSRC via their funding of the Human Systems Modelling Consortium at Leeds and use of the Cray T3D and Cray J90 supercomputers. The ESRC also helped by keeping a small social science foot in the HPC door. Now that the door is ajar, the challenge is to turn the trickle into a flood!

Stan Openshaw and Ian Turton
Leeds, December 1998

1 High-performance computing: why bother with it?

This chapter attempts to justify why high-performance computing (HPC) is relevant to geography, the social sciences and in general to GIS. It is of course also very relevant to many other sciences. If you agree then maybe the effort of learning how to program HPC hardware as outlined in subsequent chapters will be very worth while. Even if you disagree, you may still find the content of interest, particularly if you are a geographer or wish to see what applications of HPC may be relevant to geographers as an illustration of computing outside the more traditional areas of science. The underlying argument is that developments in information technology and the computerisation of society are (and have) created a data-rich world in many developed countries. There is an increasing imperative for geographers, social scientists and engineers to discover new ways of coping with this information explosion in the context of an emerging new machine age. The obvious answer is to use the new technologies and exploit high-performance computing to develop new tools to solve both old and new problems. The logic is undeniable. Sooner or later most disciplines, most sciences and even most social sciences will become fully HPC-ised in the twenty-first century. This book is not about teaching the minutiae of this or that parallel-programming language. Its aim is far more strategic. It seeks to persuade you that the subject is important and then offers implementation-independent strategic advice as how best to proceed.

1.1 An HPC point of view

The purpose of this chapter is to try to persuade you that high-performance computing and parallel processing is potentially of considerable practical interest and value to both geographers and the world of geographical information systems (GIS). We are aware that this task is not straightforward, because it requires changes in attitudes and culture. In a world where high-performance computing (HPC) systems are now running at teraflop speeds (*viz.* about 15,000 times faster than the fastest PC), three stark choices face geographers: (1) ignore it all; (2) forever be restricted in what you can do because the computer systems you use are programmed by others and are in any case not state-of-the-art high-performance computers; or (3) learn how to program the high-performance computer systems of now and the future so that you can use them to do whatever you wish them to do. Most geographers involved in GIS (and elsewhere social scientists) are already the hapless but seemingly willing victims of a virulent form of 'let others do the programming for us' form of computer escapism. As a result, they are likely to be forever restricted to software packages they have little or no control over and which more or less determine what they can and cannot do. Others, who perhaps should know better, seem to have been lulled into complacency by the increased computing power offered by PCs. They ask 'What is the point of high-performance computing when with a bit of Fortran or Pascal or C programming I can do all I want on my PC?' Indeed, some others will tell you that 'what they did in 1991 on a mainframe they can now do on a PC', while some really clever folk can do it in Unix with awk! This is all true. The point is that, sadly, this is a very negative and backward-looking perspective. What these people are doing today is more or less what they first did, albeit with considerably greater difficulty, five or ten or twenty or more years ago, and they appear to think that what was good for them when they did research is also good for you when you do your research. This is not progress but regress! It is both simultaneously very understandable and an unfortunate neglect of the immense potential that HPC systems have to offer. The computational world of the twenty-first century is really quite different from previously, and there is far more to the change than a mere increase in processor speeds. There comes a moment in time when the cumulative speed-up has been so great as to spawn all manner of new ways of doing science as well as geography and social science.

1.2 HPC stimulates new research and creates new research opportunities

Let's face it. The once rarefied frontier of computational geographical research in the 1970s and 1980s (in common with many other sciences) is now a first-year PC laboratory practical. One of the authors can remember a centrographic data retrieval program developed in the 1970s by the Census Research Unit at Durham University. It then needed the largest available mainframe to run it on 1 km grid-square census data. A later version is now used on first-year undergraduate geography computer practicals at Leeds, where it runs in a few seconds on really old PCs. That is indeed progress, but it also begs the question: whither now the research frontier in computational geography? What is it that we can now apply high-performance computers in geography to that first-year undergraduates or masters students cannot do on their PC workstations? Certainly, the PC or Unix workstation of today is broadly equivalent (or even exceeds) the performance of the high-performance computers of the recent past. That is itself a most amazing technological achievement. However, nothing remains stationary. Today's high-performance computers provide many times more performance. The real million-dollar question is how do we exploit these developments to advance useful computer applications in geography and GIS?

What new research agendas are now relevant? As computers become faster and then much faster and then much faster again how do we go about doing really useful geography using them that could not previously be done without them? And, what new areas of research can a high-performance computing environment foster?

These questions are not easily answered, since by definition it involves new applications that previously do not exist because they were computationally infeasible. Maybe it will be easier to think in terms of re-engineering legacy applications to be based on better science, where 'better science' is related to available computational speed. However, not all HPC and parallel programming need be fringing on the limits of the computationally impossible. Less prosaic questions concern issues such as what GIS data structures are relevant if the entire GIS database is held in memory, not on disk? Basically, you no longer need relational databases or recursive hierarchical data structures, because they are hard to parallelise. This is good news for the simple-minded! The multidimensional data cube model is making a comeback because it is easily parallelised! Similarly, what new geoprocessing technologies that were historically impossible due to computer power constraints can now be developed? An alternative fuzzy-logic-based GIS technology immediately springs to mind as one possibility, but there are surely many others.

GIS is commonly viewed as a PC-powered desktop technology and will seemingly forever be restricted to it. However, the growth of the Internet creates the prospect that the PC merely becomes a gateway to distributed GIS, with the functionality and data being distributed over several or more sites (or processors). It is true that many simple GIS operations have no real need for HPC, but relax the assumptions slightly and the situation soon changes. For example, conventional polygon overlay is fast but totally ignores spatial error and uncertainty. If you add an error handler, then the compute times increase by about 1000 or so times. The GIS tools that exist today pre-date modern HPC, and the speeding-up in hardware over the last decade has mainly been used to expand the number of users of GIS dramatically rather than expand the quality of the GIS science being employed by the leading systems. Does it matter that modern GIS is as simple-minded as it was a decade or more ago? Once there was no other option, but now there is! The question is, therefore, how much longer before the potential and possibilities are translated into practice. One imagines that the translation of geographical information systems into geographical information science may help to foster this revision of the fundamental GIS toolkit. If it does, then the parallel processors of the future desktop systems will be the vehicles for its dissemination, but the development will have to be done on today's HPC.

1.3 Why parallel processing is important

The basic proposition is very simple. Our view is that geographers urgently need to learn some new programming skills if they are to grasp the new opportunities for creating new ways of doing computer-based research in geography to meet the increasingly urgent needs of the contemporary world for geographical analysis, modelling and understanding; and if they want to make far greater use of the rapidly all-engulfing geoinformation swamp. Sorry, but this is unavoidable, and at least some geographers need to grasp the nettle, although 'nettle' is the wrong word here, as parallel programming can easily become a source of almost endless delight, fun and inspiration. After all, the ultimate computer system is the 'supercomputer', and parallel programming is what these super machines now feed on, but it is also what the desktop systems of the future will increasingly be based upon. One strategy is very straightforward. You seek to emulate the parallel desktop systems of tomorrow using the HPC of today as the development environment. Others are more complex. For example, you can use the HPC of today to find brute force solutions to problems that you later hope to solve using far smarter methods; the HPC results provide a benchmark. Alternatively, you go for broke! You boldly create new HPC applications far beyond what current HPC can deliver and then scale up the problem sizes as HPC hardware speeds up over a decade or more.

Maybe there is another perspective that might be useful to evoke at this point. Learning how to program the world's fastest, biggest, most expensive computer systems is like driving a high-performance car. Your 'car' goes fast already, but you still want it to go faster and faster and faster. However, fast cars are already nearly at their maximum practical limits (and way outside what is legal). By comparison, high-performance computers are still at the Ford Model T stage. Just think of all that pent-up computation power that is going to appear during the next decade. The challenge, however, is not just a 'self-indulgent mega-computer geocyberpower trip' but to do something geographically useful and worthwhile with it.

Yes, we are being technology-led, but this is not a new phenomenon in the machine age. Indeed, GIS has throughout its history been technology-led. As hardware became faster and cheaper so GIS diffused outwards from the research centres. As display screen resolutions improved so did too did the map displays. As disks became bigger and cheaper so it became feasible to build and manipulate increasing large spatial databases. GIS has always been technology-led, but then so too has much of modern science. As the computationally impossible becomes possible and affordable so uses are developed for it. The hardware tools exist and the compilers exist, but what it now needs are the big new ideas and new generations of GIS-literate computerised geographers able to make the most of it all. So here in this book we plan to explain what HPC is all about and hope that it will excite others with a similar or even greater levels of enthusiasm than it has already given us.

It is surely very obvious, after more than a few moments thought, that parallel-programming skills are going to be needed if we are to come to terms with using more of the information sloshing around in a data-rich GIS world. Parallel programming is not just for the computer experts but is for all who think (rightly or wrongly, and this judgement is really for historians to make) that they can compute their way out of at least some of the problems of doing geography

and, maybe sometimes, help the modern world to solve a few more of its problems. Parallel programming is not just for computer-obsessed nerds; it could be, but the most beneficial applications will be developed by ordinary, normal folk who see it as a tool for doing GIS, spatial analysis and geography differently from what has come to characterise the previous (and current) computationally shy age. It is really little more than the next stage in computer evolution; *viz.* more data needs bigger memory spaces and faster number crunching, which in turn stimulates the capture of more data, which needs faster and bigger computers, etc. This process has been going on for a long time, but it is only recently that everything has speeded up with an exponential growth in most aspects. As a result, a vast gap has developed between what computers were once used for in geography and what they could now be used to do. It is no longer a case of more of the same as before only faster. New skills are needed to survive in this emerging IT world. Parallel programming is one of them. The nice thing here is that it comes with a guarantee that if this does not work out as you expected and your geographical research or academic career crashes in flames, then fear not, these parallel-processing skills will probably get you a job in many other disciplines where they may be more instantly appreciated. If, on the other hand, you are able to make good geographical use of these skills then you may well have gained a massive competitive edge on many others who do not, or who have not yet joined the teraflop club.

The future of geography (and most other disciplines too) is now undeniably IT-dominated. Its tools, its data, its information, its theory, its models, its concepts, etc. will all eventually be recast into a form suitable for the machine age. So why not fight the problems caused by IT with tools that exploit IT? Why not aspire to ride out the IT wave by using IT (rather than resisting it or pretending it does not exist) and see where it takes you? It may not make you massively rich, but think of all the fun you can have doing things never done before. It is potentially exciting heady stuff, but it is also hard work. There are many battles yet to be fought, let alone won, against the anti-quantification, anti-GIS, anti-computer, anti-spatial and everything-needs-deconstructing, I-hate-computing Luddites who are slowly dominating many social sciences. These people do little that is usefully constructive or applied, and many are scared of their own reflections. If you are not one of them, or even if you are, then consider reskilling. Parallel processing, parallel programming and high-performance computing (HPC) are future essential core skills which at least some geographers and social scientists will need to possess if these disciplines are to continue to prosper. It is ironic that the knockers and critics also need this so that they too can continue to thrive! By becoming a HPC geographer you are, it seems, doing many others a good turn.

1.4 Parallel computing is the future of HPC

Parallel computing has become the key component of HPC during the 1990s, and this looks set to continue into the future. If you are to exploit HPC technology usefully, you need a good understanding of how to write portable parallel programs in a general way and also to be convinced that the effort is going to be worth while. A colleague from Lancaster University recently, quite unwittingly, summarised the choices most geographers traditionally faced. You either persuade a local computer scientist to write the parallel programs for you (because it is assumed to be too difficult for a mere geographer) or as a last resort you discover how to do it yourself. We think that geographers should learn to do it themselves as a first resort. Parallel programming in Fortran or C or Java should be a basic part of future social science research training. Anyone with a modicum of programming skill or computer interest will not find it difficult. The veneer of apparent computer science complexity is in reality extremely thin and, as this book seeks to demonstrate, is easily removed.

However, be prepared for a shock. In a serial PC or workstation environment, a program written in standard C or Fortran will probably work without much or any significant change on almost any computer anywhere in the world. A program written in Java will run anywhere where there is a Java virtual machine. However, the same program will probably not work well or at all on a parallel computer or even on a more modest multi-processor system. As Geist (1996) points out, 'in general compilers cannot create an efficient parallel programming from an existing serial program. Even if such compilers existed, programmers would still need to have knowledge about parallel programming because the most popular method of parallel computing is by using a network of workstations as a *virtual* parallel supercomputer' p. vii, Geist, Foreword to Baker and Smith (1996). All this sounds hard, and it is not helped by a plethora of parallel programming texts that litter the shelves of the bookshops and tend to complicate rather than simply do the business.

Consider just one example selected almost at random. Lewis and El-Rewini (1992: p. 57) write: 'A *parallel processor* is a computer consisting of two or more processing units connected via some interconnection network. There are two major features of a parallel processor: (1) the processing units themselves, and (2) the interconnection network that ties together the collection of processors. We argue that the interconnection network is the more important of the two, and so concentrate on the *topology* of networks before examining specific commercial parallel processors' (our emphasis). But, do geographers (or indeed many of the end-users of HPC) really need to know any or much about this? We think not and argue that geographers can greatly simplify the task by dispensing with nearly all such irrelevant technical and architectural details. They are of interest only to computer scientists, computer builders, computer engineers and historians of palaeo-computing. Geographers, social scientists, indeed most of the end-users of HPC, merely need to know enough to be able to use the technology effectively and efficiently, not how to build or design new machines or understand how extinct machines once worked. We really do not need a whole lot of unnecessary details from computer science and computer engineering. Why not simply take it for granted and concentrate on using the technology? After all, no one expects geographers to know how a Pentium II or Merced chip

works before using a PC or workstation, so why are the intricacies of parallel processor and connection network topology of any great interest to geographers or social scientists interested in learning the principles of parallel programming? Geographers should be viewing parallel computing as a tool that can be used on their problems; there is absolutely no need to know more than about 0.01 per cent of the technical details of how the hardware actually works!

1.5 Aims and objectives

So most parallel programming books disagree with a simple non-computer science approach and end up providing masses of obscure technical details about hardware that is often extinct by the time the book is published and is at best of only limited historical interest. At the same time, the secrets of how to do parallel programming often remain obscure, they are not adequately described, or are presented in a way that even experienced programmers find hard to follow. Hopefully, this book will do better than this as it seeks to offer a basic introduction to high-performance computing and the programming of parallel computers. The hope is that it is possible to achieve reasonable levels of computational efficiency via a book written in a non-technical, plain English manner that geographers and other social scientists with only a basic or rusty knowledge of computing should be able to follow.

The book aims to concentrate on telling you something of the basic things that you will need to know, ignoring as much as possible of that which is probably irrelevant. We attempt to do this (1) by offering demonstrations of what can be done with HPC in geography (both to raise awareness and foster your potential interest levels) and (2) by working through geographical examples of how to convert serial code into efficient parallel code. The aim is not to provide a programmer's manual or to teach you any geography; rather, it seeks to expose the practical thought processes that occur when parallelising serial programs or when attempting to write parallel codes from scratch. The whole secret in the effective use of parallel HPC is developing the ability to rework or re-express or redesign serial code in a parallel way that matches what current hardware can usefully exploit. It is essentially a design task and not often a translation process, and it is for this reason that automatic parallelisers will always be limited in what they can deliver on many problems. Often it is the algorithm that the code represents which needs to be parallelised.

The book by Healey *et al.* (1998) is currently the only other parallel processing book relevant to GIS. It extends the range of applications provided here but is intended for a more advanced readership. They write, 'the book is aimed at a postgraduate or professional software development audience, rather than being an introductory text' (p. 6). It does not really explain how to program parallel machines rather than present descriptions of GIS applications mainly run on now defunct systems. Instead, this book offers a range of examples and simple case studies based on geographical examples from the world of GIS and spatial analysis (not matrix algebra or other applications that geographers would find obscure and irrelevant). It is really hard to understand parallel programming using example applications that are themselves not readily understood or not even remotely applicable to anything that geographers may ever wish to do or use. However, we have not attempted to write a programming book full of the micro-details of this or that language for HPC. Rather, the purpose is to offer a more general and abstract view that gives an introduction to the concepts and principles of what is involved, glimpses of how to do it, details of the design processes that are useful, and examples of application that seek to demonstrate that the effort is, or can be, very worth while. It is particularly useful to try to demystify the programming process by explaining how case studies have been tackled using a non-technical language. It is useful for teaching purposes to describe what is happening and the reasons for many of the program design decisions. It is also important to explain how to do it and why it was done in the way it was. The focus on concepts and principles is deliberate, because these are likely to be far more enduring and portable than any code written for a specific species of HPC at a particular moment in time.

1.6 Fostering a computational culture

A strongly distinctive and attractive feature about many potential geographical and social science applications of HPC concerns their applied and seemingly relevant nature to the problems of the world. The problem is that non-geographers and non-social scientists seem to see these strengths far more clearly than geographers do! There are even some signs of envy along the lines of 'you geographers are sitting on a tremendous wealth of very useful applications that have the outstanding advantage of being understandable by Joe Public and of applied relevancy'. Yet ironically politicians seem to find it easier to spend millions supporting obscure theoretical science problems than to fund social science applications that may require them to react to the results. Nevertheless, we should be aware of and willing to argue the strengths of practical usefulness that may come from using HPC in geography and GIS.

Better computer models of the economy, disease analysis tools and location optimisation methods have the outstanding attractions of being useful as well as contributing to knowledge. They may save lives, improve quality of life and reduce waste. By contrast, many areas of computational science just do not stand up to any sort of detailed comparison, with the possible exception of weather forecasting! So why then do geographers and social science downplay the need for computation, preferring instead poor-quality, low-resolution results produced on PC platforms? Why are they not seeking improved solutions by throwing more computing power at their problems? Why do we pretend that 25-year-old methods, born at a time when computers were slow and computing time massively expensive, are still relevant? Of course it is possible to muddle through without HPC. You can even do some computational physics on a PC, but no one pretends that this is ideal. It is almost as if many of the models and computer applications in geography and the social sciences

became fossilised many years ago and (with a few exceptions) remain stuck in a pre-HPC era.

Yet almost everywhere you look there are HPC applications waiting in the wings, frozen in a Narnia-like state ready for spring! Micro-simulation models are another example that are just crying out for a HPC-powered redevelopment. In the UK, the absence of social science research initiatives in HPC has not helped. Yet where are the models of human systems? It is interesting that computer scientists are more interested in these and related applications than many social scientists! There is something very amiss here!

Historically, the great problem with vector supercomputers from a geographical perspective is that hardware speeds and memory restrictions impacted more severely on geographical applications, which typically involve very large amounts of data, than on many problems in physics and chemistry. In short, the vector supercomputers were neither big enough nor fast enough. This is not arrogance, just a statement of fact. Most supercomputers were and many still are optimised for 'number crunching' without much data and not 'number munching' with masses of data. So it is only recently that HPC has started to become big enough to interest geographers, and even then the computational performance is probably too limited to meet potential people systems modelling needs and also some GIS needs. Well that is what we believe, and we are sticking to this storyline despite the raised eyebrows such claims often generate among the HPC *cognoscenti*. However, the emergence of teraflop machines dramatically changes this situation.

It is argued that many areas within geography and also many GIS applications will benefit from the adoption of an HPC-based geocomputational paradigm; see Longley *et al.* (1998), Couchelis (1998) or Openshaw and Abrahart (1999). However, being realistic, there is unlikely to be a sudden HPC revolution that will suddenly sweep all before it. Instead, in those areas that need it and where a computational paradigm may be helpful, then there is a way forward but probably only if those who are interested start to learn basic parallel-programming skills. If the current HPC machines are too slow and access is restricted then be patient: soon there will be much faster ones. But – and it is a really big BUT (see Figure 1.1) – you need to start developing the new approaches now and then safeguard your software investment by using standard portable programming languages and conforming to whatever international standards exist. Fortunately, you do not need access to the world's fastest HPC to begin the research. With modern parallel-programming tools you can now write portable, scaleable codes that can be developed and proved to work on low-end HPC platforms (multi-processor workstations, work farms based on Unix workstations and even PCs running NT) before moving on to more powerful multi-processor leading-edge HPC platforms. The essential challenge is to discover how to use HPC to extend and expand our abilities to model and analyse all types of geographical systems and not merely those which are already computerised. It would be a dreadful waste if all they were used for was to make

BUT

Figure 1.1 A big BUT. Source: Openshaw (1991, 1992, 1993, 1994, 1995, 1996, 1997, 1998, 1999, etc.) (yes, it's a joke that people from Leeds and elsewhere may appreciate!)

old legacy tools run faster, resulting in a kind of HPC-based revival of old-fashioned quantitative geography.

The opportunities are far broader than any backward-looking or historical view would suggest and offer a potential far beyond what many old quantitative geographers may once have only ever dreamt of or read in sci-fi books. Make no mistake: old-fashioned quantitative geography was essentially theory rather than computationally powered. The vision of vast number crunching to explore unimaginably complex geographical data spaces for geographical patterns to discover new concepts and build new models is not that of traditional quantitative geography or indeed of contemporary GIS. However, it is this vision that is now becoming more appropriate. In some areas, almost instant benefits can be gained, for example by switching to computationally intensive statistical methods to reduce reliance on untenable assumptions or to discover new information about the behaviour of existing models. In other areas, entirely new geocomputational applications may be expected to emerge, albeit far more slowly. As HPC continues to develop it is likely that many subjects, not just geography, will have to undergo major changes in how they operate. The combination of large amounts of data due to GIS, the availability of new AI tools and other types of computing-intensive analysis and modelling technologies, the increasing accessibility of HPC hardware, and emerging new needs for using geographical information all look set to create a new style of computational geography that in the longer term may well revolutionise many aspects of the subject by creating new ways of doing nearly all kinds of geography. The present is a very exciting time for HPC-minded geographers; it is also a time when many new significant developments are likely to occur. Parallel programming for geographers is almost certainly one of them.

1.7 Plan of the book

Our target audience is geographers interested in computing, spatial analysis and GIS. However, we believe that much of the content is relevant to the other social sciences and also to other areas of science where HPC is already more firmly entrenched. There are no strong prerequisities. Part of the content is about more general aspects (*viz.* illustrative applications in geography, descriptions of how parallel processors operate and how they can be programmed),

which is presented in plain English stripped of much of the computer science jargon. Probably two-thirds of the book talks about the art of programming in a parallel way via a small number of case studies. The examples make use of Fortran code, but this is no real restriction, because no attempt is made to teach any Fortran and the examples are sufficiently simple for anyone with more than a snatch of programming knowledge to understand them. Again the focus is on a simple non-technical (as far as possible) description and explanation of the thought processes involved in developing efficient parallel code. If you wish to develop a high level of practical parallel-programing ability, then this book provides a justification, a context and case studies within which you can practise the technical skills obtained from language-specific manuals.

This book sets about trying to meet these objectives as follows. Chapter 2 provides a general survey of HPC applications in geography and GIS, focusing on the barriers to greater usage and emphasising the potential. A number of case studies are briefly reviewed. Chapter 3 examines the principles and the concepts that are involved in HPC and parallel processing that are judged relevant for a non-computer science audience. Chapter 4 takes a closer look at different hardware types and programming models that allow us to make good use of HPC. Again there is a minimum of complexity. These two chapters set the scene for developing a working knowledge of the different HPC programming approaches in Chapter 5 (vector parallel programming), Chapter 6 (data-parallel and shared-memory programming), and Chapter 7 (simple message passing). The skills learned in these chapters are then used to create a parallel geographical analysis machine as a case study in Chapter 8. Chapter 9 offers some sage debugging and performance hints. Chapter 10 is mainly concerned with revision by looking at an another case study based on parallel code used to benchmark HPC. Finally, in Chapter 11 there are some suggestions about possible future research agendas.

2 High-performance computing applications in geography and GIS

This chapter seeks to add substance to the more general arguments raised in Chapter 1. It offers a review of HPC in a geographical context, identifies those types of application likely to benefit most and then reviews some of the work that has used HPC. It sets the scene for subsequent chapters concerned with the practicalities of doing HPC. It is important to understand the new opportunities and the potential of HPC as a justification for learning the relevant skills. The hope is that this top-down look will help to stiffen your resolve and boost your enthusiasm levels.

2.1 Introduction

There is considerable excitement in many traditional sciences about developments in high-performance computing (HPC). Computation is now regarded as a scientific tool of equal importance to theory and experimentation as supercomputers have stimulated new ways of doing science; see Hillis (1992). These developments are of enduring and far-reaching practical significance. Yet in geography and many of the social sciences, few seemingly even know what the words mean, even though these HPC developments are important and increasingly essential here too. As Openshaw (1995a) has repeatedly argued, a supercomputing-based paradigm is potentially highly relevant to many areas of human geography, with possible applications that go far beyond the very narrow historical domain of quantitative geography and geographical information systems. It provides an enabling computing environment within which many new approaches can and will be developed. Moreover, the same paradigm is also relevant to many other social sciences and humanities, particularly those with existing or potential or as yet undiscovered large-scale computational problems. New HPC-powered developments using artificial intelligence (AI) and computational intelligence (CI) based technologies are now rapidly expanding the potential for computer applications into many new areas of geography and qualitative social science; see Openshaw and Openshaw (1997) for a review of and introduction to AI in geography. Many of these tools need HPC to power them. However, there are still significant impediments, and outside traditional scientific disciplines HPC activities have historically been at a low level. Indeed, the only other

book that discusses HPC and GIS is that by Healey *et al.* (1998). It is useful to begin this book, which is dedicated to changing this situation, by having a general look at why HPC should have been so neglected and by discussing some of the potential benefits and opportunities for HPC applications in geography and GIS in the immediate future.

An obvious starting point is to understand what the words mean. The terms high-performance computing, parallel processing and supercomputing are often used interchangeably, which is more than a little confusing. However, there seems to be an increasing preference for the abbreviated HPC because of its broader and generic meaning, whereas supercomputing has historically been linked to one particular type of HPC (the vector processor). A high-performance computer (also termed HPC) is usually defined as computer hardware based on vector or multi processor parallel computers (or some mixture) that offer *at least* a two orders of magnitude increase in computing power than is available from a top-end workstation. This definition is perpetually changing in absolute terms, as current personal computers (PCs) would almost certainly have been called HPCs less than a decade ago. In fact, the performance gain from using leading-edge HPC hardware is now more likely to be at least two or three orders of magnitude as highly parallel processors take over from the earlier vector supercomputers. This whole area is now developing at a rapid rate, with most new current HPCs having only a two- to three-year life. The driving force is the speeding-up of processor chips due to increasing clock speeds and the rapid obsolescence of slower chips. As a result, the mainframe era of the 1960s to the 1980s has been superseded but not yet entirely replaced by what may be termed client server and distributed workstation-based computing. At the HPC end of the spectrum there have been similarly dramatic developments. The early 1990s saw the emergence of a parallel-computer-based approach to supercomputing. The idea of parallel processing is not new. However, both the hardware and software systems have only recently reached a degree of maturity that has made them into a practical proposition for HPC and have fostered a belief that the future lies in this direction until maybe quantum computers become a practical reality.

It is likely that future historians will recognise that the computing world underwent a major technological change during the early 1990s and that by the mid-1990s it was firmly in a new era of highly parallel supercomputing. A highly or massively parallel processor (MPP) is a computing system with multiple CPUs (or processors) that can work concurrently on a single task if the user is clever enough to program them to exploit the parallelism in his or her algorithm. Theoretically speaking, a parallel machine with 100 processors should be able to perform 100 times as much work as a single processor, provided the task is suitable for parallel computation. However, it was only recently that the technology needed to assist this prospect had matured sufficiently to become the dominant HPC machine architecture. Parallel hardware is destined to triumph over single-processor vector supercomputers because of the inherent speed limitations of individual processors. Machines capable of sustained teraflop performances are expected by AD 2000, with a further ten- to 100-fold increase during the following decade. The US ASCI (accelerated supercomputing initiative) is the current driving force as attempts are made to simulate nuclear explosions using HPC. Indeed, already in 1998 IBM announced that its Pacific Blue parallel machine had reached 3.5 teraflops, a speed equivalent to about 15,000 400 MHz PCs (in 1998, this was the fastest PC chip speed). These developments will soon dramatically increase the performance and sizes of HPC hardware available to many scientists *and* geographers – if they want to use it. Indeed, it is possible that many current computing-intensive problems in science will not be sufficiently computing-intensive to utilise these new systems fully. Maybe there are geographical and social science applications currently computationally infeasible that will soon become a practical proposition. A teraflop is 1 million million floating-point calculations per second (a single arithmetic addition counts as one floating-point calculation).

It is the planned rapid speeding-up in HPC hardware that makes the notion of a supercomputer-based geography an increasingly viable proposition. It is inevitable that sooner or later these hardware developments will create major new research opportunities pertaining to the use of high-performance computers in geography and also in many of the social sciences. The HPC hardware exists; the challenge now is to find relevant and useful applications that really need it and that are sufficiently important to justify the cost. Some critics will argue that this is computation for computation's sake, that computer speeds are driving the research and that this is the opposite of what should be happening. The response is simply that currently much of geography and GIS is actually based on computation-minimising technologies and that this has involved a large number of simplifying assumptions. This made sense when there was no alternative but much less so now that the restrictions on computation have been dramatically relaxed. Many new approaches to old problems and new approaches to previously non-computable problems are now possible or are rapidly about to become possible. This is important. HPC is not just doing more of the same faster but also opens up many new applications for computer-based approaches that previously simply did not exist. It is useful, therefore, to try to 'kick-start' the process of change by emphasising, for a while at least, the importance of computation and then challenge others to come up with the geo-parts! Geography has always been a technique-, paradigm- and philosophy-driven subject, and typically the more mature concepts have appeared afterwards. So why not here too?

An immediate and principal technical obstacle to computational geography on parallel machines is the need for a major change in programming technology. Parallelising old serial code is seldom a trivial task unless the code is already in an implicit parallel form. Thinking in parallel is important but hard, as there is usually much more to parallel programming than putting serial code through an automatic parallelising compiler. Often it involves an entirely new way of thinking about problems and how to solve them, requiring brand new algorithms and a certain amount of hard work. Maybe it is a legacy of many decades of serial thinking that is the hardest self-imposed barrier to overcome. However, hard as it may appear, now is the ideal moment to try.

2.2 Parallel programming

The good news is that there is an emerging consensus that the long-term future (i.e. next twenty years) of HPC looks set to become increasingly and exclusively based on large parallel machine architectures. Parallel HPC has looked about to triumph twice before (mid-1980s and early 1990s), but each time a mixture of false promises, over-exaggerated performance claims from the vendors, and usability questions almost killed it. Yet soon, in the early years of the new millennium, it is set to triumph at all levels of workstation architecture and not just at the top end. In the workstations and PCs of the future, multi-tasking (now a common feature but called SMP) will have given way to true multi-processing. The principal obstacle at present is the lack of efficient PC software that will permit efficient multi-processor computing. One imagines that soon the problems will be resolved here too. So at the top end parallel hardware is bound to triumph ultimately over single-processor vector supercomputers because there is only a single processor. It is quite straightforward: faster single processors merely mean an even faster multi-processor parallel machine built up from lots of single processors.

A parallel processor is a computing system with multiple processors or CPUs all working concurrently on the same problem. Distributed processing is another term for the same class of machine. A parallel machine can have virtually any number of processors, which can be organised into a number of different hardware architectures linked by different network typologies. Ultimately, what really distinguishes a massively parallel processor from a highly parallel processor is a matter of semantics. The promise of parallel programming is essentially better price/performance, speed, scaleable performance, an ability to handle bigger problems and/or more computing-intensive ones than previously. Its principal scientific attraction is the hope that computing times will decrease as the number of processors and the available memory increase, thereby providing a massive increase in the available computational power at an affordable price. Whether this happens and over what number range of processors it applies depends on the algorithm being run, its computational complexity, the nature of the specific problem, the architecture being used, the skills of the software developer and the year!

Currently little, if any, of this can be predicted, because parallelising serial code is seldom a trivial task unless the code is already in a parallel form suitable for the hardware on which it is to be run. There is usually much more to parallel programming than merely rearranging a few critical 'do loops'. Often it involves an entirely new way of thinking about computing problems and how to solve them. It is not just a means of making programs run faster but it also offers a new way of problem solving that seeks to decompose complexity into sets of autonomous processing tasks, which occasionally interact. Some would claim that this offers a whole new engineering design approach, with improved computational performance being a by-product. Others argue that parallelism is a natural phenomenon that is widely evident in the world about us. It may well simplify many modelling and analysis tasks once it is possible to escape from a serial thinking mentality, which has been forced upon us by conventional single-processor mainframe computers, vector supercomputers, PCs and workstations. An example would be the replacement of a sequentially structured micro-simulation model (where each record in a database is a person who follows deterministic rules applied in a probabilistic manner) by one based on multiple distributed agents, each representing a person, which is an autonomous processing task run in a parallel environment; see, for example, O'Hare and Jennings (1996).

However, defining generic applications that are by their nature suitable for parallel processing is not sufficient justification to invest in the necessary parallel-programming effort. *They also have to present a formidable computational challenge.* Some of the implications are considered in more detail in Section 2.7. What point is there in converting serial code that runs on a single-processor workstation in 30 minutes to run on a parallel supercomputer with 256 processors in 10 seconds! Certainly, there is a software challenge, but the computational intensity of the task may well not justify the effort involved. The 'laptop test' might be appropriate here. If the applications seem likely to be easily runable on a laptop PC in less than a week it is probably not worth while to parallelise it. If the estimated run times look likely to require more than a few months, then it begins to look far more interesting (but it still may run in 10 minutes if the code is optimised). If the estimated run times are greater than a reasonable waiting period and the code has been tuned to deliver maximum performance, then it is a potential HPC application.

An additional criterion is that the parallel application should offer some significant 'extra benefit' that could not be realised without it. There should be some evidence of either new science or better science or of better results or more timely ones. The importance of application speed should not be overlooked. Modern IT systems gather data in real time, so it would be very nice if the geographical analysis and modelling tools were also able to produce results so rapidly that they could be used in real time; see, for example, Xiong and Marble (1996). However, real-time applications also require that these geographic tools are highly automated and maybe even embedded within hardware systems so that they can run continuously forever checking and monitoring databases for the unexpected. This sort of application is clearly a good use of HPC, and the potential for the achievement of practical benefit should not be overlooked. The danger is that it is very easy to become so enthused by the thrill of parallel computing that the unique benefits that this should be providing are overlooked. Parallel processing is not an end in itself but a means to better problem solving, more useful tools and hopefully better science, and this is how it should be viewed in geography and the social sciences.

The biggest gains will probably come from those applications that were previously impossible but which can now be solved and, as a result, offer something judged to be 'worth while' knowing or having; or by being able to produce better-quality solutions to old problems of interest. It is important, therefore, in a relatively HPC-free subject such as geography, that geographers be selective in what they seek to use the new parallel-processing technology for. They should focus on those applications that most need the extra power and memory and not merely those which are simplest to code and which offer nothing extra other than the claim that there is now a parallel version of code that never really needed it or no longer needs or only needs it because of gross inefficiency in the original code or algorithm. There is another issue here in that the computational challenge of the applications also need to be scaleable so that the benefits continue even on machines 100 or 1000 times faster than those available at present. This is not as difficult as may at first appear, because the computational load imposed by many geography applications reflects spatial data resolution. As finer-resolution data become available, so the computational demands increase at least linearly and often exponentially. There is another issue here. Parallel programs that in 1999 need the world's fastest and biggest machines will within five to ten years be running on desktop systems. If you wish to preserve your research software investment, then program it in a portable parallel form. Fortunately, as this book later describes, this can now be done without too much difficulty and with some good prospect of being successful.

2.3 Geocomputation

Another reason for believing that HPC is a most significant technological development is that once computers become sufficiently fast and offer sufficiently large memories then they provide new ways of approaching geography and also many GIS applications based on what can only be termed a 'geocomputational paradigm'. Geocomputation is a relatively new term invented (or first used in its current form) in 1996. It is defined as the adoption of a large-scale computationally intensive approach to the problems of doing research in all areas of geography, including many GIS applications, although the principles are more generally applicable to other social and physical sciences; see Longley *et al.* (1998), Openshaw and Abrahart (1999). It involves porting current computationally intensive activities on to HPC platforms as well as the development of new computational techniques, algorithms and paradigms that can take particular advantage of HPC hardware and the increasing availability of spatial information.

The driving factors are threefold: (1) developments in HPC stimulate the adoption of a computational paradigm to problem solving, analysis and modelling supporting the development of new methodologies; (2) there is a need to create new ways of handling and using the increasingly large amounts of spatial and other information about the world; and (3) the increased availability of AI tools and CI methods (Bezdek, 1994), which offer new tools that are readily applicable to many areas of geography and social science. Geocomputation also involves a fundamental change of research style with the replacement of computationally minimising technologies that reflect an era of hand calculation, short-cuts and all the simplifications that this has engendered by more robust and less assumption-dependent, computing-intensive technologies. In spatial analysis, the assumption of stationarity resulted in the development of global models that are study region dependent. As computer speeds increase so new and more acceptable (from a geographical perspective) non-stationary modelling tools are emerging; see, for example, Fotheringham *et al.* (1997). Similarly, instead of relying on asymptotic distributions of statistical measures, Monte Carlo alternatives can be applied that are non-parametric and far more appropriate, albeit 1000 or more times more computationally expensive. There is also an opportunity to convert theoretical work into practical tools; for example, much of the spatial optimisation theory in Wilson *et al.* (1981) has not yet been operationalised, partly due to historical computational difficulties that no longer exist, although not many people yet realise it. For example, in creating the entropy-maximising spatial interaction model much depends on the validity of Stirling's approximation. An alternative computational approach could now be used to test its applicability. Likewise, it becomes feasible both to model systems using HPC and to construct computational experiments designed to test the acceptability of existing theories: for example, central place theory or hierarchical decision making in human flow data.

Geocomputation also comes with some grand ambitions about the potential usefulness that may well result from the fusion of virtually unlimited computing power with smart AI-based technologies, which has the potential to open up entirely new perspectives on the ways by which we do geographical research and, indeed, social science. For instance, it is now possible to think about creating large-scale computer-based experiments in which the objects being modelled are artificial people living out their lives in realistic computer-generated artificial worlds (Dibble, 1996). HPC provides a laboratory within which many geographical and social systems can be simulated, studied, analysed and modelled, if only we are bold enough to try. People behaviour modelling is no longer an impossible dream but a potentially practical reality.

While geocomputation may initially appear to be technique-dominated, it is much more than just playing with HPC for its own sake. HPC is not an executive toy! The driving force is the 'geo' part, with a distinctive focus on real-world applications. It can be regarded as the geographical analysis and modelling equivalent or follow-on from GIS. Now that GIS is here, the next revolution is geocomputation some time before 2010. Like GIS, it is essentially applied in character, but this emphasis should in no way diminish the need for solutions that rest on a sound theoretical understanding of how geographical systems really work. However, it has already been suggested that it is important not to neglect the prospect of using large-scale computer-based simulation to test, verify and understand existing theory. The Star Trek challenge is appropriate to seek to create new computational tools that are able to suggest or discover new

knowledge and new theories from the increasingly spatial-data-rich world in which we live to complement the knowledge we already possess or which can be gained by other means. A whole new subject of geographical data mining and knowledge discovery still remains to be developed. When it is developed, HPC will be the critical ingredient.

It has already been emphasised that geocomputation is much more than just using computers in geography. It is simultaneously a tool, a paradigm and a way of thinking about solving problems that is extremely relevant to the future of geography and many other social disciplines. There is an argument that geocomputation would have developed sooner if the HPC technology had been more advanced. Indeed, until as recently as 1995 it could reasonably be claimed that neither the power nor the memory capacities of the leading HPC machines were sufficient for many of problems of immediate geographical interest. What has changed since then is the maturity of parallel computing, the continued speeding-up of microprocessors, a vast increase in memory and the availability (after twenty years or so) of compilers that bring parallel computing within the existing skill domain of computationally minded geographers. The standardisation of high performance Fortran (HPF) and also of the message-passing interface (MPI, MPI2) greatly eases the task of using parallel supercomputers in many areas of geographical application as well as producing reasonably future-proof portable codes for the foreseeable future.

2.4 Raising HPC awareness

The view is expressed that in general terms the HPC hardware infrastructure needed to support a major revolution in how geography can be performed is now well developed and rapidly developing. The problem is that many geographers have not yet either realised that this is happening or understood the possible implications. In common with many other social sciences, most geographers have neglected the developments going on in the HPC world outside their disciplines. Additionally, there is no established geo-supercomputing culture that can readily benefit from HPC in the UK (in common with the USA and EU). The absence of Social Science Research Council initiatives in HPC has not helped. Two major reviews of supercomputing in the UK (Catlow (1992) and EPSRC (1995)) made no reference to any significant social science or geographical applications, and in the last decade very few geographers have been using any of the UK's available supercomputers. For example, in 1994, of the twenty-eight early users on the UK's latest, biggest and fastest parallel supercomputer (the Cray T3D at Edinburgh) there was only one geographer. By 1997, the total number of users had rocketed to over 1000 and the number of geographers to about three.

One reason may be that currently there is not a single funded grand challenge computational project anywhere in the world that is explicitly geographical in nature. A grand challenge project in science is one which is identified by a relevant discipline-specific peer group to be simultaneously of such critical importance and significance that it has to be tackled and yet it presents such severe computational problems that it is on the edge, or just beyond, what is computationally feasible with available hardware. Such a project is characterised by immense complexity, a strong link between the quality of the science and machine speed, and the use of computation as a substitute for experimentation that would otherwise be too expensive or impossible. The argument is that solving these very large and complex problems could produce enormous benefits. Examples are weather forecasting, modelling global climatic change, the human genome, fluid turbulence, aerodynamics and quantum chromodynamics. However, are there really no grand challenge problems of a computational nature that are relevant to geography and the social sciences? Of course there are! The problem is that geographers and social scientists have been very backward at putting them forward.

Openshaw (1995a) argues that human systems modelling is of equivalent (if not greater) importance and concern than any of the so-called grand challenge projects in other areas of science. Is it right that scientists claim they can model virtually all of the major physical and environmental systems while ignoring *all* the principal human systems? Historically, this neglect is understandable because of complexity, chaotic behaviour and non-linearities, lack of data, slow computers, absence of tools, etc. However, developments in IT, in the amount of data being collected and the availability of HPC many thousands of times faster and bigger than even a decade ago all suggest that human systems modelling (is becoming feasible and that the relevant research councils should seriously consider how to tackle this most important of subjects. The potential benefits in terms of better scientific understanding, improved planning and commerce that could result from an ability to model the spatial behaviour of people would be equivalent or even greater to the payoff that HPC is supposedly providing in many other areas of science; *viz.* from developing new drugs or new materials or new discoveries in physics. After all, people do matter and maybe it is time that some (let alone more) of the world's HPC resources were devoted to studying the behaviour of people rather than atoms or car crashes or molecules!

2.4.1 Some reasons for the neglect

There are a number of reasons for the current neglect of HPC in geography and the social sciences. Maybe by understanding them the best way of overcoming them will become evident.

1 Until recently, the available supercomputing hardware was probably not big enough or fast enough to offer much that was worth while, given the complexity of geographical modelling and analysis in its various human and physical settings. Indeed, it is noted that virtually all current big science computing applications of HPC started small and progressively increased in size and complexity as computer hardware developed. In geography, it is different and more difficult. The starting point required HPC to be above a certain minimum size and speed threshold before it became a viable technology. For example, the geographical analysis machine of Openshaw *et al.* (1987) required the most extraordinary efforts to 'squeeze' it within the limits provided by the then largest available supercomputing hardware. Ten years later, it runs quite easily on a workstation or a PC. It may still need a HPC, but only in those special applications that require a high quality of result reassurance.

2 In human geography, the statistical and mathematical modelling revolutions of the 1960s and 1970s were followed in the 1980s by a focus on soft qualitative approaches which de-emphasised computation and devalued programming and quantitative analysis skills by developing a style of doing geographical research that was not computable at that time. There was also a feeling that quantitative geography had failed to deliver much of what was promised in the late 1960s. On the other hand, HPC now promises speeds tens of millions of times greater than thirty years ago. Many of the historical criticisms are simply no longer relevant, although many of the critics still live in the past computing world of the early 1970s and punch cards and either totally fail to understand the present because they have absolutely no interest in it or think it is just the same as that long since past.
3 There is a strong lack of computing-intensive traditions to build on and few examples of either applications or demonstrations of success. This is partly a consequence of the previous two comments and partly slowness on the part of geographers developing applications in response to HPC hardware developments. Hopefully, this book will help to change this situation over the next decade or so.
4 Some social scientists have strong philosophical objections to logical positivism, and these are widely but wrongly perceived to present insuperable obstacles to the adoption of a computational approach. Yet no knowledgeable person would view computational geography as a revival of what some human geographers consider to be a discredited normal science paradigm. It could have this effect, but it is really philosophy invariant. Computation is just a tool, and how the tool is used and what it is used for and the context in which it is used depend on the interests, skills and value systems of the user, which are themselves grounded in contemporary society.
5 The apparent complexity of the programming and the need for new skills that many geographers may perceive as a difficulty at a time when many no longer write computer programs of any kind. This is a far more serious problem, at least initially, at a time when few generally useful HPC-oriented software packages exist that are GIS- or geography-relevant. Once parallel software starts to appear so (it is hoped) the process will become self-reinforcing.
6 Some regard computing as a substitute for thinking. This is plainly incorrect and grossly underestimates and undervalues the immense conceptual and physical effort involved in writing useful software. To survive in the IT age of future 'thinking' without HPC somewhere in the background will soon be an unthinkable fantasy!

7 There is no research agenda that emphasises computation as a paradigm for doing geographical research and no supportive resource infrastructure in place to help geographers to use HPC or get started with it. This will change, but why wait when you can actually teach yourself most of the skills in a relatively short period of time? Hopefully this book will help. The doors providing access to HPC are open if you care to look and know what to do with it.

It is even possible to attribute the current disinterest to the historical frustrations of previous attempts at scientific geography and the hardness of the task in the 1970s of seeking to be scientific at a time when both the computer hardware and data availability were so woefully inadequate. It is argued that these particular restrictions no longer exist, although it should be appreciated that there may well still be still significant problems due to widespread computer and HPC illiteracy. There is a concern that the neglect of HPC is potentially disastrous if it continues for too long. Geography has always been a vibrant discipline able to embrace new ideas and methodologies quickly. HPC is revolutionising many of the leading areas of science. More and more previously insoluble problems in geography and GIS can now be given computational solutions. It is time more geographers woke up to what is now possible.

2.4.2 Teraflop computing is here, but what is that?

Openshaw (1994a) suggests that by 1999 it is quite likely that HPC hardware available for use by geographers will be 10^9 times faster (and bigger in memory) than what was available during the quantitative revolution years of the 1960s, 10^8 times faster than was available during the mathematical modelling revolution of the early 1970s, 10^6 times since the GIS revolution of the mid-1980s and at least a further 10^2 times faster than when this book was written. One problem appears to be that most geographers have failed to appreciate what these developments in HPC mean. For instance, a CRAY T3D with 512 processors has a theoretical peak performance of 76.8 gigaflops, but what does *that* mean? A gigaflop is 1000 million floating-point operations per second, but again what does it mean in a geographical context? One way of answering this question is to create a geography-based social science benchmark code that can be run on the widest possible range of computer hardware, ranging from PC to Unix workstations to massively parallel machines. The widely used science benchmark codes measure machine performance in terms of matrix algebra or problems in physics, but it is not at all clear what relevance this has in a geographical context.

Openshaw and Schmidt (1997) have developed a social science benchmark code based on the spatial interaction model that can be run on virtually any serial or parallel processor (see also Chapter 10). The benchmark is freely available from the World Wide Web. Table 2.1 provides a preliminary assessment of the performance of some 1997 HPC hardware in terms of how many times faster it runs than a humble 486 PC running at 66 MHz. These results are already out of date. It is the general conclusions that are more important. For small problem sizes, the best performance is the SGI Onyx, followed by a vector supercomputer (the Fujitsu VPX240). However, once problem sizes increase to reflect greater data availability then soon there is no alternative to the massively parallel Cray T3D with speed gains of about 1335 times for a 10,000 by 10,000-zone matrix (equivalent to the best-resolution ward-level journey to work or migration data for all of the UK from the 1991 census). This run took 2.4 seconds, while the even larger 25,000 by 25,000 benchmark required 13 seconds. Now you might think, 'Well, thirteen seconds. Is that all' or 'Was it really worth the effort?' This view neglects the fact that without an HPC the problem could not be run at all due to the massive amounts of memory it needs. So it is important to appreciate that HPC is not just about computing speed but also about memory. The larger memory sizes required in these latter two runs reflect problems that previously were impossible to compute. Now imagine that your application involves running this 13-second model 1000 or 10,000 or 100,000 times, for example in an investigation of data uncertainty propagation effects or in bootstrapping the model to estimate confidence intervals.

Table 2.1 Relative performance of a selection of available HPC hardware on a social science benchmark code in relation to a 486 PC.

Hardware		*Problem size: numbers of origin and destinations*				
	Number of processors	*100 by 100*	*500 by 500*	*1000 by 1000*	*10,000 by 10,000*	*25,000 by 25,000*
Massively parallel						
Cray T3D	64	88				
	128		241	258		
	256			545	665	
	512				1335	1598
Parallel						
SGI Onyx	4	218	221	192	np	np
SGI Power Challenge	4	51	66	63	np	np
Vector supercomputer						
VPX240	1	162	195	196	np	np
Cray J90	8	8	35	39	np	np
Workstation						
SGI Indy	1	10	10	9	np	np
HP9000	1	14	12	10	np	np
Sun Ultra 2	1	18	17	16	np	np
Personal computer						
Pentium 133 MHz	1	3	4	4	np	np

Note: Benchmark problem sizes greater than 1000 by 1000 cannot be run on a 486 PC. The times are estimated using linear interpolation which provides a good statistical fit to a range of smaller-sized problems.

One way of explaining what these changes in HPC hardware mean is to ask how would you do geography if that PC on your desk was at least 5000 times faster and had 5000 times more memory? Probably it soon will be! It is likely that some geographers would not know what to do with the extra speed: some would not want it, but some would spot major new possibilities for using it to do geography differently. It is this type of researcher who will switch to geocomputation and will be best placed to benefit from the next two or three generations of HPC hardware. It is these people who will be the 'stars' of the next generation of geographical computing.

2.4.3 *Generic opportunities*

The opportunities are essentially fourfold:

1 to speed up existing computer-bound activities so that more extensive theory-related experimentation can be performed or to enable real-time analysis of geoinformation;
2 to improve the quality of results by using computing-intensive methods to reduce the number of assumptions and remove short-cuts and simplifications forced by computational constraints that are no longer relevant;
3 to permit larger databases to be analysed and/or to obtain better results by being able to process finer-resolution data and make good use of very large computer memory sizes, and finally;
4 to develop new approaches and new methods based on computational technologies to provide new analytical tools and models, both of which are going to be highly important in the geoinformation-rich world of the future.

All of these are important, although some are much more readily attainable than others. Indeed, in some applications there are almost instant benefits that can be gained almost immediately with a fairly small degree of effort. It is also important to recognise that HPC does not just offer two to four orders of magnitude increases in computing power above that offered by a workstation or PC but it also offers a similar improvement in memory sizes. These attractions are both of the 'and' and the 'or' variety. You can use the computation power on problems with small memory needs and on problems with large memory needs; or just on problems that require vast memory but modest computation. The principal attraction of HPC was once its exclusively number-crunching capabilities, but now it can also offer vast memory spaces for data-hungry applications. Instead of using disk-based database management software, you can store your gigabyte (and soon terabyte) databases in memory. There are potentially many geographical applications that could benefit from this, particularly those involving geographical data mining of massive databases and large-scale computer modelling.

It is also important to become both more adventurous and entrepreneurial. Soft computing technologies designed for modelling complex qualitative systems are now available for use in geography. Powerful natural-language search algorithms are just waiting for their first large-scale applications involving the analysis of text databases to extract geographical content and meaning and for other geographical purposes. At the same time, advances in artificial intelligence and computer vision provide whole new toolkits of geographically relevant methods that are rapidly becoming feasible because of developments in HPC (Openshaw, 1994e; Turton, 1997). Increasingly, it is becoming possible to think about doing almost all kinds of geography, both the quantitative and the qualitative, human and physical, quite differently in the HPC era. Geographers need to be aware of what is now possible even if they might prefer to ignore it. Put bluntly, the opportunity now exists to solve an increasing number of the problems in geography by throwing vast amounts of computational power at them. This works well in other sciences, so why not in the social sciences, where the problems are even more challenging? It is clearly an appropriate time to revisit them at last and reassess what can now be done with the latest technology under the guise of a new style of computational geography (Openshaw, 1994a). Again it is emphasised that computational geography is much more than statistical geography and GIS, because it is not necessarily statistical or mathematical or GIS-oriented but is unashamedly computational. The current HPC developments challenge key areas of current wisdom and in the longer term look set to change fundamentally how geography will be done in the twenty-first century. The present is an ideal time to start developing major new geographical applications that can begin to exploit the new opportunities by considering how to use HPC hardware.

2.4.4 *HPC and geography*

Geography is certainly a late entrant into the field of HPC, but it will not be the last as most social sciences lag even further behind. However, this deficiency needs to be remedied speedily if geography is not to fall too far behind other hard sciences and if British geographers are going to maintain and develop their expertise in this important internationally competitive area. In the UK, there was in the mid-1990s an extensive new technologies initiative (funded by JISC) that actively offered training workshops, courses and help with curriculum development, but geographers and the social sciences who needed to become involved failed to do so. HPC involves a learning curve and a reskilling process as well as a change of culture and perhaps of philosophical outlook. The computationally active researchers need to learn data parallel languages (e.g. high-performance Fortran) and adopt new programming techniques such as message passing that are essential to the future parallel-processing world (e.g. MPI). However, there are some benefits in being a late entrant into this new and rapidly developing HPC area. They include:

1 it is possible to learn from the experience of other areas of science and focus on those applications likely to benefit most;
2 the hardware and software environments are now so much better developed, which should ease some of the problems;
3 there are few or no serial legacy HPC codes that have to be ported regardless of how well they work;
4 many potential geographical applications that are map-related are naturally parallel at a coarse level of granularity;
5 many geographical applications are of fairly low complexity from a computational perspective and involve relatively small amounts of code (typically a few thousand lines of Fortran), which should ease the algorithmic redesign, rewriting and porting activities while also offering some hope of attaining high levels of performance on parallel hardware; and
6 HPC as used in a geographical context is not that different from its use in other sciences, making interdisciplinary knowledge transfer attractive; for example, the use of common algorithms (*viz.* CFD, simulated annealing, Monte Carlo methods) and a similarity of some problems (*viz.* zone design and space tessellation around an aerofoil, or the similarities in spatial map pattern recognition and robotic vision).

With a little effort, geographers can catch up and start to make rapid progress in this area. As this book shows, it is not that difficult! Virtually anyone with a modicum of rusty programming skill could do it if they wished and could see some real benefits being obtained.

2.5 HPC applications in geography and GIS

A key question is probably whether or not the effort is likely to be worth while and what types of geographical application really do need HPC. It is important not to mistakenly assume that the current small number of visible geography HPC users at national centres means that there are (1) no others or (2) no problems that need HPC in geography. On the contrary, there are a vast number of potential HPC-relevant applications and a large but not readily visible community of active HPC-powered geographers in the world. Maybe it is time that more stood up to be counted and considered HPC versions of their current serial codes; for example, geographical weighted regression (GWR) involves N^2 more computation than a conventional regression model; see Fotheringham *et al.* (1996). This methodology is applicable to non-linear regression but this would require HPC, although it is easily decomposed into N-independent subproblems and hence ideally suited to parallel processing.

2.5.1 *A typology of applications*

It is useful to identify a basic threefold typology of HPC applications relevant to geography by identifying a small number of generic, application-

independent computational tasks that appear to be intrinsically suitable for parallel supercomputing.

A first category might be termed traditional *legacy modelling applications*, which are obvious and immediate HPC applications that can be ported on to parallel hardware. Many existing mathematical and computer models are highly data-parallel and are well suited for parallel computation, particularly models that use matrix algebra: for example, input–output models, many econometric models, multi-regional demographic forecasting models, spatial regression models, spatial econometric statistical models, time-series forecasting, and the family of spatial interaction models and their many derivatives. The only real justification for HPC here is to make these existing models run much faster so that they can be run at a finer level of spatial resolution on the largest available databases and thus offer improved levels of representation, resolution and accuracy. Additionally, HPC enables the use of computing-intensive statistical procedures (such as the bootstrap, jack-knife Monte Carlo simulation) to estimate uncertainty and error propagation in computer models and in principle any computer-based analysis procedure of arbitrary complexity including sequences of GIS operations. This could involve injecting noise into data to represent sources of uncertainty and running a computer program to obtain a set of results. This would then be repeated 100 or 1000 or so times; see, for example, Openshaw *et al.* (1991). This is a naturally coarsely grained parallel task that adds value by generating confidence limits on results that previously would not have had any. It allows geographers to become much more realistic by addressing many of the problems that once had to be assumed away in the interests of tractability; see also Turton and Openshaw (1998).

A second category covers early types of what are *classical but implicit HPC applications* in geography. Some modelling and analysis tasks are naturally parallel as they involve the repeated and independent application of the same procedure (i.e. model, equations, etc.) at many different map locations simultaneously. GIS has an implicit parallel application with many opportunities for speeding up via HPC at varying scales of granularity. Many exploratory spatial analysis, modelling, search and location-optimisation problems in geography involve a search over a two- (occasionally three-) dimensional map grid that is either explicitly parallel or readily rendered so by some kind of spatial decomposition. This will allow better-quality solutions to be obtained and also make economic the application of analysis methods that previously could not be applied easily or indeed applied at all. Several other modelling and analysis tasks process very large amounts of data by the repeated application of the same basic computational procedures. For example, micro-analytical simulation modelling of a population is currently infeasible except on small data sets. The ability to handle multi-gigabyte databases easily makes it possible to scale up some types of simulation model that offer considerable potential for modelling the behaviour of whole populations at a micro-level to generate results at a more aggregate scale (Clarke *et al.*, 1995).

A third category is that of entirely new *HPC-dependent methodologies* that are being created by the increasing availability of HPC. The use of HPC to power new styles of modelling and geographical analysis that are intensely computational and that were (and some may still be) impossible (without teraflop computing speeds). It is important to begin serious development research. Many exciting developments are now theoretically possible (Openshaw, 1994g). For example, the search for invariant and recurrent 3-D objects (representing theoretical spatial pattern concepts) in a spatial database might easily involve the computation of 10 million fast Fourier transforms; see, for example, Turton (1997). Without very fast HPC hardware, this type of application is impossible and unthinkable. When what was previously impossible becomes practicable then a fertile mind should soon be able to think up many new possibilities that were previously unthinkable! It is in this fashion that a computational paradigm develops and entirely new areas of research emerge.

2.5.2 Some driving factors

The rapid spatial data explosion occasioned by GIS is another major driving factor. Developments in IT are creating an immensely spatial-data-rich world. Developments in database technology and the falling costs of storage have stimulated the increasing use of data warehouses. This is a very significant long-term development because it means that more and more micro-detail of people's activities and behaviours in space will be available for analysis and, one day soon, modelling. Major new geographical information-processing technologies are needed. The most obvious paradigm for dealing with the problems of data riches created by IT is to use other aspects of IT (i.e. HPC) to tackle them. If good, or much, or any, use of these spatial data riches is going to be made, then it is highly likely that many of the new techniques will probably have to use HPC hardware and hence utilise parallel programming. There is no other way of managing a situation in which data volumes continue to grow so much faster than single processor speeds.

Another aspect is the increasing imperative to develop new tools for the analysis of highly important databases simply because they exist and the prospect of analysis provides either a commercial benefit or a community good (Openshaw, 1994d). Other HPC needs will be created as geographers start to exploit computational technologies borrowed from other disciplines (e.g. computational fluid dynamics (CFD)) and apply them to large-scale geographical problems. Many AI tools are also intensely computational, particularly neurocomputing, evolutionary computing, cellular automata and fuzzy logic modelling. Furthermore, new computer modelling methodologies are creating new ways of studying human society using distributed AI (DAI); see, for example, Gilberts and Doran (1994) and O'Hare and Jennings (1996). DAI is the study of what happens when a set of 'intelligent' computational entities are allowed to interact and possibly communicate. In theory but not yet in practice, most aspects of human systems can be studied, including those where social beliefs, cognitive

processes and emotions are important (Gilbert and Conte, 1995). DAI offers some prospect of a big step forward in the modelling of human systems by computer experimentation with artificial societies. In the GIS arena, there is a real prospect of developing smarter geographical analysis and modelling tools. In both cases, there are two principal barriers to overcome, one technical and the other attitudinal and methodological. The former will solve itself but the latter one requires social science discipline. The disciplines are sufficiently broad-minded to tolerate a computer-based experimental approach that runs counter to traditional practice and established philosophical paradigms.

There are also some other compelling popular scientific and politically appealing reasons for viewing HPC as becoming increasingly important in a geographical context. With a high percentage of Europe and a rapidly increasing percentage of the world's population now living in urban areas, there should be a very strong imperative to develop better models of urban systems and to create an enhanced human urban systems modelling capability. It is surely unacceptable that much more is known about atmospheric circulation on Mars or the behaviour of this or that endangered species of whale than of the behaviour of even the most basic of human systems. The urban and transportation models that exist today are often well over twenty years old (Batty, 1976; Wilson, 1970). The code may well be recent, and certainly modern GUIs make them look nice and perhaps easy to use, but the underlying technology is very old, their spatial resolution is poor, and the handling of most of the principle process dynamics is crude beyond belief. It is also apparent, should we look, that many national governments probably waste billions of pounds of public money by using old and inefficient geographical technologies. Many key databases affecting us all relating to environment, health, crime, deprivation, etc. are stored, guarded and archived usually without more than a minuscule fraction of their total information content ever being used or analysed (Openshaw, 1994d). The same is true in business. How on earth can UK plc remain internationally competitive in the IT age if the fullest use is not being made of data resources and applicable IT technologies. Geo-targeting is widely used to target potential customers, but the methods in general use have changed little in the last twenty years. Where is the next generation of dynamic and adaptive, self-optimising, safe geo-targeters? Geography is the obvious source of many of these needed new developments, but so far not much has been produced. There is a risk here of 'missing the boat', with possibly catastrophic disciplinary ramifications for both geographical and social science. HPC provides the platform for a rebirth of many computer-based activities that are extremely relevant. It is an opportunity that should not be missed.

2.6 Some examples of HPC applications in geography

Despite a widespread neglect of supercomputing within geography there have been some useful developments. In the UK, the EPSRC's HPC Initiative (1994–97) funded a small geography project to port a selection of existing serial and vector codes on to the Cray T3D (Turton and Openshaw, 1998). This small but diverse portfolio of applications may mark the faltering beginnings of an HPC culture within geography. A key objective was to demonstrate some of the new science that can be performed now via case studies involving parallel supercomputing. Healey *et al.* (1998) provide some other physical geographical illustrations.

2.6.1 *Parallel spatial interaction modelling of very large data sets*

One of the earliest uses of parallel computing in geography has concerned the parallelisation of the basic spatial interaction model; see Harris (1985), Openshaw (1987). This model is central to several important areas of regional science, urban and regional planning and spatial decision support (Wilson, 1974; Birkin *et al.*, 1996). For illustrative purposes, the aggregate simplest spatial interaction model can be expressed as

$$T_{ij} = A_i\, O_i\, D_j\, B_j \exp(-bC_{ij}) \qquad (2.1)$$

where T_{ij} is the predicted flows from origin i to destination j, A_i is an origin constraint term, O_i is the size of origin zone i, D_j is the attractiveness of destination j, C_{ij} is the distance or cost of going from origin i to destination j, and b is a parameter that has to be estimated. The model was first derived in a theoretically rigorous way by Wilson (1970) using an entropy-maximising method. Clearly this model is implicitly highly parallel, since each T_{ij} value can be computed independently. Parallelisation here can be important because the model presents a computational challenge since computing times increase with the square of the number of zones (*N*). Small *N* values can be run on a PC, but large *N* values may need a supercomputer. Note that the quality of the science reflected in this model reflects both the number of zones (more zones provide better resolution than few) and the specification of the model (more sophisticated models require more computation). Developments in IT over the last decade have dramatically increased the availability and sizes of spatial interaction data sets, while the structure of the model has remained largely unchanged. Consider an example. The 1991 census provides journey-to-work and migration data for 10,764 origin and destination zones. A parallel version of Equation (2.1) has been run on the KSR1 parallel supercomputer at Manchester. It had to use straight-line distances because the storage of a 10,764 by 10,764 matrix of C_{ij} values based on network costs was infeasible at that time. Openshaw and Sumner (1995) report that the calibration of a doubly constrained spatial interaction model for the entire set of UK journey-to-work flows took 29 minutes on the KSR1 64-processor parallel supercomputer, compared with 264 hours on a single-processor Sunsparc 10/41 workstation. The same code run on the later 256-processor Cray T3D at Edinburgh required less than 3 minutes. A singly constrained model of the same data took 17.6 hours on the workstation, compared with 8 minutes on the KSR1 and 40 seconds on the Cray T3D (Turton and Openshaw, 1997). In this

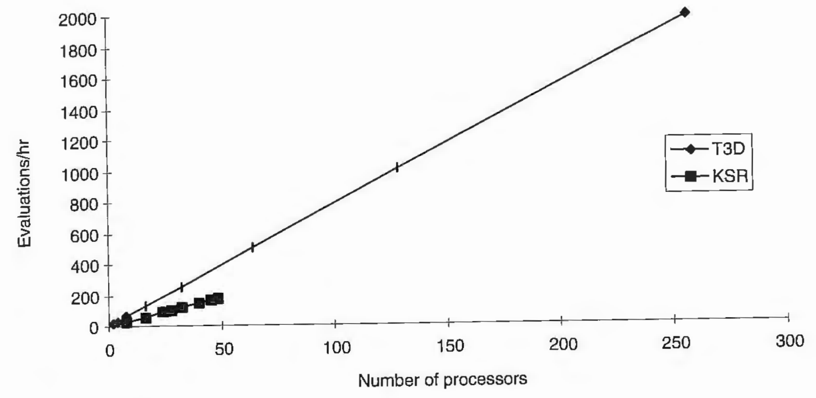

Figure 2.1 Performance of a singly constrained spatial interaction model.

latter instance, the parallelised code runs two orders of magnitude faster than the serial code, and wall clock computing times diminish linearly with the numbers of processors being used; see Figure 2.1. Scaleability is a very desirable property in the world of parallel HPC as it opens up the possibility of being able to model the largest available spatial-interaction data sets. Should this model ever be used with network distances for the C_{ij} variable instead of straight-line distances then there is no longer any alternative other than the Cray T3D as the limiting factor is the need to store $10{,}764^2$ C_{ij} values. However, this also begs the question as to how long a typical GIS would take to compute inter-zonal network times for the entire road network of the UK. This may well require a parallel GIS! However, by current standards even this census flow data set is really small. For instance, in the UK there are 1.6 million postcodes and 27 million households for which interaction data sets probably already exist: for example, telephone traffic, EFTPOS transactions. Additionally, the flows can be disaggregated by mode, gender and socio-economic structure to create the need to handle μ by μ by K sizes of flow table, where K could be in the range of 5 to 25. However, current HPC now makes it possible to model the largest flow data sets, which represent the spatial interactions of the space–time behaviours of some important aspects of the workings of an entire country: a task of both considerable practical value and immense geographical fascination.

2.6.2 *New parameter estimation methods*

Not all HPC applications require the use of large data sets. Diplock and Openshaw (1996) demonstrate some of the benefits of using genetic and evolutionary strategy-based parameter estimation methods as a replacement for conventional non-linear optimisation methods. Figure 2.2 shows that even for the simple spatial interaction model described in Equation 2.1, the function landscape for a residual sum of squares function is very complex because of

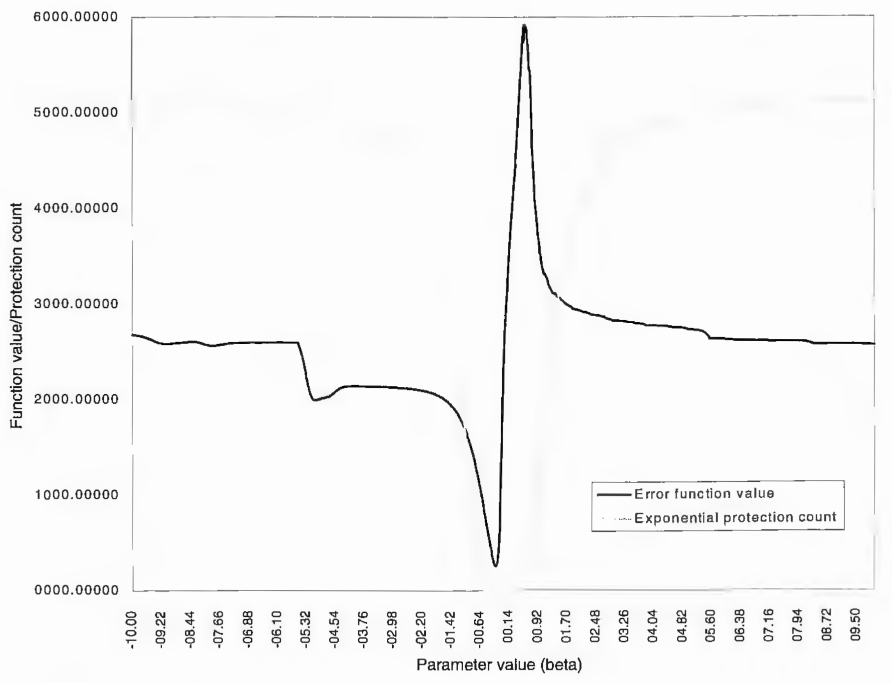

Figure 2.2 Arithmetic instability plot for a singly constrained spatial interaction model.

arithmetic instabilities due to the exponential deterrence function, which can generate very large and very small numbers, depending on the parameter β. In fact, the region where there are no risks of arithmetic exception conditions being generated is surprisingly small. The problems become worse when more parameters are used; for example, a two-parameter competing destinations version is much more complex; see Figure 2.3. Yet it is this function 'landscape' of flat regions, vertical cliffs and narrow valleys leading to the optimal result that conventional parameter-optimisation methods have to search. If they hit any of the cliffs or fall into local sink holes or find flat regions they get stuck because they have no way of escaping or of even telling you that this has happened. The new methods are able to work well on these problems as they are far more robust. However, the implications here are extremely serious, as the results imply that virtually all statistical and mathematical models with exponential terms in them could very easily produce the 'wrong' result and hence there is no assurance that the conventional non-linear optimisers are always safe to use. There are particularly serious implications here for logit and loglinear modelling, which involve the optimisation of models rich in potential exponential function problems. Once there was nothing you could do about it, but now there is provided that you can afford a factor of 100 to 1000 times more computation. This is a good example of one type of application where HPC can have an almost immediate

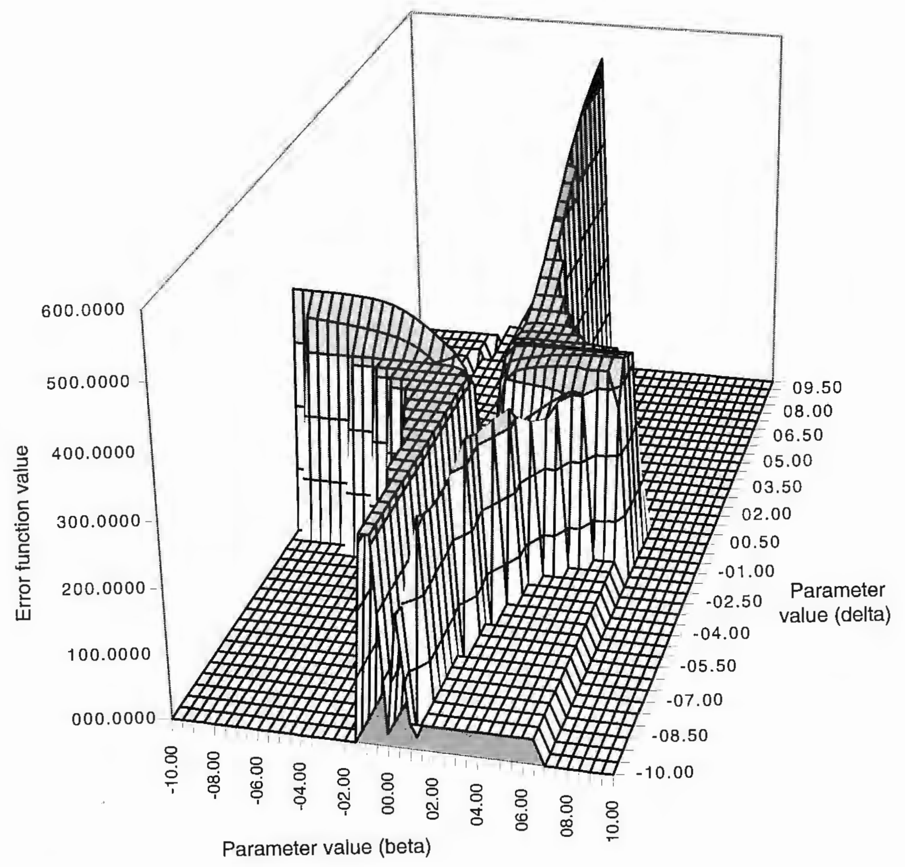

Figure 2.3 Arithmetic stability map for a two-parameter competing destinations spatial interaction model.

impact as a plug-in replacement for an older conventional technology. These new methods are also parallel or can be recast to emphasise these aspects.

2.6.3 Parameter landscapes

Another illustrative potential use of HPC is provided by the geographically local regression methods of Fotheringham *et al.* (1997) and Brunsdon *et al.* (1996). This reflects an increasing desire to move away from global model parameters to allow for spatially structured non-stationary effects. One way of doing this is to have a moving window that defines a local view of a large study region, estimate some statistic for the data within this window, map its value and move the window on a little. Another approach is to re-estimate the regression model either for each data point or for a lattice, using a kernel smoothing of the data values that reflect the local spatial distribution.

Compared with a conventional modelling approach, the extra computational load is at least N^2, but it is highly parallel. The resulting maps of parameter landscapes may offer new insights into local variability, perhaps indicative of missing explanatory variables, that would otherwise be lost or hidden in global statistics. This strategy meets one of Openshaw's (1994b) GISability criteria for useful GIS spatial analysis methods, namely that the results should not depend on the arbitrary definition of a study region. There is every indication that the potential benefits of developing this type of geographically weighted statistical technology may be considerable, provided that we can afford the computational cost of doing it properly.

2.6.4 Better spatial network and location optimisation tools

The basic spatial interaction model is often embedded in a non-linear optimisation framework that can require the model to be run many thousands or even a few millions of times in the search for an optimal solution to a planning problem, for example to determine the optimal spatial network for a set of *M* facilities given *N* possible locations, when *N* is large. There are many different types of important public and private sector spatial optimisation problems; for instance, where is the best location for a new hospital or a new hypermarket, which bank branches should be closed first with least impact on customer accessibility, where are the optimum locations for transmitters to provide maximum coverage of a population, or even where are the optimal locations for mobile paramedic vehicles at different times of the day given recent historical patterns of demand? As finer-resolution data become available so it becomes important to both use more zones and seek to improve the quality of results being provided by spatial optimisation heuristics that were originally developed over twenty years ago.

A collaborative project between EPCC and GMAP Ltd has already demonstrated some of the potential business benefits that can be gained. George (1993), and Birkin *et al.* (1995) describe the use of a retail spatial interaction model to optimise a network of car dealers using the CM-200 parallel processor. They achieved a speeding up of 2260 times compared with the original serial code. Turton and Openshaw (1996, 1997) describe how they ported this same model on to the Cray T3D and ran it using data for 822 shopping centres and 2755 origins. This version ran at an amazing rate of over 70 million model evaluations per hour, with a computing speed of 7.6 gigaflops using 256 processors. This is an astonishing 2.8 million times faster than the original serial code when run on a Sun workstation. The Cray T3D version also produced results twice as good as was previously attained, because the extra model-crunching power was used to develop a better optimisation algorithm. It is this type of application for which HPC is so well suited, because the quality of the results is directly related to the number of models that can be evaluated in a fixed time period. It is a good example of where major applied benefits can be gained by becoming more HPC-minded, provided that the application is sufficiently important to justify

the cost of the computation. While it is unlikely that many commercial organisations would want to buy a parallel HPC costing several million pounds, a few hours of computing time would probably cost no more than a workstation. However, as we suggest later, an even cheaper (almost zero-cost) option would be to simulate the performance of parallel HPC using existing PCs or workstations at night (instead of leaving them idle or switching them off). Of course a few hours may now become a day or two, but is the increase in elapsed time that significant? The point here is that HPC need no longer be either an exclusive and expensive club but is within the reach of many commercial organisations (and geography departments) if they were so minded.

2.6.5 Using HPC to create new models of geographic systems

There is also a need to improve the quality of the models being used in geographical research by 'mining' the spatial data riches created by IT and GIS. There are revolutionary computational technologies that now offer new ways of building models that either replace the existing models based on mathematical and statistical approaches or can be viewed as complementing them. One approach is to create an automated modelling system that uses genetic algorithms and genetic programming techniques to search for potentially useful new models. The automated modelling system (AMS) method of Openshaw (1988) used the Cray I vector supercomputer in an early attempt to define and then explore a small part of the vast universe of alternative spatial interaction models that could be built up from the available pieces (e.g. variables, parameters, unitary and binary operators, standard mathematical functions, and reverse Polish rules for well-formed equations) by using evolutionary programming algorithms to 'breed' new model forms. These methods are explicitly parallel (each member of a population of models can be evaluated in parallel) and also implicitly parallel (the genetic algorithm's schemata theorem). The problem with AMS was the use of fixed-length bit strings. Koza (1992, 1994) describes how this restriction can be removed by using what he terms 'Genetic programming' (GP). The AMS approach has now been redeveloped in a GP format. However, tests indicate that computing times of several weeks are needed on a fast-vector supercomputer (the Fujitsu VPX2400) and that GP is far more suitable for parallel than for vector HPC. The computational challenge is considerable. Imagine the task of evaluating 100,000 non-linear models, each of which contains a varying number of unknown parameters that have to be estimated using one (or ideally multiple) data sets. This would probably require several billion model evaluations. However, as HPC hardware becomes faster so this model-crunching strategy becomes increasingly attractive. The results from porting the genetic programming codes on to the Cray T3D suggest that not only can existing conventional models be 'rediscovered' but also that new model forms with performance levels two or three times better can be fairly easily found (Turton *et al.*, 1996, 1997; Diplock, 1996). If the new methods work well on many other data sets, then they would constitute a means of extracting knowledge and theories from

Table 2.2 Comparison of the performance of different types of spatial interaction model.

Model	*Residual standard deviation*	*Index of model performance*
Traditional gravity model	20.7	78
Entropy-maximising model	16.3	100
Best genetically bred model	12.7	128
Best genetic programming-based	11.2	141
Best neural network model	7.4	220
Best fuzzy logic model	13.1	124
Best hybrid partly fuzzy logic	11.6	141

Notes: All models are origin-constrained. Entropy-maximising model is that shown in Equation 2.1. The genetically bred model is based on four model pieces. The neural network model is a feed-forward perceptron with fifty neurons in a single hidden layer. The fuzzy logic model's fuzzy rules and membership functions were optimised by a genetic algorithm with four membership sets for each of the input variables and eight fuzzy output sets. The hybrid partly fuzzy model uses fuzzy weights associated with each input to create a mixed distance decay, entropy, intervening and competing destination model.

the increasingly geography data-rich world all around us. It is becoming increasingly possible to compute our way to better models. The problem at present is that even a single run of this approach on any reasonably sized data set could easily fully occupy a Cray T3D with 512 processors for several months. The full benefits of this approach will have to wait for teraflop (or faster) HPC, but it is possible to handle small or simple problems now and thus develop the code now that it is ready once there is sufficiently fast hardware on which to run it.

Other new approaches to building new types of spatial models are described in Openshaw (1998a). Table 2.2 gives a comparison of the performance of a selection of genetic, evolutionary, neural net and fuzzy logic spatial interaction models. In general performance, improvements of over 200 per cent over conventional models are possible, and this may be more than sufficient to justify the 10,000 to 100,000 times more computation that they involve. Some of these new models are purely black boxes (*viz.* the neural network models), but others are capable of plain English expression (the fuzzy logic models) or are in equation form. The principal constraint on the further development of some of these computational modelling methods are slow HPC speeds. Again the best strategy is to start the development process now using manageably small data sets, prove the concepts work and then scale up the problem sizes as HPC hardware improves. 'Start now, start small and dream big' is a useful slogan.

2.6.6 Flexible data reporting geographies under user control

A very common spatial data management need is the development of better ways of partitioning the map space. Many spatial planning problems involve the design of zoning systems: for example the creation of parliamentary constituencies that are of an approximately equal size and compact in shape, the identification of sales areas and facility catchment areas to equalise sales potential or

population accessibility and to design zones for reporting statistical information that are simultaneously safe from a data confidentiality point of view, statistically comparable and meet user needs; see Martin (1998). GIS has created the prospect of flexible geographical aggregation of many spatial data sets that hitherto were reported only for arbitrary and fixed sets of areas. The problem is the lack of tools for engineering zoning systems and for coping with the fact that the same data aggregated to different sets of areas will often produce completely different results; see Openshaw (1976, 1978). Statisticians refer to this as the modifiable areal unit problem (Openshaw, 1984). There is, therefore, an increasing need for automated zone design tools that will allow the use of the most appropriate zones for any given purpose.

The zone design task is a special type of optimisation problem:

$$\text{optimise } F(Z)$$

where F(Z) is some user-defined function sensitive to the aggregation of zonal data specified by Z, and Z is an aggregation of N original zones into M regions ($M < N$) such that the members of each region are contiguous with other members of the same region and each of the N zones is assigned to only one region. There may be other constraints on the nature of the data generated by Z, i.e. that the regions have to have a maximal level of compactness in their shape or be above a minimum population size. This is a specialised and hard type of optimisation problem, because Z is discrete (it is a zoning system) and F(Z) is discontinuous, non-linear and non-convex, and probably has multiple optima. Various heuristic algorithms have been developed to solve this problem; see Openshaw (1976) and Openshaw and Rao (1995). Originally, only small problems could be handled, but the availability of digital boundary information for the 150,000 census enumeration districts used in the 1991 UK census has emphasised the importance of being able to handle thousands of zones. It is likely that by 2001 1.6 million digital postcode boundaries or 32 million address points will be available as basic spatial building blocks for creating flexible user-specific zoning systems. There is a real prospect that the automated design of census output areas for the 2001 census will provide considerable financial savings and possible additional invisible benefits from the use of 'better' spatial reporting frameworks that seek to simultaneously preserve data confidentiality and minimise the amount of aggregational damage done to the data.

The zone design problem can be solved using HPC. The best results currently require the use of a simulated annealing algorithm (Openshaw and Rao, 1995) but computing times depend on the size of N. More complex constrained problems may take 1000 times longer. Fortunately the zone design code has been redeveloped in a parallel form; see Openshaw and Schmidt (1996), Turton and Openshaw (1998). Consider an example: the current allocation of urban deprivation grants in the UK depends on the DoE's deprivation indicator computed at the census ward level. There is concern among local authorities that the use of wards under-represents the real situation in some areas. It may also exaggerate

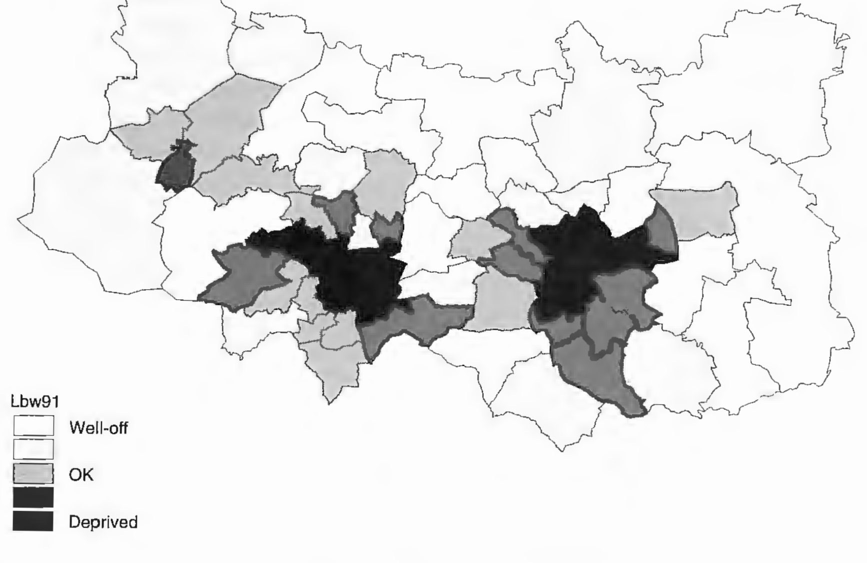

Figure 2.4 Deprivation areas in Leeds/Bradford based on wards.
Copyright: HMSO, JISC and ESRC.

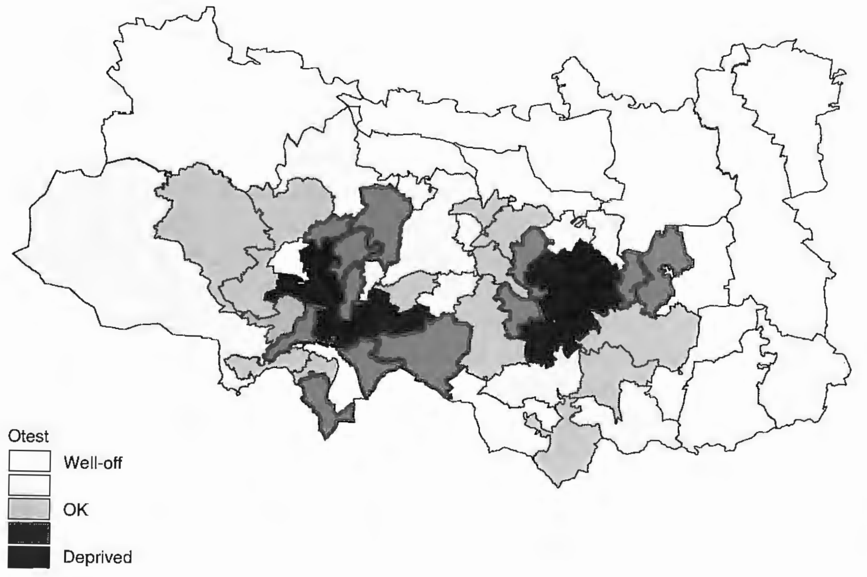

Figure 2.5 Deprivation areas in Leeds/Bradford based on ward-like areas re-engineered from enumeration districts.
Copyright: HMSO, JISC and ESRC.

the extent in others. To investigate this problem, the census enumeration district data for Leeds–Bradford has been re-engineered to maximise the number of areas that would qualify for urban aid, subject to the constraint that the areas should be compact, of a similar population size and of the same number as the current wards. Figure 2.4 shows the ward-based results, and Figure 2.5 the new re-engineered enumeration-district-based results. Even better results would have been obtained if smaller building blocks (i.e. unit postcodes) had been used. In 1991 this was not possible, and the cost was a potentially large misallocation of public funds. In 2001 it will be feasible. Potentially, this ability to engineer purpose-specific geographical frameworks for reporting statistics is of considerable practical importance and would seem to be very relevant to many areas of spatial data management. Openshaw and Alvanides (1999) provide further examples of zone design.

2.6.7 *Improved spatial classification methods*

The multivariate classification of spatial data is a very useful data reduction device. The spatial data explosion of the last two decades has increased the number of observations from a few tens (in the 1960s) to 150,000 census output areas in 1991, to 1.6 million postcode areas and 32 million households in 2001. Spatial data exists for everyone, but not all databases contain everyone. The most important change during the 1990s has been the implicit addition of geography to most databases via postcodes and postal addresses. Massive multi-gigabyte databases containing many millions of records about many millions of people are an increasingly common occurrence in commerce and government. An ability to summarise the geographical information contained in these vast databases is extremely important for a multitude of research and applied purposes. Classification methods such as multivariate cluster analysis have a long history, and most of the methods in common use date from the 1960s. They can be 'scaled up', but in a geographical context the special nature of the spatial information is usually ignored. For example, the 1991 census data are available for 150,000 small areas in Britain with up to 10,000 variables for each area. The data are a mixture of 100 per cent and 10 per cent coded information, many have been randomised to preserve confidentiality, some values have been suppressed, and the statistics are reported for geographical areas that vary in terms of size, shape and heterogeneity. Additionally, there are strong spatial autocorrelation effects, and the typical statistical distribution is more often J-shaped than bell-shaped. One solution is to develop neural-network-based classifiers that attempt to include rather than ignore the problems of spatial data classification. A Kohonen self-organising map-based approach is one such method that has been adapted to handle the problems of spatial classification (Openshaw, 1994c; Openshaw *et al.*, 1995). This algorithm is parallel but only at a very fine level of granularity. It had to be completely rewritten in a parallel data form so that it would produce good levels of performance on the Cray T3D; see Openshaw and Turton (1996) for details. On the Cray T3D with 256 processors a single run takes 10 hours, but the results are quite dissimilar from those produced by a more conventional method and tell a very different story about the structure of Britain's residential neighbourhoods.

2.6.8 *Intelligent exploratory spatial analysis systems*

As previously noted, a major by-product of the GIS revolution of the mid-1980s has been to add geographical *x*, *y* coordinates on virtually all people- and property-related computer systems. Unfortunately, there is as yet little appropriate geographical analysis technology able to efficiently and comprehensively explore these large and complex spatial databases for patterns and relationships without being told in advance precisely *where* to look, *when* to look, and *what* to look for. It is interesting that one of the earliest applications of supercomputing in geography concerned this problem. Openshaw *et al.* (1987) describe a prototype geographical analysis machine (GAM) that was able to explore a spatially referenced child cancer database for evidence of clustering. The GAM used a brute force grid search that applied a simple statistical procedure to millions of locations in a search for localised clustering. It was run on leukaemia data for northern England and is credited with the discovery of the Gateshead cancer cluster, a previously unknown problem. The long computer run times required the use of supercomputers; see Openshaw and Craft (1991). A parallel version of the latest GAM/K code has been developed and is discussed in depth in later chapters. Although the GAM no longer needs an HPC to run it, it may still be necessary if large-scale Monte Carlo simulation is used to validate the findings; see Openshaw *et al.* (1999), Openshaw (1998b).

The same basic GAM type of brute force approach has been used to search for spatial relationships. The geographical correlates exploration machine (GCEM/1) of Openshaw *et al.* (1990) examines all 2^{m-1} permutations of *m* different thematic map layers obtained from a GIS in a search for localised spatial relationships. It too is massively parallel, because each of the 2^{m-1} map permutations is independent and can be processed concurrently. Like GAM, GCEM was initially forced into a vectorisable form even though it is far more amenable to parallel processing. Here is yet another good idea that has been waiting, for over a decade, for faster parallel HPC; see also Openshaw (1998b).

An important new need is to broaden the exploratory pattern search process to include all aspects of spatial data (e.g. location in space, location in time, and multiple attributes of the space–time event) as well as to discover how to make the search more intelligent. Openshaw (1994d, 1995b) describes the development of space–time–attribute creatures: a form of artificial life that can roam around what he terms the geocyberspace in an endless hunt for pattern; see Openshaw (1994b). The claim to being intelligent results from the generic algorithm used to control the search process and the use of computational statistics to reduce the dangers of spurious results. It is strongly dependent on having sufficient parallel computational power to drive the entire process. Openshaw and Perree (1996) show how the addition of computer animation can help users to

visualise and understand the geographical analysis. This type of highly exploratory search technology is only just becoming feasible with recent developments in HPC, and considerable research is still needed to perfect the methods. However, it promises a radically different and understandable approach to exploratory spatial analysis in GIS that is powered by HPC. No doubt there will be many other variations on this broad theme once it is realised that there are no longer any meaningful computer restrictions on what you may wish to do. It is a great time to be inventive.

2.6.9 *Using HPC to build geographical knowledge systems*

GIS has provided a micro-spatial data-rich world, but few or no tools are able to help identify either the more abstract recurrent patterns that exist at higher levels of generalisation or new concepts from the data riches (Openshaw, 1994e). Geography contains many theories about space that can be expressed as idealised two- and three-dimensional patterns that are supposedly recurrent. Traditionally, these concepts and theories have been tested using aspatial statistical methods that require much of the geography to be removed purely so that analysis can be performed. For example, does the spatial social structure of Leeds as shown by the 1991 census conform to a broadly concentric ring type of pattern? This hypothesis can be tested using statistical methods by first defining a central point, specifying a series of rings of fixed width and then using a statistic of some kind computed using census data to test the *a priori* hypothesised trends in social class. However, this clearly requires considerable precision and is not really an adequate test of the original hypothesis, which did not specify ring widths, identify a central point or define at what level of geographic scale the pattern exists. A possible solution is to use pattern recognition and robotic vision technology to see whether any evidence of a general concentric geographical structure exists in the census data for Leeds, after allowing for the distorting effects of scale, site and topography; see Turton (1998). If no idealised concentric patterns exist, then which of a library of different pattern types might be more appropriate? The HPC revolution of the mid-1990s provides an opportunity to become less precise and more general by developing spatial pattern-recognition tools that can build up recurring map pattern libraries of the many different types of recurrent idealised forms. Suppose you ask the question: how many different spatial patterns do British cities exhibit? Currently, this cannot be answered, but at least the tools exist to allow geographers to start to find out. Openshaw (1994e) argues that a more generalised pattern-recognition approach provides the basis for a fresh look at geographical information with a view to developing entirely new ways of extracting useful new knowledge from it. However, this will only become possible as HPC enters the teraflop era and it becomes feasible to apply pattern templates to many millions of locations at many different levels of geographical resolution. It is computationally extremely intensive but highly parallel and has promise as one approach to geographical data mining.

2.6.10 *New results from old models*

A related opportunity for a quick gain in benefit from HPC is the use of the bootstrap to estimate parameter variances; see Efron and Gong (1983) for details. This is quite straightforward but needs an HPC to make it feasible. You merely have to run the model a few hundred or a few thousand times or $N-1$ times (where N is the number of observations). This is naturally parallel, because each run can be assigned to a different processor. Indeed, no great parallelisation effort is needed and this strategy can be applied to many existing models in order to identify confidence intervals for predictions. Research with a multi-region model used to make population forecasts for the European Union has identified the error limits in forecasts made for 2021 to 2051. The results are surprising, as they suggest that currently there are no reliable long-term forecasts for the EU as the confidence limits are extremely wide. The problem appears to be due to uncertainty in the fertility and mortality rate forecasts; see Turton and Openshaw (1998) for further details.

2.7 Parallel GIS applications

2.7.1 *Supercharging GIS*

Apart from geocomputation, GIS is the other area where parallel processing is likely to be useful. Healey *et al.* (1998) are correct to argue that performance will soon become a serious concern in GIS. They write:

> The expectations of potential users are high, yet the throughput of useful results from GIS analysis is often limited. Processor, memory, disk and network constraints still temper the enthusiasm of even the most energetic and insomniac of postgraduate students. Similarly, contract deadlines tick past . . . while dozens of workstations stand idle at night. Power on the desktop has brought excellent interactivity to the user interface, but it has yet to be harnessed enterprise-wide for cost effective GIS processing.
>
> (pp. 1–2)

Parallel GIS offers an obvious solution, even if it may not be an exclusively HPC one. They list the following reasons (pp. 1–2):

1 even though computer chip speeds are increasing they are no match for the growth of available GIS and remote-sensing data;
2 a growing need for real-time GIS applications with fast or even sub-second response times on large and dynamic data sets;
3 the more exploratory, combinatorial or interactive kinds of analysis require hardware far more powerful than workstations;
4 potential users of GIS have expectations of high performance, yet the throughput of useful results from GIS analysis is often limited;

5 the increasing use of mapping and analysis in conjunction with the World Wide Web, with millions of potential users, will result in enormous demands for multi-streamed performance; and
6 given these demands for both high-performance computation and data input and output it is now a matter of concern that GIS is still largely based on algorithmic approaches originally developed for slow serial processors.

They write 'Parallelisation can . . . be seen as a way of "supercharging" a GIS to give increased performance' (p. 91). Their vision is one in which the parallel GIS implementation is totally transparent to the user; it looks and feels the same but it runs much faster. So 'with GIS applications clamouring for enhanced throughput, and the potential offered by parallel processing, bringing the technologies together would seem to offer substantial benefits, if successful' (pp. 3–4). We entirely agree.

2.7.2 *Some problems*

Healey *et al.* (1998) also outline a number of potential problems, including:

1 until recently, parallel processors focused on number-crunching computing power rather than input–output bandwidth, whereas GIS is both compute and input–output intensive;
2 GIS data structures are complex and reflect a serial computing era based on relational databases, which are not readily parallelised;
3 many GIS operations are multi-stage, requiring the application of several algorithms in sequence, not all of which are parallelisable;
4 computation and input–output may be interleaved during the same operation;
5 it may be appropriate to link the code for individual operations to a proprietary database manager, which may not be compatible with parallel hardware; and
6 while there is a shortage of skilled staff with parallel programming expertise, there are even fewer who are also knowledgeable of GIS algorithms.

Healey *et al.*'s (1998) strategy is to explain and document the issues and problems so that future work can build on it in a structured manner. It is useful, therefore, to briefly review and comment on the parallel GIS applications that they suggested. In particular, it is essential to consider whether the granularity or scale of the parallelism in many GIS applications is appropriate for present and future parallel hardware. These topics are covered in greater detail in subsequent chapters, but the current purpose here is to try to establish whether or not parallel GIS is likely to be a viable HPC concern or more relevant to a more limited small-scale multi-processor workstation technology.

2.7.3 *Parallel GIS algorithms: for HPC or multi-processor workstations?*

Healey *et al.* (1998) identify vector polygon overlay and raster–vector conversion as the most important fundamental GIS algorithms for parallelisation but then added vector–raster and a few others to their list: generalisation, terrain modelling, parallel database management, spatial analysis and remotely sensed image analysis. However, the various chapters of their book are mainly concerned with theoretical issues presented in a rather abstract and hard to comprehend manner rather than with their implementation in working GIS systems.

A critical distinction to bear in mind is between looking for parallelism *within* an algorithm and/or looking for it *at a coarser level.* As HPC hardware becomes faster, so the units of work that a task is decomposed into will need to increase in computational size. For example, once it made sense to try to parallelise a sort program or a floating-point operation at some fairly microscopic level; today, it almost certainly does not. Also, basic common sense needs to be applied. Hence there is almost certainly no point in parallelising point-in-polygon algorithms, because they are already fast enough and the parallelism is hard to find. The same may be true for polygon overlay and many other common GIS functions.

Indeed, is it really worth the effort of parallelising GIS functions that typically require only a few seconds or minutes (or less) of computing time, or should you instead concentrate on the parallelism in the application calling the GIS operation a large number of times? If the latter never happens and the former is all that ever happens then there may well be little to be gained, and parallel GIS is more a multi-processor workstation type of problem than a serious challenge to HPC.

This point can be clearly seen in the chapter on 'Spatial Analysis' by Densham and Armstrong (1998). They analyse the performance of a Getis and Ord (1992) g-statistic. In its original form, the amount of computation is N^2, where N is the number of points or zones being processed. They obtained good results on various parallel machines, but you need very large N values before computing times increase to become even slightly troublesome. For example, they quote a single-processor KSR that took 5919 seconds to process 10,000 points but only 124 seconds when fifty-nine processors were used. However, a modern workstation (five years later on) would be able to run this application in a few hundred seconds (maybe much less). Additionally, the crafty and cunning programmer would use clever spatial data-retrieval algorithms to try to remove the N^2 effect so that it would take only a few seconds on a PC. Likewise, Densham and Armstrong (1998) also present a shortest path parallelisation run on a transputer, parallel interpolation and hill shading that all substitute computing power for intelligence when calculating nearest neighbours in point data. There is no dispute about their achievements from a historical parallel algorithms perspective (the transputer no longer exists), but their methods do not need HPC and, if better algorithms were used for the nearest neighbour search, they would most certainly run fast enough on a PC for most applications.

A similar conclusion may one day be applicable to many but not all of the parallel algorithms discussed in Healey *et al.* (1998). The most important exceptions are almost certainly those applications where either enormous quantities of data are involved or where after careful tuning of spatial analysis code the total run time is unacceptably large or where the same sequence of operations is likely to be repeated many thousands of times (e.g. in Monte Carlo simulation of error propagation). On the other hand, from a GIS developer's perspective the fact that parallel GIS is unlikely to need HPC for many of its applications is a very useful conclusion, since it is likely that modest numbers of tightly coupled processors are probably all that most end-users will ever aspire to, and even here many of these developments will probably be driven by hardware availability rather than any real need for a new parallel GIS.

So our view is that parallel GIS is both a good idea and an obvious development, but to be frank most of GIS does not need an HPC platform. Indeed, in many cases the extent of the parallelism is small enough to suggest that relatively few processors will be all that most general-purpose users of GIS will probably need. However, there are some important exceptions and these will discussed further later.

2.8 Overcoming access barriers

Geographers and social scientists will probably always have problems in attaining sufficient HPC resources for very large-scale projects, although more modest applications will tend to succeed far more easily. The reason is simply that the historical justification for many HPC centres throughout the world is that they support not just theoretical physicists, chemists, and engineers but also a broader constituency that includes geographers and social scientists with suitable applied problems. Indeed, it is probably worth noting here that the only real social science plus point at present is the growing belief (albeit held by non-social scientists) that in the near future an increasing number of supercomputer applications will occur outside the traditional large supercomputer-using areas of hard science. These new application areas are thought likely to relate to the commercial exploitation of HPC, where data mining linked to large data warehouse developments will need vast amounts of computing resources if they are to be properly used (Small and Edelstein, 1997). Data mining is currently of interest to large companies (telecommunications utilities, banks, large retail chains, insurance companies, etc.). The investments in creating the data infrastructures are well advanced. Their hope now is that the resulting data resources can be 'mined' for new knowledge, hidden patterns and relationships that can be used to maximise profits, reduce waste, develop new business and generally help large organisations to compete and survive in the emerging IT age. There is also a great and pressing need to start to use for both blue skies research and application for more of the spatial data riches created by developments in IT. Currently, it can be viewed as a crime that most databases relevant to the public good (particularly related to health, wealth and crime) are not being analysed to any significant extent. HPC provides the platform for doing it; what is currently missing are many of the tools, the software, and a geographical analysis and modelling culture that makes the entire enterprise worth while.

3 Parallel and high-performance computing: concepts, principles and theory

The next stage in understanding what HPC has to offer involves developing knowledge of what the various words and jargon mean. However, this is not about computer engineering trivia or computer science theory. It is about the different types of parallel computing. There is a brief and simple look at Amdahl's law and an even briefer look at the historical setting within which the current hardware is located. The single most important prerequisite is learning how to think in a parallel way. Much of the chapter is concerned with explaining how to think parallel. Indeed, some will argue that thinking or rethinking serial problem solving in a parallel way is simultaneously the hardest and most fun part of parallel programming.

3.1 What is parallel computing?

Parallel computing is no big deal! It is merely the use of more than one processor to solve a problem. Nor is it a new idea. Almasi and Gottlieb (1989) write, 'Parallel processing is ready to happen. The demand is there, and now the technology is there . . . A new dimension is being opened. It's an exciting time' (p. v, preface). Indeed, a decade later the same statement could still be made, except that now parallel processing is not only here but is also happening on a large scale. It is no longer promise but reality. It is also very exciting, since the existence of a rapidly spreading user-friendly and applicable parallel-computing technology is one of the most significant and far-reaching developments to have occurred since the invention of the electronic computer. Parallel computing is regarded as so important because it has the potential to improve performance, reduce costs and increase productivity in a bewildering range of potential application areas that span all subjects and cross most disciplines. This long-awaited revolution has finally happened and is still happening! If you are going to jump on to a bandwagon then it is now a good time to do so; the risks are greatly reduced and the relevant skills well within the reach of most geographers.

The basic driving force is the desire and prospect for higher-performance computers so that users can solve bigger and bigger and bigger and bigger problems. Hwang and Briggs (1984: p. 40) go so far as to claim:

Fast and efficient computers are in high demand in many scientific, engineering, energy resource, medical, military, artificial intelligence, and basic research areas . . . Parallel processing computers are needed to meet these demands . . . ***Without using superpower computers, many of these challenges to advance human civilisation could hardly be realised.***

(Author's emphasis)

Well maybe one day, possibly soon, reality will match the hype.

The idea is simple enough. If one parallel HPC machine is not fast enough then turn it into just one node of a few thousand connected by a very high-speed network. This strategy is expected to deliver sustained teraflop-speed hardware soon and petaflop not many years later. The general widely held belief is that 'by the year 2000, parallel computing will be as mainstream as personal computers were in 1989' (Almasi and Gottlieb, 1989: p. ix). This expectation is, indeed, quite reasonable, although 2000 may well be 2010 or later. Parallel processing offers more computing power, more memory and potentially a more natural approach to problem solving by thinking about how to solve problems in a parallel way. There is an argument that parallel problem solving is a more natural approach and that serial computing has for far too long forced programmers to shoe-horn naturally parallel problem solving into a serial format. It is important therefore that geographers start to develop the new computer programming skills necessary to start to make good use of this core technology of the future available now.

So in principle parallel programming certainly sounds like a great idea. Anyone who has ever written a program possessed of any degree of computational complexity is sure to be interested in the idea of running their code on M processors in $1/M$th of the time it would have taken on one processor. Suddenly computer runs on a workstation or PC that could have taken a year (if any users had ever possessed sufficient patience and determination to wait that long) can now be completed in less than a day, and the hitherto impossible computational task becomes a practical everyday proposition. The only problem is how to do it in reality? It is certainly an appealing prospect to have M separate CPUs all busily working on the same problem – just think of all that computing power – but how does it work? How does a simple-minded geographer come to grips with the challenges of parallel processing and parallel programming? One way is to dash out and read several 'Parallel Processing' books written by computer scientists but, unless you are extremely lucky, the effect will almost certainly be the equivalent to a cold shower of sufficient intensity to totally deter waverers and to greatly dampen potential enthusiasts possessed of anything less than a steely determination to persevere. Of course there is a catch. Writing software and designing algorithms that work well on machines with multiple processors inside them need not be trivial or easy. However, the problems can be explained without undue complexity, so read on – you have little to lose!

3.1.1 *What do the words mean?*

Perhaps a useful starting point is to consider what the words mean. There are various definitions of the term 'parallel computing', which all have more or less the same meaning. Parallel processing is self-evident. It is 'only' a matter of getting multiple processors to work simultaneously on your program, which previously had to make do with only one. More formal computer scientist-like definitions are as follows.

Baker and Smith (1996) would agree:

'Parallel computing is the use of more than one Central Processing Unit (CPU) at the same time to solve a single problem' (p. 1).

Chalmers and Tidmus (1996) write:

'Parallel processing is the solution of a single problem by dividing it into a number of sub-problems, each of which may be solved by a separate agent' (p. 2).

Similarly, Chandy and Taylor (1992) explain:

'A parallel program is simply a collection of co-operating programs that together satisfy a given specification' (p. 3).

Almasi and Gottlieb (1989) explain that a parallel processor is

'A large collection of processing elements that can communicate and co-operate to solve large problems fast' (p. 5).

Hwang and Briggs (1984) write

'Parallel processing is an efficient form of information processing which emphasises the exploitation of concurrent events in the computing process. Concurrency implies parallelism, simultaneity, and pipelining' (p. 6);

while Krishnamurthy (1989) writes

'A simple-minded approach to gain speed, as well as power, in computing is through parallelism; here many computers would work together, all simultaneously executing some portions of a procedure used for solving a problem' (p. 1).

The key common point is to note that parallel processing is the solution of a single problem by using more than one processing element (or processor or node or processing element or CPU). This feat can be achieved in various ways: indeed, parallel programming is all about discovering how to program a computer with multiple CPUs in such a way that they can all be used with maximum efficiency to solve the same problem. This is how we would define it and it is good to know that the experts all agree.

However, it is important not to overemphasise the parallel bit, because it is not really all that novel or new! Indeed, parallelism is widely used, albeit on a small scale, in many computer systems that would not normally be regarded as being parallel processor hardware. Morse (1994) writes: 'If by parallel we mean concurrent or simultaneous execution of distinct components then every machine from a $950 PC to a $30 million Cray C-90 has aspects of parallelism'

(p. 4). The key distinction is whether or not the parallelism is under the user's control or is a totally transparent (i.e. invisible) part of the hardware that you have no explicit control over and probably do not know that it even exists. It is only the former sort that we need worry about since it is this which we would like to believe is under our control.

3.1.2 Jargon I

Like many other areas of technology, parallel computing is a subject with some seemingly highly mysterious jargon of its own. Yet it occurs so often that you really do need to memorise at least some of it and either know in general terms what it all means or know sufficient so that you may successfully guess the rest. There are the various words or abbreviations that previously you may have either never come across or never really understood what they mean. You will never be able to join in the small talk at an HPC conference bar unless you master the basic vocabulary and terminology! So here goes.

The term 'HPC' is very easy: it stands for high-performance computing (or 'computer' depending on context), but the definition of what is 'high-performance' is vague, relative and almost constantly changing as hardware continues to improve. It is a characteristic feature that today's workstations now offer levels of performance (or better) than only three–five years ago required extremely expensive HPC hardware in the form of vector supercomputers. According to Openshaw and Schmidt (1997), a Cray J90 (an HPC with a performance typical of early 1990s state-of-the-art hardware) processor is slower and has less memory than a 1997 vintage Sun Ultrasparc 170 workstation.

An HPC can be used to refer to any vector or parallel computer that offers at least one or two and maybe soon three orders of magnitude more computational power and memory than you are likely to get from a top-of-the-range workstation at any given moment in time. So a supercomputer or super power computer is merely an HPC! It is 'super' because compared with 'ordinary' computers it runs much faster and has more memory. The Cray I vector supercomputer ran between thirty and fifty times faster than a large conventional mainframe when it was launched in 1976. Twenty years on and a Pentium II PC now does better than a Cray I for much less than one-thousandth of the cost. Some people talk about high-performance PCs, but do not be so readily tricked. The size of problem and the computational load required by leading-edge scientific research and application also continues to keep pace with HPC developments. The Cray I-based HPC problems located at the rarefied leading edge of science in 1976 could now easily be part of a second-year undergraduate computing exercise! Finally, all things are relative. In 1998, the Cray T3E is probably at least 10,000 faster and bigger than any so called high-performance PC. Such is progress. However, it brings with it a very serious problem: you can no longer run the same program on both the PC and the Cray T3E. The code needs to be changed for the parallel machine. Additionally, the next few orders of magnitude of HPC speed-up will probably come only from increasingly complex parallel

hardware. It will soon no longer be possible merely to sit back and by buying the latest machine every year or two gain a factor of two in performance. Those easy times are nearly gone. You could invest effort in performance optimisation and gain another few speed-up factors, but the real gains will result from going parallel.

So what does MPP or HPP stand for? Perhaps surprisingly, MPP does not mean massively powerful processor, although that is a very good description of what the hardware attempts to deliver. Currently, convention would suggest that a massively parallel processor (MPP) would have tens of thousands of CPUs in a single system box. One view is that these real MPPs, perhaps best epitomised by the Connection Machine (Hillis, 1985), are now extinct, although they could well be revived or rediscovered again at a later date. Instead, current leading-edge machine qualify more as highly parallel rather than massively parallel, emphasising a coarser grain of parallelism. However, these seemingly far more modest machines in terms of the numbers of CPUs they contain are still called MPPs! Additionally, a highly parallel processor (HPP) would probably have between 16–32 and a few thousand individual CPUs (usually a power of 2, such as 1024). Then to make matters worse 'massively' and 'highly' are variable rather than absolute terms. Most people use them interchangeably! Ah well, at least now you know.

'Concurrency' is an alternative term used to describe parallelism, and it can be used interchangeably, depending on your preference. 'Multiprocessing' is another word with a similar meaning. Strictly, it refers to the execution of independent tasks on multiple processors (also called multi-tasking), but then is that not what parallel processing is all about!

Other terms that are often used interchangeably are processor, CPU (central processing unit), uniprocessor, microprocessor node and processing unit or element. It is easiest to assume that they all mean the same thing – a self-contained computer (usually on a single chip) – and ignore the bewildering complexity caused by seeking any more accurate degree of clarification.

3.1.3 Jargon II

More basic terminology follows. Some key definitions of measurements with their abbreviations may also be helpful when attempting to understand the extremes of HPC; see Table 3.1. Typically today, a grand challenge problem in science is currently regarded as one requiring a sustained number-crunching power of over 1 Tflop or 1 teraflop or 10^{12} floating-point operations (e.g. 1 flop = 1 add or multiplication operation) per second and address computer memory spaces of 1 Tbyte or 1 terabyte or 10^{12} bytes of computer memory; or some large fraction thereof (note that 32 or 64 bits, i.e. 4 or 8 bytes, is needed to store a single floating-point number). To put this into perspective, a 64-Mbyte memory workstation (1995 vintage) would be about 100,000 times slower and have a memory space 15,000 times smaller than a 1-Tflop and 1-Tbyte machine. At the time of writing, these Tflop machines do not quite yet exist, but they are

Table 3.1 Some basic definitions and abbreviations.

Abbreviation	*Full term*	*Explanation*
Mflop	megaflop	1 million floating-point operations per second
Gflop	gigaflop	1000 million floating-point operations per second
Tflop	teraflop	1 million million floating-point operations per second
Mbytes	megabytes	1 million bytes of computer memory
Gbytes	gigabytes	1000 million bytes of computer memory
Tbytes	terabytes	1 million million bytes of computer memory

Notes: Some useful prefixes.
mega = a million
giga = a thousand million
tera = a million million

expected soon. It is anticipated that these supercomputers will be highly parallel with a few thousands of processors. The notion of a teraflop (please note: not terror flop as no doubt social theory-rich but data- and computing-free geographers will term it once they discover what the word means!) geography is something that forward-looking geographers should be eagerly anticipating. As GIS moves on beyond data storage and manipulation to proper analysis and modelling, it will need it! The research challenge for the near future is how to use multi Tflop computing hardware as a tool in geography? What problems actually need this amount of computer power? What software development and research can be done with scaled-down problems with current hardware?

3.2 Why parallel processing is important

3.2.1 Driving factors

The motivation for parallel computing is the need for bigger and faster computers. Historically, the impetus came from

1 the need to solve large problems that ran too slowly even on the fastest vector supercomputers; for example, various military, weather forecasting and scientific engineering applications;
2 to tackle problems too large for any other sort of computer; and
3 to provide more cost-effective solutions to problems that could be solved, albeit on more expensive hardware.

In science, this reflects the increasing adoption of a numerical and computational approach to performing experiments as a substitute for laboratory experiments and building prototypes, both of which are either very expensive or too hard to construct. In many areas of computational science the quality of the results (and hence the science) strongly depends on the amount of processor power available. In geography and the social sciences, there are broadly equivalent needs but also some very different ones related to the difficulty of studying and modelling the behaviour of people systems and the growing importance of data mining and other inductive approaches to knowledge discovery. As HPC becomes faster and bigger so new approaches and new geocomputational paradigms will appear (see Chapter 2) that will create new research and application opportunities. Most of which do not yet exist. However, after a while it is no longer clear who is driving what! The quest for faster and bigger computers seems to have developed a momentum of its own. As computer speeds and sizes increase so does their importance, the areas of application increase, and everyone wants more! Most users, it seems, are merely responding to the available opportunities, which are themselves a reflection of previous user needs plus a bit more. While users are undeniably grateful that these ever-expanding hardware opportunities exist, seemingly it is probably not the users any longer who are the driving force. Once it was the military (nuclear weapons design and warhead re-entry dynamics) and the needs of cryptographers (national security monitoring of coded telecommunications traffic). Today, it is still the needs of the military (simulation of nuclear explosions as a substitute for live testing) that is encouraging the US Accelerated Strategic Computing Initiative (ASCI). There are plans for 3+ Tflops systems by 2001 and 10+ Tflops for 2004, compared with current 1998 systems running at about 0.5 Tflops. A most dramatic performance when it materialises. However, there is also an increasing number of scientific applications waiting to use the faster hardware. There is less doubt than at any time previously that there is a real commercial payback from both the basic research and more purely commercial activities. Examples would include the creation of new materials and chemicals, safer planes, car crash testing leading to safer car designs, and data warehousing.

Another crucial factor in the recent history of HPC was the end of the Cold War in the late 1980s. This was initially responsible for the demise of several of the HPC vendors and slowed the pace of new hardware development for a while. However, the quest for speed is seemingly unstoppable, even though the world market for leading-edge machines is surprisingly small. Most advanced countries have only a few! Many commercial organisations are more interested in parallel database hardware optimised for transaction processes and SQL queries than in more general-purpose parallel processors. However, this may be a transitory feature, because the effective use, data mining and modelling of terabyte databases will almost certainly require HPC with a vast number-crunching *and* database-processing capability. At present, these two features are largely independent attributes served by different hardware. Combine them and add a real-time analysis or modelling or fraud prediction/detection component and the potential attraction of holding your terabyte databases in memory on a massively powerful HPC could become more apparent. However, even in a research context the number of HPCs is far smaller than might have been expected. For some reason, science is quite happy to spend £200 million on a single space probe (which may well fail on launch) but find it hard to spend even £20 million on a new HPC? Again for reasons related to legacy thinking and inertia, HPC popularity has still not reached all areas of science. Given the choice between computer experimentation by simulation or experimentation by building hardware, too many scientists still prefer the latter. Critics argue, 'What is the point in watching a computer animation of structures that can be directly observed via some leading-edge microscope?' Well maybe they are missing the point! Nevertheless, HPC is still (amazingly) a minority sport. For instance, the UK's science community has in 1997 one large Cray T3D (three years old), a small T3E, a few Cray J90s and one ageing Fujitsu VPX240. That is really not much. One explanation is that there is still resistance to computational science, another is that in 1997 in the UK, apart from the Cray T3D and T3E, you could probably do almost as well using networks of fast workstations.

The marketplace is still small and congested, yet the quest for faster and bigger HPC seems set to continue unabated. Currently, the race is on for the Everest of the HPC world, the first teraflop machine that really does run useful user code at that speed. These machines and their successors will reopen the gap between workstations and supercomputers and may well encourage a new burst of parallel programming activity in the early years of the next century.

So who or what is the real driving force? We do not think that anyone really knows. After all, we live in an IT age where the pace of innovation and development appears to be almost constantly increasing at a linear or superlinear rate. Geographers, too, are being influenced by an almost perpetual maelstrom of change. Here also the feasibility of new and more computationally intense approaches is entirely dependent on the availability of very fast and very large computing systems. Soon there will be sufficient computing resources to permit the emergence of feasible computational grand challenges in human geography and many social sciences on a par with those of the physical sciences. At first such applications will appear novel and leading-edge, then they will be accepted and taken for granted, and later surpassed by others we cannot as yet envisage in any great detail. Speculations can be made that these may possibly include:

1 modelling more and more aspects of the behaviour and dynamics of people;
2 better spatio-temporal forecasting of socio-economic systems;
3 policy impact prediction on people modelled at the micro level;
4 large-scale locational optimisation of key public and private facilities under environmental constraints;
5 automated monitoring of public health and disease databases for advance warning of problems;
6 better planning models (everywhere);
7 automated real-time analysis of key databases;
8 flexible and safe geographical reporting of personal data;
9 major criminal investigations; and
10 vastly improved financial modelling of national economies.

The only good aspect to note here is that there are no significant amounts of legacy codes to port.

3.2.2 The quest for speed

It is important to appreciate that parallel computing is interesting because it promises faster computing and far larger memories spaces than serial computing and not just because it is parallel *per se*. Indeed, to be sufficiently attractive to justify the effort in users learning new programming skills and in redeveloping their code and algorithms for parallel environments, parallel processors have to be able to offer significantly better performance if they are to attract more than a few users. That point has now been reached.

There are a number of reasons why parallel processing is important from a user's point of view:

1 There are limits to the ultimate performance of serial or single-CPU computers (due to the speed of light, quantum effects, performance bottlenecks in their design), and it is likely these limits may be reached fairly soon (within a decade).
2 There are important problems such as grand challenge problems that require large-scale computation for their solution and that cannot now be solved by any other means.
3 Parallel processing provides a means of attaining faster computing at an affordable price.
4 Faster serial processors merely make an even faster multi-processor machine built from them.
5 Finally, parallel processing is unavoidably the future of HPC.

There is now a growing consensus that parallel processing is an inevitable and, for at least some of us, an unavoidable, technology. It is the future of large-scale computation, at least for the next few decades until other types of non-conventional, perhaps biological or molecular, computing machines can be invented and built.

3.2.3 The inevitability of parallel processing

Morse (1994) offers four reasons for the inevitability of the eventual and widespread use of parallel processing. These have been developed further and are expanded below.

Barriers to clock rates

Crudely put, the speed of a computer depends on its clock speed. These have increased rapidly as microprocessor designs have become more dense. Speeding up has been achieved by putting more components closer together on a single chip. Indeed, over the last decade it has been possible to obtain faster hardware by riding the so-called 'CMOS bow-wave'. In addition, superscalar designs that can deliver two or three arithmetic results per clock cycle, multi-level caches, improved compilers designed for RISC hardware (for example with predictive branching) have provided additional speed improvements of two or three times. These developments combined with faster clock rates have delayed the introduction of parallel machines, because users could gain quite large performance increases without it. The problem is that there are real upper limits on chip clock speeds and on the extent of these improvements likely to be possible with existing technologies. Current technology is getting near to these limits. Once clock rates go well above 1000 MHz (a billion clock ticks per second) then probably the only way of building faster computers is to have multiple processors. It is unlikely that affordable 2000 MHz chips can be mass produced using existing technology without multiple technical breakthroughs, perhaps involving the development of a quantum processor. However, it would be fairly easy to attain 2000 MHz by using two chips, each of 1000 MHz.

System reliability

Parallel processors should be more reliable than large single processors because they use reliable mass-produced microprocessors rather than small-volume highly customised chip sets. Fewer chips inside processors generally increase reliability because there is less to go wrong. Slower clock speeds cause less stress on chip designs, while relatively modest power and less heat result in longer life and greater reliability.

Redundancy in design

This could be used to increase the reliability of parallel machines, although at present the reliability of a machine with 256 CPUs may well be less than a machine with only one because the risk of any one of the 256 developing a fault in a given time period is far higher than the risk that any particular processor will fail in the same period.

Memory bandwidth

This limits the performance of uniprocessors or parallel machines with only a few processors. The performance of a machine depends not just on processor speed but also on memory accessing (read/write) speeds. The problem here is that historically processor speeds have increased far more quickly than memory access speeds. Memory is now a slow device compared with processor speeds and represents a major constraint on processor performances with user code. A simple rule of thumb suggests that typically two words of data from memory need to be accessed for each floating-point operation. As floating-point arithmetic speeds double memory speeds probably need to quadruple! Unfortunately, over the last ten years or so the reverse has been happening, and the situation is becoming worse. In five–ten years time the five–tenfold increase in memory read/write speeds will be more than offset by the ten- to 100-fold increase in processor

speeds. One solution is to parallelise memory access by distributing the memory to the processors. This is an elegant solution but it causes other problems, since each processor can only access its local memory quickly and needs longer to access memory held by other processors. Different machine architectures handle this problem with different levels of efficiency.

Cost–performance is greatly improved

The microprocessors used in many parallel systems come 'free' because they are developed for a mass market and not purely for a niche HPC sector. Indeed, there has been such a vast investment in the development of fast-processor technology because of the emergence of mass markets for them: for example, in games consoles, embedded controllers in consumer products, personal computers, multimedia, etc. For example, it has been estimated that the Pentium and Alpha chips cost much the same to develop as the highly customised Cray J90 chips. However, these development costs are spread over a mass market, producing an invisible but major benefit to builders of highly parallel machines that use these components. Scalability is possible in that more power and/or memory merely requires more processors to be added, while faster individual microprocessors produce even faster MPPs or HPPs.

In short, if we want continually faster computers then going parallel is soon going to be the only way of achieving this goal.

3.3 Highly and massively parallel processing

3.3.1 Building faster machines

From a fairly simple-minded point of view the theoretical total peak power of a parallel machine is found by multiplying the speed of a single processor by the number of processors. A less simple-minded approach would argue that peak Mflops is not a useful measure of machine power, since most user applications will run at only 10 to 50 per cent of peak. Or as one cynic explained, peak megaflops (Mflops) is that rate which the chip manufacturer guarantees will never be exceeded by any user's job! However, let us ignore this distraction and briefly examine the challenge of building supercomputers. Morse (1994) notes: 'In general, massively parallel approaches achieve high processing rates by assembling large numbers of relatively slow processors' (p. 4), which, he could have added, are cheap and mass produced. Consider the problems of building a 1 gigaflop (Gflop) computer. You should know by now that 1 Gflop is 1000 million floating-point operations per second. Table 3.2 outlines some of the alternatives. Note how much easier it is to adopt the multi-processor route because the speed of the individual CPUs becomes less as their numbers increase. This argument assumes that programs can be written that can fully utilise multiple CPUs with 100 per cent efficiency to perform the same amount

Table 3.2 How to build a 1 Gflop computer (= 1000 Mflops).

	Number of processors	*Speed per processor (Mflops)*
serial	1	1000
parallel	10	100
highly parallel processor (HPP)	100	10
	1000	1
massively parallel processors (MPP)	10,000	0.1

of work in a concurrent manner as that performed by a single CPU. In practice, it is not as straightforward as this table implies, because processor speeds also influence the granularity of the parallelisation task. It also assumes that a microprocessor rated at 100 million flops can actually achieve this on code written by a typical user using real data, so maybe it is safe to divide the theoretical peak speed by at least four. However, this does little damage to the general argument being made here.

Consider now the problem of creating a 1-Tflop computer. One teraflop is a million floating-point operations per second. Table 3.3 shows how this could be built via a parallel approach. Note that now a single-processor option is probably no longer feasible. If you are not convinced then as punishment examine the following fragment of Fortran program, which you will not be able to run on any existing computer anywhere in the world today.

```
REAL X (1 000 000 000 000), Y (1 000 000 000 000), Z (1 000 000 000 000)
DO I = 1, 1 000 000 000 000
Z(I) = X(I) + Y(I)
ENDDO
```

Here there are three trillion numbers (1,000,000,000,000) stored in memory (as arrays X, Y and Z) and a trillion adds to perform, which is why it cannot yet be run. Consider now how to build a computer fast enough and with sufficient memory to run this problem in 1 second (which would make it a 1 Tflop machine). Demmel (1996: pp. 3–4) explains it like this:

> The speed of light is an intrinsic limitation to the speed of computers. Suppose we wanted to build a completely sequential computer with 1 Terabyte of memory running at 1 Teraflop. If the data has to travel a distance r to get from the memory to the CPU, and it has to travel this distance 10^{12} times per second at the speed of light c = 3e8 m/s, then r <= c/10^{12} = 0.3 mm. So the computer has to fit into a box 0.3 mm on a side. Now consider Terabyte memory. Memory is conventionally built as a planar grid of bits, in our case say a 10^6 by 10^6 grid of words. If this grid is 0.3 mm by 0.3 mm, then one word of memory occupies about 3 Angstroms by 3 Angstroms, or the size of a small atom. It is hard to imagine where the wires would go!

Table 3.3 How to build a 1 Tflop computer (= 1 million Mflops).

	Number of processors	*Speed per processor (Mflops)*
serial	1	1,000,000
parallel	10	100,000
highly parallel processor (HPP)	100	10,000
	1000	1000
massively parallel processors (MPP)	10,000	100

It is also unlikely that memories of this size and density could be built in the near future (or at all) based on current technology. On the other hand, as Table 3.3 suggests, building 1-Tflop computers is probably within reach now via a parallel processor route. It would certainly appear feasible to think of a machine with 1000 processors operating together to run the code fragment.

So it appears that the immediate future of HPC is therefore unavoidably parallel, and as time goes on so the numbers of parallel computers will increase. There is no alternative unless practicable quantum computers can be built that are capable of solving HPC problems or some other basis for computing is discovered (*viz.* biophysical). In both cases, research and development times would be sufficiently long not to harm your investment in conventional HPC skills. Even then, they will have to be programmed with interfaces to current parallel codes. So learning how to program current parallel machines is essential as well as a safe investment for the future. Those geographers who do so early may well gain significant benefits later on when the long-awaited age of parallel computers finally dawns and a Tflop-processor-powered geography becomes a practical proposition.

3.3.2 A tarnished past

It is a sobering thought that none of this 'the future is parallel' argument is that new. It was fervently believed in by some parallel computer missionaries over thirty years ago when the world's first real parallel supercomputer (the ILLIAC IV) was built (in 1967), albeit forced to use hardware technology that was quite inadequate for the design concepts. Similar claims that the future is parallel were repeated twenty years ago, and it has been fashionable in a few quarters ever since. The problem was that uniprocessor speeds continued to increase, and the physical limits inherent in their design then seemed much further away than they do now. As a result, the immediate need for parallelism turned out to be less than had been expected, while the 'promise' of the technology turned out to be far harder to deliver than many once believed. The dreams of the late 1980s parallel computing salesmen never really materialised. There were also various good excuses for wishing to delay the onset of a parallel programming future. There were, and still are, a number of obstacles. In particular, there is market resistance to MPP even within the HPC world. Quite naturally, users with long or complex codes developed over many years for vector supercomputers are reluctant to have to stop their science while they try to recode them for parallel hardware. They fear cost, delays and the need to invent new algorithms, and until recently doubts about code stability and the longevity of this or that species of parallel computer. It was always easier merely to ask for a new Cray vector supercomputer every three years or so and thus postpone the unthinkable a little longer. There are also perceptions of immaturity and an awareness of the potential dangers of being at (or beyond) the fringe or frontiers of computing does not help. Additionally, there is a track record of failed parallel processing promises, tales of practical implementation problems and also many misconceptions.

Morse (1994) uses the following joke to illustrate some of these problems. He asks 'What is the difference between a used car salesman and an MPP salesman?' The answer is: 'the used car salesman knows when he is lying!' Bell (1994) refers to the four 'flat tyres' of the parallel bandwagon caused by lack of systems software, skilled programmers, good heuristics for the design of parallel algorithms, and good parallelisable applications. Healey *et al.* (1998) added a fifth; namely commercial failures among parallel hardware vendors, which damage user confidence. However, the 1990s has also witnessed an immense maturity of virtually everything in the parallel processing arena, and it is now (in the late 1990s) a much more stable domain. Most of the problems of the past no longer apply, and the cosy world of running old code on new generations of bigger and faster vector machines is about to end. There is also a wider appreciation that an MPP with relatively few processors soon starts to equal a vector supercomputer. Suppose that a two-processor Cray T3E equals a one-processor Cray Y-MP, then a 512-processor T3E is equivalent to 256 Cray Y-MPs and yet costs only a few times as much as a single Cray Y-MP.

Nevertheless, one common misconception continues unabated. This concerns the belief that MPP can deal with only certain classes of highly specialised problem, so the technology is not more generally useful or general-purpose. If this were true then it would, indeed, be of interest only to a handful of users with fairly esoteric problems, and few problems would ever run efficiently. However, this is simply not the case. Morse (1994) notes that: 'Over the past 10 years, MPPs have been programmed – efficiently programmed – for applications in virtually every area of interest to both the research and commercial worlds. It would be far more difficult, in 1994, to list applications not suitable for parallel processing (provided only that the problem size is sufficiently large) than the reverse' (p. 13). Maybe it should be added that it is only since about 1994 (the Cray T3D marked the beginning of a new more mature era of parallel processors) that MPP has really matured and started to deliver the performance it always promised but never quite managed. Also, the gap between what parallel machines can deliver and other types of more traditional supercomputer has started to widen. MPP has caught up and overtaken other approaches to achieving practical and useful HPC at an economic and affordable price.

3.3.3 Ongoing problems

MPP is not without its own very special list of problems.

1 The small installed base and a small market share may make access to leading-edge hardware difficult at present. It is rationed and oversubscribed; for example, it would have been easy in 1996–97 to 'blow' the entire UK social science share of the Cray T3D on a single run of a model breeding machine (Turton *et al.*, 1996).
2 The potentially high cost of porting legacy serial code is another major disincentive. In the UK, the EPSRC's High-Performance Computing Initiative (1994–97) spent over £2 million encouraging the porting of serial and vector codes on to the Cray T3D to try to wean researchers off vector supercomputing. It worked, and this book is one of the unintended by-products.
3 Porting can be lengthy, costly and risky: a task not helped by an earlier lack of parallel programming standards. Indeed, it is only in the mid-1990s that vendor-independent and hardware-independent standards have started to appear, such as the message-passing interface (MPI) standard and also the highly parallel Fortran (HPF) specification, in place of a previous mishmash of vendor-specific libraries and *ad hoc* compiler extensions that previously held back progress towards a real standard and rendered most code non-portable and tied to this or that soon-to-be-obsolete machine.
4 Much existing code is performance-optimised for particular hardware and may need recoding for different machines. When dealing with grand challenge projects or other urgent computing tasks, it is always tempting to sacrifice portability for an extra few percentage points of efficiency. However, there is a growing recognition by industry and academic funding bodies that skilled programmer time costs far more than machine time, even for the biggest supercomputers, and that portability of code is extremely important for those applications that warrant it.
5 Validation of results is important. Parallel processor results should be identical to those produced by the serial code and serial algorithm, but if there are differences then which one is correct? It is often hard to verify or corroborate results that can only be produced by leading-edge parallel machines, as parallel machines can create a whole set of software 'bugs' that exist only in a parallel world. Parallel bug extermination is not always easy or speedy, or guaranteed.
6 Vendor instability is a problem in the field of HPC. There is a history of high mortality rates, and historically there has been a high risk of wasted investment in porting code to yet another about to become defunct machine. Short machine life expectancy is another problem. Most leading-edge parallel machines last three years or less before being made obsolete by advances in technology. The answer is to write portable code and avoid anything that ties you to a specific model of a particular machine, unless of course the application is never, ever again going to be repeated.

7 Where is the geography MPP software coming from? The new parallel software packages that are needed before many geographers can start to make use of the power of the parallel machines that are now available will have to be written by geographical users. This is because only geographers are best equipped to recognise the potential parallelism of geographical problems. There will obviously be a place for computer science-based parallel programmers to do some of the work, but much of the proof of concept work, the awareness raising and the initial developments will need to be undertaken as 'blue sky' research before the rest of the geography and computer programming world will take notice. Hopefully, this book will help with some parts of this task. This section also begs the question, what sort of geography codes (1) actually need HPC and (2) have a large set of potential or real users. One answer is the generic geographical analysis and modelling applications discussed in Chapter 1. Ideally, these facilities need to be accessible over the Internet. Prototype systems already exist; see Openshaw *et al.* (1998).
8 Thinking parallel is not particularly easy, at least initially. Nevertheless, it does present some interesting intellectual challenges and could easily become an obsession. This book seeks to help you gain these particular skills.

The good news is that (1) many of these problems are not relevant in a geographical or GIS context and (2) by being late entrants in this area geographers have unwittingly avoided many of the problems that afflicted other physical sciences that began their parallel processing era a decade or more ago. Parallel programming is now far easier and it continues to become easier all the time. Furthermore, a significant fraction of the scientific HPC community do not write their own code but use a relatively small number of codes to do their science. They use HPC because the software they use has been ported there and they can justify the need for additional computational power by the science they do. As new machines appear so these codes are ported and seem to achieve considerable longevity. Potentially, the same is also true of geography, GIS, and the social sciences. Their own HPC culture could be 'kick-started' by making generic and widely useful code available . Maybe the programs associated with this book will help to do this, but there need to be others as well.

3.4 Examples of thinking in parallel

To run your code on a parallel machine or to write code for a parallel machine you need to think about how the algorithm (that the code represents) can be split into pieces that can be run concurrently. Parallel programming is essentially a divide-and-conqueror approach to programming tasks. Yet this seemingly simple task is made hard because most programmers are trained to think in a highly serial way. In essence, you will need to 'reprogram' your own thinking and problem-solving procedures to become parallel before you can start rewriting

your codes. It is not particularly difficult once you get going. So let us consider some examples of parallel algorithms that may help to stimulate the parallel-thinking neurons in your head!

3.4.1 Example 1: emptying a swimming pool using buckets

Suppose there is a swimming pool that is full of water. Further, let us assume that a single person with a single bucket will take time *T* to complete the task of emptying it. Now imagine how long it may take two persons to perform the same amount of work. They may manage it in time $T/2$, and *N* persons might be expected to do it in T/N. The pool is of a finite size and contains *Z* buckets worth, so would the quickest emptying time be achieved if *Z* people were used? Almost certainly not! It is clearly a parallel task (since in theory each bucket-carrying person can operate independently in parallel), but there are various possible inefficiencies in the system, which means that it is not as 100 per cent scaleable parallel as it may at first appear to be. These inefficiencies may include delays or overheads due to congestion in gaining access to or from the pool, the need to avoid collisions, bucket dropping, people who work at different speeds, difficulty of access to the water in the pool as the level falls and even the capacity of drain into which the buckets are emptied. So there is an upper limit on the number of people who can be used: more people may not necessarily mean a better performance, which could easily reach a peak then deteriorate. The moral of this particular story is that any problem can be divided into only a finite number of subtasks or subproblems, each of which corresponds to the smallest task that can be run in parallel or concurrently without suffering from interaction effects of one form or another. As a result, there may well be an optimal number of subtasks before performance starts to deteriorate. There may also be an optimal size for each of the subtasks. These features recur later.

3.4.2 Example 2: clearing a room containing 100 chairs

This is another parallel task, but here there is an easily defined upper limit of 100 persons who can work on it (i.e. each person would have exactly one chair to carry) unless of course the chairs are particularly heavy. However, in practice the optimal number may once again be far smaller due to congestion delays, lack of manoeuvring space and the bottleneck created by queues in the doorway. Indeed, the optimal number of chair carriers could be considerably smaller than 100, perhaps as small as two. On the other hand, it would be possible to imagine a situation in which the chair carriers might be so well organised as to form a chain or line of chair (or indeed bucket) passers, which would eliminate congestion effects and maximise efficiency. If you can figure out how best to achieve an optimal result by devising an appropriately parallel algorithm then clearly you are well on your way towards being able to think in parallel!

3.4.3 Example 3: not all tasks are parallelisable!

If your definition of a task cannot be broken down into subproblems that can be executed in parallel then it is not parallelisable. Your only hope now is to change the task or alter the algorithm being used to solve it, but even then not all tasks can be parallelised. Consider a classical problem. If one woman takes nine months to produce a baby, why cannot nine women produce one baby in one month? The answer is obvious: it is biologically impossible. However, all is not lost, because nine women could produce nine babies in nine months, giving an average throughput of one baby per month! Indeed, this is one of the simplest types of parallelism. Replication requires no communication between the processors and no contention for resources. Therefore the speeding-up is almost exactly N times, albeit with a latency of about nine months before the first 'result' is obtained!

3.4.4 Example 4: building a skyscraper with 500 floors

Now consider how you might build the 500 floors of a skyscraper in parallel. Clearly, the obvious answer of building each floor at the same time is impossible, since you cannot start work on the second floor until you have at least completed the structural frame for the first floor. One answer would be to build the floor structure first and then finish all 500 floors in parallel. Another alternative would be to build the skyscraper on its side and then lever it upright when it is finished! Maybe this is an example of the sort of extreme lateral thinking you may need in order to produce parallel solutions to some hard-to-parallelise problems! There are many problems of skyscraper type where there is a data dependency. This can cause serious problems for parallelisation attempts, and this is one of the reasons you should not expect to see a compiler that does all types of parallelisation automatically for you in the near future. You will probably always have to do most of the really hard algorithm parallelisation design work yourself, since only you may be smart enough (at present) to rethink the problem or invent a new design for the algorithm that the program implements in such a way that all data dependencies are removed or at least greatly reduced. Yet experience suggests that even seemingly strongly serial algorithms can be redesigned as parallel ones, although it is important not to increase (by too much) the amount of work being done in order to achieve parallelism. The parallel simulated annealing method used in zone design is a good example of this type of successful redesign; see Openshaw and Schmidt (1996).

3.5 Can parallel machines ever be used efficiently?

Another potentially very serious constraint on the performance of parallel computers is provided by Amdahl's law. Amdahl (1967) claimed that it is impossible to use large numbers of processors efficiently on a single problem. His argument is, or was, frequently used, particularly by serial processor vendors, to prove that

there was no point in using hundreds of processors to solve a problem. Amdahl was the designer of the IBM 360 series of mainframe architecture. Later he went on to design the Amdahl range of IBM plug-compatible mainframes, so he had a strong vested interest in serial technology and maybe this needs to be taken into account when examining his arguments. He writes, 'For a decade prophets have voiced the contention that the organisation of a single computer has reached its limits and that true significant advances can be made only by the interconnection of a multiplicity of computers in such a manner as to permit cooperative solution . . . Demonstration is made of the continued validity of the single processor approach and of the weakness of the multiple processor approach!' (p. 483). It is the sort of calculation that could have been done on the back of a proverbial beer mat but, nevertheless, it is still a very important criticism of parallel computing and deserves closer scrutiny.

In measuring the performance of parallel systems, there are two key concepts to consider: one is speed-up and the other is efficiency. Speed-up is simply defined as

$$T_1/T_N$$

where T_1 is the time taken to run a program using only one processor, and T_N is the time taken by N processors working together in a parallel computer. In the ideal situation, as N increases, the so-called elapsed or wall clock time T_N should decrease by a factor of N. It is important to observe that wall clock time is the elapsed time since the start of the run. It is important not to confuse wall clock time with the total processing time. This is the amount of time each processor was busy processing, and on a multi-processor machine this could be anything up to N times the wall clock time. However, let us not dwell on total processor time, since after all this is why you use an MPP. It is the wall clock time that really matters. This is also the time metric in which humans live. So be careful always to disentangle the total processor time from the total elapsed time and, if you use only the latter, then you will avoid getting stuck in a time warp!

Scaleability is the speed-up of a program as the number of processors being used increases. In a problem with good scaleability there is a roughly linear relationship between elapsed time and the number of processors being used. Good scaleability implies an efficient use of processors, and this is the dream of all parallel algorithm designers. It may sometimes be possible to achieve a seemingly unlikely superlinear speed-up due to either subtle changes in the efficiency of memory accessing or a more efficient performance of an algorithm as more processors are introduced. In practice, few algorithms achieve linear scaleability; most reach an optimal level of performance and then deteriorate (sometimes very rapidly). The challenge is to design parallel algorithms that are able to exhibit linear speed-up now and on machines not yet built. Dream on! Note that

$$T_1/T_N = N$$

is the ideal case. Typically T_1/T_N will be less than N because of various inefficiencies and less than perfect parallelisation of the code. Nevertheless, this reduction of wall clock time as the number of processors increase is the principal reason why parallel processing is so important. Efficiency is the actual speed-up on N processors compared with the ideal maximum. Hence

$$\text{efficiency} = \text{actual speed-up} \div \text{ideal speed-up}$$

which is the same as saying that efficiency equals the speed-up on N processors divided by N. Amdahl (1967) caused a major stir because he suggested there is an upper limit to the speed-up that can be obtained which is independent of the number of processors being used. This independence aspect (if it occurs in practice) indicated that parallel processing has no future as it means that it is not possible to speed up an application indefinitely by using more and more processors. Amdahl's concern is that as more and more processors are used the machine will be used less and less efficiently. His argument is essentially that the total run time of a program on a single processor can be divided into two components:

1 a non-parallelisable or serial part (T_S) and
2 a parallelisable part (T_P).

The total time on one processor is therefore

$$T_1 = T_S + T_P$$

For N processors it is

$$T_N = T_S + (T_P/N)$$

It is only when you plug some numbers into these equations that the really potentially devastating nature of Amdahl's law becomes apparent. If five processors are used (N = 5), and suppose that $T_S = 5$ and $T_P = 20$, then

$$T_5 = T_S + (T_P/5)$$
$$T_5 = 5 + 4$$
$$T_5 = 9$$

So the total wall clock time for five processors is nine units, compared with 25 if only one was used. The speed-up is T_1/T_5, which is 25/9 or 2.67 times, whereas the maximum possible would have been five times. The percentage efficiency is $100 \times 2.67/5$, or 56 per cent. In other words, the machine is being used for only 56 per cent of the time, and for the remaining time it has four idle processors. Now suppose we can increase N towards infinity, then efficiency goes to zero and maximum speed-up gets stuck at around five times since the serial time remains constant and totally dominates the run time. Hence

$$T_{1000000} = T_S + 0$$
$$= 5$$

Adding more processors cannot reduce T_S, and this ends up as the time taken by the total program. It is common sense, but if it happens in practice it is quite devastating. Amdahl (1967) observed that for typical Fortran codes he looked at in the mid-1960s, the T_S fraction was about 40 per cent and that this has probably been constant for about ten years. Moreover, he argued that it is highly improbable that this serial part could be reduced by more than a factor of three as it appears to be sequential and hence unlikely to be amenable to parallel-processing techniques. He reasoned, therefore, that 'the effort expended on achieving high parallel processing rates is wasted unless it is accompanied by achievement in sequential processing rates of very nearly the same magnitude' (p. 483). This is probably an impossible task, and based on this assumption it can be argued that parallel processors with more than a small number of processors can never be used efficiently.

Even worse follows. You can also compute the maximum speed-up for any given values T_S and T_P. If T_S is 75 per cent, the maximum possible speed-up is four times regardless of how many processors you use. Now suppose that T_S is only 1 per cent and $N = 1000$, then the maximum possible speed-up is less than 91 times! If T_S is 99 per cent and N is 10,000 the maximum speed-up is still only 100 times and the efficiency is 0.01 per cent; in other words, most of the 10,000 processors would be idle most of the time. What a waste of a possibly expensive resource! In other words, according to this calculation you cannot use more than a smallish number of processors efficiently, and it implies that an MPP with hundreds or thousands of processors would be hopelessly inefficient and hence a waste of time ever developing.

This was a deeply depressing result at that time which almost certainly set back MPP developments by at least as few weeks! The impact was less than expected because there was not much MPP in the 1960s! It is now accepted that Amdahl was in fact quite correct from a historical point of view. In the late 1960s, problem sizes were too small for MPP and, had it existed, it could not then have been used efficiently. However, there was a flaw in one critical assumption. The ratio of T_S to T_P is not constant even for the same program and usually varies with problem size. Typically, the parallel or T_P part grows far faster than the serial or T_S part. So if the T_S part can be made an arbitrary small fraction of T_p by making the problem size bigger, then Amdahl's law breaks down. It is this 'loophole' that parallel processors exploit. Instead of using more and more processors to solve the same size of problem you only need to either increase the problem size for a fixed number of processors or else expand the problem size as the number of processors increases so that the serial fraction will tend to zero, offering linear speeding-up with increasing numbers of processors. This is also an equally dramatic and common-sense result.

Consider another example. The serial part could be regarded as the time taken to read data into a program, although it could also be reading–writing to memory. Let us assume that this depends on N (the number of cases or zones). However, in most models that geographers apply the computing activity in the potentially parallel part increases as N^2, and it is this which rapidly dominates computing times. Who cares if reading the data takes 1 second for ten cases and 1000 seconds for 10,000 cases when the computing time goes from 1 second to 1 million seconds! When Amdahl was writing down his 'law', memory space limits would have restricted the number of cases that could be considered to a domain where parallel processing would not have been worth while. The prominence given to Amdahl's law by many vendors of serial hardware and the degree of annoyance felt by many vendors of parallel machines can still be seen in their obvious delight in reporting to users how much better they are doing than Amdahl's law would suggest.

A revised Amdahl's law was later produced by Gustafson *et al.* (1988). He too argued that Amdahl's assumption that problem size was independent of the number of processors was incorrect. Instead, problem sizes tend to increase with the number of processors, in which case the upper limit on efficiency is linear with the number of processors. According to Gustafson's law, maximum speeding-up is now

$$(T_S + (T_P \times N)) / (T_S + T_P)$$

or

$$N + (1 - N) \times T_S$$

This suggests that an almost linear speed-up with increasing numbers of processors is possible *if the problem size increases with the number of processors*. So MPPs only need large problems to be used efficiently, which is one of the main justifications for using parallel processing in the first place. It is only necessary to ensure that the non-parallelisable part is a minuscule fraction of the total time. On suitable problems, the non-parallelisable part grows slowly compared with the parallelisable part, or if it does not then it is this criterion which you have to design scaleable parallel algorithms to meet. The secret, if there is one at all, is to ensure that there are no serial bottlenecks in the algorithm. This is partly a matter of good design backed up by profiling of the code to find any serial bits, which then need to be removed by either changing the code or modifying the algorithm.

So a key consideration is therefore whether 'your' problem is big enough to be appropriate for MPP or whether the problem size can be increased as the number of available processors increases. In geography, there are good grounds for believing that in general this will usually be the case, due to:

1 the spatial data explosion of the 1990s, which dramatically increased problem sizes by the simple expediency of producing more data at a finer level of spatial, temporal and attribute resolution (well, that is the essential defining characteristic of a spatial data explosion!);

2 an interest in computationally intensive modelling and simulation procedures that can be scaled up as computer speeds increase and more data become available, thereby promising to improve the quality of the results; and
3 a growth of interest in a geocomputational paradigm for doing all kinds of geography.

3.6 Building a wall in a parallel way

Now let us return to parallel thinking mind exercises. Wall building is often used as a metaphor for explaining parallelism (Fox, 1988), although ditch digging would be just as good. Wall building is a traditional craft that can and has often been parallelised in practice, perhaps without those involved ever realising it because the parallelism is natural. The parallel sections in wall building are in the middle of a stretch of wall and the serial parts are at the ends of the parallel parts. Increase the number of workers and the time needed to build a wall decreases. For example, if one worker can build 1 yard of wall per day and the wall is 73 miles long and it takes three days to do each end section, then using the arguments from the previous section:

$$T_1 = T_S + T_P$$
$$T_1 = 6 + 126{,}480 \text{ days}$$

In other words, it would take one worker 126,486 days to complete the task.

Now let us use 1000 workers:

$$T_{1000} = T_S + T_P/1000$$
$$T_{1000} = 6 + 127 \text{ days}$$

and it takes only 133 days, giving a speeding-up of 951 times with an efficiency of 95 per cent.

Now let us try 10,000 workers:

$$T_{10,000} = T_S + T_P/10{,}000$$
$$T_{10,000} = 6 + 13 \text{ days}$$

The wall now takes 19 days, producing a speeding-up of 6657 times with an efficiency of 67 per cent. However, note that one-third of the total time is now being expended on the T_S end sections of the wall (e.g. 6 days of the 13), which is serial and can only be done by a single worker no matter how many may be employed on the rest of the wall.

So, wall building is a useful metaphor to demonstrate parallelisation effects. There is no doubt that wall building is a parallel process and that the duration of the building work depends on the length of the wall and the number of workers being used. Note that here the ratio T_S/T_1 is small, suggesting a maximum

Table 3.4 Time taken to build a wall (in days).

Number of workers	*Length of wall in miles*			
	1	*10*	*50*	*100*
1	1766	17,605	88,006	176,006
100	24	182	886	1,766
1000	8	24	94	182
5000	7	10	24	42
10,000	6	8	15	24

Source: based on Morse (1994: p. 41).

Table 3.5 Speed-up times for wall builders.

Number of workers	*Length of wall in miles*			
	1	*10*	*50*	*100*
1	1	1	1	1
100	73.6	96.7	99.3	99.7
1000	221	734	956	967
5000	251	1760	3667	4191
10,000	294	2200	5867	7334

potential speed-up of 21,000 times! Table 3.4 shows the times taken for a range

Table 3.6 Wall building: percentage efficiency.

Number of workers	*Length of wall in miles*			
	1	*10*	*50*	*100*
1	100	100	100	100
100	74	97	99	99
1000	22	73	94	97
5000	5	35	73	84
10,000	3	22	59	73

of different wall lengths and numbers of workers. It is nicely parallel. Table 3.5 shows speed-up rates, and the associated levels of percentage efficiency are shown in Table 3.6. However, efficiency is still not 100 per cent, because of the serial sections at the start and end of each piece of wall, but note how it increases both with the length of wall and with a reduction in the number of workers.

On the other hand, if you can build a 100-mile wall in 24 days with 10,000 workers, is that not very useful if the alternative is to wait 482 years? Likewise, who really cares if you can build a 10-mile wall in 8 days even at 22 per cent efficiency, compared with 48 years at 100 per cent? Does efficiency really matter

that much? When the enemy is approaching, is it not more important to get the wall built as rapidly as possible by throwing vast amounts of effort at it regardless of the levels of efficiency displayed? If so, do the same arguments also apply in a scientific computing context? After all, let us be rational here. An idle processor in an HPP machine probably has a total capital cost of only a few thousand pounds, not several million. So who cares if some are idle? The ability to solve massive problems quickly is itself a very important practical benefit that far outweighs more theoretical considerations relating to less than optimal levels of performance or efficiency.

3.7 A brief history of parallel computing

With the benefit of hindsight, it is amazing that Amdahl's law did not kill off MPP completely! However, it did not do so probably because few understood the implications and because in the 1960s the entire subject was still at a very primitive stage of experimental development. MPP was at least a decade or two away from becoming a useful tool rather than a computer science plaything. Nevertheless, the idea of parallel computing is not new! ENIAC, possibly the world's first general-purpose digital computer, was conceived as a highly parallel machine with twenty-five independent computing units in 1946. However, parallel machines are harder to build and program than machines with a single CPU, so it is hardly surprising that single-processor technology became dominant and that most computer algorithms were written for single-processor CPUs.

Lewis and El-Rewini (1992) tersely but neatly summarise the history of computing as follows:

1 the dawning of the age in the 1950s
2 the age of mainframes in the 1960s
3 the age of minicomputers in the 1970s
4 the age of personal computers in the 1980s
5 the age of parallel computers in the 1990s

To this may be added another step that is still unfolding:

6 the age of clustered parallel processors in the 2000s.

This basic sequence of computing developments mainly reflects advances in the hardware used in the building blocks, from valves (1940–1950s), diodes and transistors (1950–1960s), small and medium-scale integrated circuits (1960–1970s), and very large-scale integrated devices (1980s onwards). Increases in speed and reliability coupled with dramatic reductions in cost and physical size have greatly increased computer performance, particularly during the 1990s. This has been coupled with the repeal of Grosch's law (Ein-Dor, 1985) which, simply put, states that the best price/performance ratio is to be achieved by using the most powerful uniprocessors. However, you should note that right now hardware belonging to all but the earliest of these periods of development still survive in one form or another. Indeed, mainframes have (perhaps perversely) become fashionable again in the mid-1990s as file servers for very large numbers of users, to preserve legacy code worth billions, and because of falling hardware costs and improvements in throughput and performance.

Table 3.7 gives some landmark historical events in the history of parallel computing. The story is still running. If you wish to join in or contribute to it then you need to be able to write parallel programs.

Table 3.7 Some landmark historical events in HPC.

1967	ILLIAC-IV: a 4-Mflop parallel machine with 64 processors, each with 2048 words of memory
1969	CDC 6600: first pipelined computer
1976	Cray 1: first vector supercomputer, 160 Mflops peak
1977	C.mmp: 16 processors and 2.5 Mbytes of memory
1979	ICL DAP: a 1024 single-bit array processor
1981	BBN Butterfly with 256 processors
1981	C^3 p hypercube
1981	DEC VAX 11/782 two-processor machine with shared memory
1982	Alliant
1982	Cray X-MP vector supercomputer with four processors
1983	Sequent
1983	Goodyear MPP, a 132 × 128 1-bit array processor
1985	Meiko Computing Surface based on transputer
1985	First Teradata database computer
1986	TMC CM-1 64K 1-bit processors
1987	TMC CM-2 64K 1-bit processors
1988	Cray Y-MP vector supercomputer with sixteen processors
1989	Fujitsu VP200 vector supercomputer with two processors
1990	Aliant FX/2800 with 28 i860 processors; Mass Par MP1 with 16K 4-bit processors, each with three pipelines
1991	CM-200
1991	KSR1 with 32 processors, shared memory
1992	TMC CM-5 with 1024 Sparc processors
1993	Cray T3D-2048: DEC Alpha processors
1996	Cray T3E-2048: faster DEC Alpha processors
1998	US ASCI systems for Tflop computing

3.8 Conclusions

Parallel processing is conceptually easy to describe; it is just a matter of subdividing a task into a number of unrelated subproblems. The subsequent implementation on a multi-processor machine is not always (if ever) straightforward, but it is becoming easier. Most of the difficulty is due to unfamiliarity with the programming techniques that are now needed to exploit the hardware. Also, it is important to try to make do with whatever real or virtual parallel system you happen to have access to. Not all, or many of them require major investments of many millions. You can create your own virtual parallel supercomputers using networked workstations or PCs you already own or have access to. These virtual machines can be run at night when they would otherwise be idle. Morse (1994) notes: 'As the knights of the MPP pursue their quest for the Holy Grail of Tflop computing, the most attractive potential commercial market for MPP at the low end, goes largely untended' (p. 23). Remember though that parallel processing is *scaleable* computing. Code and algorithms that scale up to more and more processors also scale down, if properly coded.

According to Wilson (1995: p. 497), the future of massively parallel computing depends on how two key questions are answered:

1 Can large parallel computers be made cost-competitive compared with networks of workstations?
2 Can parallel programming be made time-competitive with sequential programming?

In some ways, the first question is irrelevant provided that your code has the right level of parallelism. You can write code that works well on both; indeed, this should be your aim right now! The second question is a matter of language, standards, software tools, program generators and compiler development. In a geographical context, there is also a third question. Who or which geographers are going to do it and take the initiative in this area of future critical technology? Parallel processing is here to stay, but how many geographers know this as yet or are prepared to do anything about it?

4 Types of parallel-processing hardware and programming paradigms

This chapter takes a closer look at the nitty-gritty of HPC hardware. It is not designed to convert the reader into a computer scientist and covers only the bare necessities. However, we are convinced that the general reader will want to know a little more about the hardware, which affects how HPCs are programmed, but most of the computer science can be skipped as it is irrelevant. If you are not interested in any of this then skip the chapter. If you know you are only interested in a particular type of hardware, then go straight to the relevant chapter; *viz.* Chapter 5 for vector supercomputers, Chapter 6 for shared-memory machines, and Chapters 7 and 8 for distributed-memory multi-processors. If you are confused, then maybe reading this chapter will help.

4.1 Automatic parallelisation software

This chapter briefly describes and discusses the different types of parallel computer in an abstract way. It is designed to help you to understand the general nature of parallel-processing hardware. It may help you to make decisions as to what type of programming paradigm to adopt, as this usually depends on the broad class of parallel machine that you intend to target, although this is far less relevant than previously. Now you might think that none of this computer hardware detail is necessary at all, because your serial code can be automatically transformed into a parallel form via a clever compiler or by automatic parallelisation software that hides all the low-level detail and complexity. Indeed, it would be really nice if there existed an automatic parallelisation program that took in serial code and magically produced efficient parallel code targeted at any specified machine architecture. At a stroke, all existing legacy serial code worth billions of ECUs in replacement costs could then run unchanged after recompilation and you could avoid completely the need to develop any specific parallel programming skills. However, it is unlikely that this will ever work well other than on the simplest of problems, because parallelisation involves far more than simple code or language translation. The problem is so complex because it is not just the code that needs to be parallelised but the logical intent of the underlying algorithms. Even if automatic parallelisation tools could be developed, it may always be easier and more efficient to write new parallel algorithms

as parallel code without first forcing them through a serial filter. It makes absolutely no sense whatsoever to write new code for a parallel computer in a serial form so that it can subsequently be parallelised by some clever software! Let us be honest. Code for parallel machines really needs to be designed for parallel execution. This is the essential challenge of parallel computing.

So the initial difficulty in writing parallel code results from the fact that often it is the algorithm and not just its representation in serial code that needs to be parallelised. The algorithm may also need to be changed to suit the type of parallel hardware on which it is to be run, although (if practicable) this is best avoided. Indeed, there are three key components that can interact: (1) knowledge of the application, (2) the nature of the algorithms used, and (3) the computer science aspects of its hardware implementation. Very few people will be equally knowledgeable about all of them. If geography codes are going to be parallelised successfully then geographers are going to have to do it themselves by learning sufficient of the computer science to be able to cope, because only they will have a good and detailed knowledge of the algorithms and the nature of the problem being tackled. This application knowledge becomes even more important when existing serial algorithms have to be re-expressed or redeveloped or reinvented in a parallel form. You have to fully understand the serial code and its implicit algorithms before you can properly parallelise it. It is not simply a translation process, except in the most trivial cases.

On the other hand, not all serial algorithms are in a highly non-parallel form. It is these which present fewest problems, but even here you need to know enough about parallelisation to recognise that this is the case. It is important not to be lulled into a false sense of parallel happiness. For instance, merely because a compiler will identify many parallel loops in your code does not mean that you cannot help it to do many times better by subsequently rearranging your loops in an even more efficient way. However, before you spend six weeks experimenting with different code changes, first be sure that the effort is going to be worth while. A reduction of 10 per cent in computing times is not worth while. Equally, a reduction by 500 per cent in code that will never be run again may not be worth while if time taken for the code changes far exceeds the expected elapsed time of the run on your workstation. A good rule of thumb is that you parallelise only those problems where the benefits far exceed the pain!

The good news is that it is not hard to develop the necessary parallel programming expertise. Also, in some cases it is possible to write an elegant parallel solution to a problem that works better and runs faster than a serial algorithm as well as being easier to understand. A triple bull's eye! Indeed, this is one of the nicest aspects of parallel processing. The single-processor machine is only a special case of multiple-processor hardware. However, code designed for the former seldom works well on the latter, but code designed for multiple processors often works well on a single processor, provided that it is written in a sufficiently general parallel way rather than for this or that oddball architecture. Computer scientists can easily become their own worst enemies here, and geographers would be well advised to try to avoid the same pitfalls. Here is some generic advice. If you are going to invest three months of your life programming a parallel machine instead of relaxing, playing cricket or drinking, then you should ensure that (1) the problem is sufficiently important to justify the effort; (2) will not require another three months reprogramming in one year's time when a different lump of HPC hardware comes along; and (3) that it will work.

It is useful therefore to review briefly the principal computer architectures on which parallel programs can be run. It is often important to know a little about these aspects, because it affects the programming paradigm you adopt. It is interesting, or it can be in a 'New Scientist' sort of way, if you are at all curious as to how these machines work. Maybe it will also make you feel better that you understand more of what you are doing. However, you really do not need to know all the micro techno details about how the hardware actually works before you can start to program it! Few geographers know much about how a PC works, so let us leave 99.9 per cent of the internal workings of parallel hardware to the computer scientists and engineers and get on with the far simpler task of using it in geography. After all, no one will ever expect a geographer to be able to design or build a parallel computer, but it is reasonable to expect that we can program them so that we can start to make good use of them. If you are uninterested in computer hardware then why not skip this chapter. The authors would claim to have (between them) twenty years experience of programming HPC hardware of various types and antiquity without knowing much about the hardware. Indeed, recollects one of the authors, if he had realised that a Cray I only offered at best a tenfold speed increase back in 1982 then he might never have bothered to use it and would possibly have become a much better cricketer!

4.2 Computer architectures

Most parallel processing and programming books are what can only be described as obsessed with the physical details of hardware and how they work. Sadly, most of what they describe is obsolete trivia that are often of only historic interest by the time the book is published! Geographers need to know something but not everything, and much of the detail is frankly of interest only to the historians of computer science rather than of any great practical utility to a programmer. Nevertheless, a very useful way of developing a general level of understanding is by classifying the different computer hardware types as a means of simplifying a fairly complex situation.

4.2.1 Flynn's classification of hardware

A good starting point is the old Flynn classification of hardware types: virtually every book on parallel processing refers to it. Flynn (1972) devised a simple taxonomy of computer hardware that is still widely used, even though it is oversimplified. He classified computers according to whether they process single or multiple data items and whether the same or different operations are being applied to each item. It is important to appreciate that most modern computers are more complex than Flynn's taxonomy would indicate, and many may have elements of two or more types of hardware in them somewhere. Nevertheless, it is still helpful to identify the link between hardware and programming paradigm. Table 4.1 gives Flynn's taxonomy and Table 4.2 shows some of the more common hardware labels often used for the four basic architecture types.

Table 4.1 Flynn's taxonomy.

Number of data streams	*Number of instruction streams* single	multiple
single	SISD	MISD
multiple	SIMD	MIMD

Table 4.2 Some common hardware labels.

Number of data streams	*Number of instruction streams* single	multiple
single	Von Neumann machine	pipeline computer
multiple	vector and array processors	multiple-processor parallel machine

An **SISD** (pronounced 'siss-dee' or in full) is an abbreviation for **S**ingle **I**nstruction applied to a **S**ingle **D**ata computer. This is the conventional von Neumann computer with a single processor or CPU which executes instructions sequentially, e.g. a PC or workstation or conventional uniprocessor mainframe. There may still be elements of parallelism hidden away inside this hardware (e.g. pipelining of arithmetic operations or overlapping of serial instructions), but true parallelism is not supported, and more significantly, none of this invisible hardware parallelism is controllable or directly accessible by the user. However, as Dowd (1993) points out, knowledge of some of this invisible parallelism can be used to improve single-processor performance dramatically. This is important, because modern processors are very finicky. Pipelines do not always work, and the fancy tricks designed to help the memory to keep up with the CPU can be easily defeated by an accidentally clumsy bit of code. Suddenly, the code optimisation and memory optimisation tricks once employed on top-end vector supercomputers become relevant if you want to squeeze maximum performance out of your home PC or workstation or even a single processor on an MIMD (see later for a definition) parallel machine.

An **SIMD** (pronounced 'sim-dee', ideally spoken with a real or false US accent) is an abbreviation for a **S**ingle **I**nstruction applied to **M**ultiple **D**ata computer. This type of computer applies the same instruction (or computer operation) to multiple items of data simultaneously. This may appear to be highly restrictive and of little value, but it has been found to be a useful paradigm for many scientific problems that perform lots of arithmetic on a regular set of data. SIMD machines operate in a synchronous manner. Typically, there is a single instruction stream that is acted upon by many different processors in a strict lock-step fashion, so that each processor does exactly the same instruction on a different piece of data, or is told to do nothing. This can be visualised as the computer equivalent to soldiers being drilled by a sergeant major, or dancers in an aerobics session following instructions from their trainer. The entire troupe does exactly what they are told concurrently. It is not possible for any of the processors to carry out different instructions at the same time. This is a very fine-grain type of parallelism that has historically operated at the level of individual instructions or arithmetic operations. The vector supercomputers of the 1980s and later are SIMD machines, as indeed were the so-called 'array processors'. This form of limited parallel machine works well, and machines of this type have been at the forefront of HPC for about two decades and have only recently been superseded by other types of parallel hardware.

An **MISD** (pronounced 'miss-dee'), which stands for **M**ultiple **I**nstructions applied to **S**ingle **D**ata computer, is a somewhat strange technology. It would require a number of processors to perform different operations on the same piece of data at the same time. We found it hard to figure out why you would want to do this unless you are a computer scientist interested in weird computing! It is a highly specialised and seemingly a very restrictive form of parallelism that is often impractical, not to mention useless, as the basis for a general-purpose machine. If a real MISD machine can be built, maybe its only role in the foreseeable future will be as some kind of computer engineer's plaything, but maybe we are biased (or just plain computer ignorant)!

An **MIMD** (pronounced 'mim-dee') stands for **M**ultiple **I**nstructions applied to **M**ultiple **D**ata computers. This type of machine has multiple processors that execute their own programs with their own data. Communication between the processors allows them to co-operate in the solution of a single problem. This is the most general form of parallelism yet discovered. The processors no longer have to operate together in a highly synchronised global manner but instead local synchronisation is imposed only when required during the running of a program; for instance, a processor will not be allowed to start the next loop in the program if it requires results from the completion of the previous loop by all the other processors. This allows each processor to work on problems of uneven load without having many idle processors if the programmer is sufficiently clever in the structuring of the task. It is this sort of computer that is currently the dominant HPC parallel-processor technology. Note though the potential for confusion that can arise here. An MIMD machine is typically built from SISD components, and some even utilise SIMD processors! Maybe this is a reflection of the generality of the MIMD concept. Presumably MIMD could one day, fairly soon, also become the dominant HPC workstation architecture, with the

difference between the real HPC and the not so HPC ends of the spectrum being reflected in the number of processors being used.

4.2.2 *Classifying parallel machines by how they access memory*

Another way of classifying parallel machines is by the way their memory is accessed by the processors. Treleaven *et al.* (1982) classify MIMD designs distinguishing between shared memory and private or distributed memory. This is a very important practical distinction and is repeated by Morse (1994), who argues that while a top-level architecture discriminator is between SIMD and MIMD there is a very important distinction between shared-memory (SM) and distributed-memory (DM) hardware, which has considerable practical relevancy to programming and efficiency.

A shared-memory machine is made up of a number of separate processors, each of which shares the same global memory of the machine. Data stored in any part of the memory is instantly available to any processor. Figure 4.1 gives an illustration of what this looks like. Shared-memory systems are the easiest to program, but there are currently limits on the number of processors that can be used because of memory contention problems.

A distributed-memory machine is composed of many processors (or processing elements – PEs – or nodes), each of which consists of a processor and some local memory. The processors are connected together by very fast internal communications network. However, a processor can only directly and quickly read and write to its own local memory. It can access the memory belonging to other processors, but only by asking for it via an explicit message; see Figure 4.2 for a

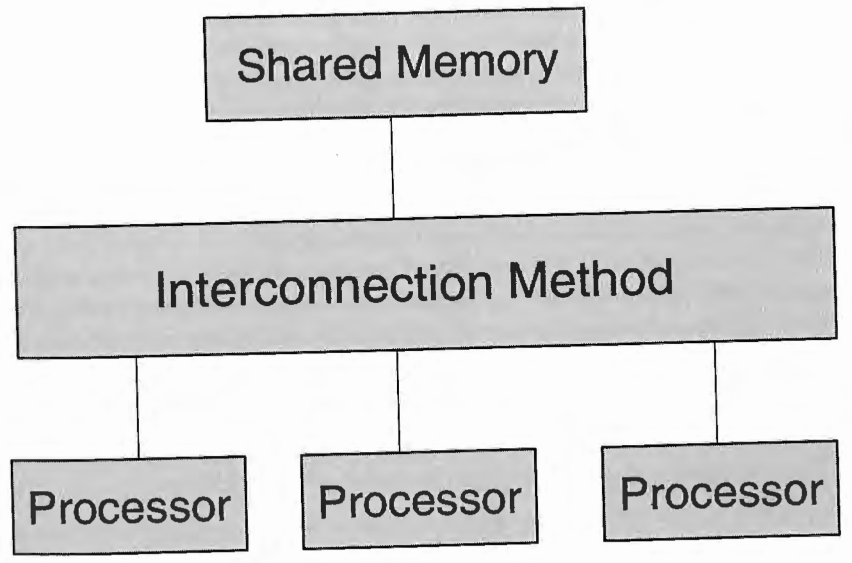

Figure 4.1 A shared-memory machine.

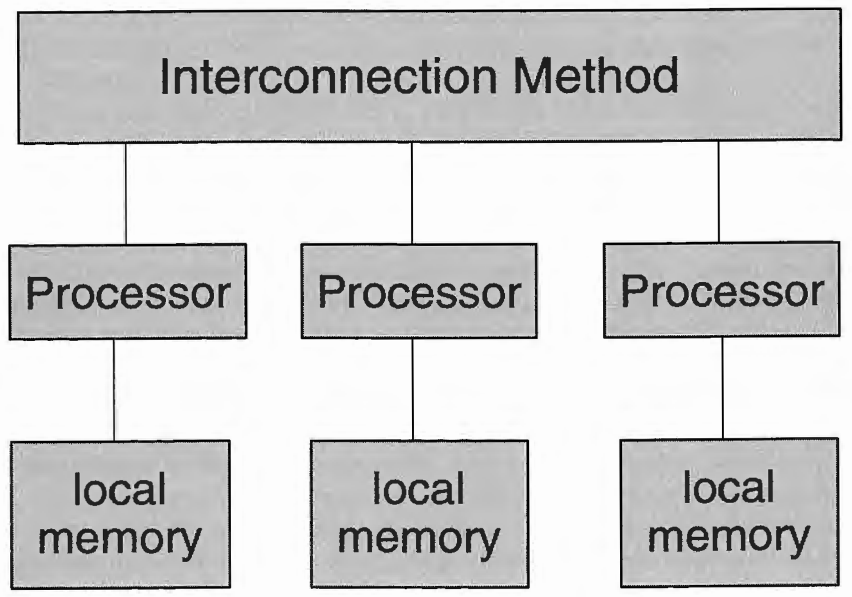

Figure 4.2 A distributed-memory machine.

simple diagrammatic representation. This is the origins of the term 'message passing'. However, the time taken to access non-local memory depends on the locations of the communicating processors in terms of the interconnection network, the size of the message and the bandwidth of the interconnector. However, it is important to note that the distributed-memory approach is the more general paradigm for parallel computing because the technology scales (in theory at least) to almost any number and types of processor. The cost is that it is far harder to program, since the communication of data and results between processors has to be explicitly handled by the software that YOU have to write. By contrast, the global shared-memory hardware (Figure 4.1) approach is far easier and simpler because much of the complexity is invisible to the programmer (although it is still there behind the scenes).

At the risk of further confusion, it may be worth pointing out that a single-processor SISD computer also has a form of interconnection network linking its CPU to memory (typically termed a bus or channel interface). The multi-processor version merely has more processors hung on this internal network, which runs much faster and has greater bandwidth.

4.2.3 *Classification by number of processors (or CPUs or PEs or nodes)*

Parallel hardware can also be categorised by the numbers of processors it contains: There are a number of possibilities:

1 one processor means that it is not a parallel machine;
2 parallel hardware (PP or parallel processor) with small numbers (two to 32) of typically very powerful highly customised processors with shared memory and increasingly vector processing capabilities (typically a shared-memory MIMD or an NUMA (non-uniform memory access) or an SMP box);
3 highly parallel machines (HPP or highly parallel processor) with medium-power processors (typically eight to 256) with shared memory (typically a global shared-memory or distributed-memory MIMD);
4 highly to massively parallel machines (MPP or massively parallel processor) based on medium-power processors (100 to a few thousand) (typically an MIMD with distributed memory);
5 massively parallel machines (MPP) with many thousands (4096 to 65,536) of low-power processors (typically a synchronous SIMD architecture but also becoming popular in the ASCI drive towards teraflop computing); and
6 the ASCI type of approach, where the MPP machines are themselves nodes on a network of many others.

In general, as a rule of thumb, more processors are harder to program than few. Parallel machines with only a small number of processors and global shared memory offer the least-effort route to parallel programming as serial code will run without change on a single processor. It is here where clever compilers can make the parallelisation task appear almost trivially easy and transparent; although this need not deliver the best performance possible, it does offer an easy route for suitable code. However, as the number of processors increases so too does the potential performance benefit and the greater the complexity of the task. Typically, it is only when you have 64 (or more) processors that distributed-memory machines begin to outperform shared-memory machines. Shared-memory machines tend to use high-performance but highly customised chips, whereas distributed-memory machines are usually built from high-performance workstation processors which are slower. The current trend is to build shared-memory hardware from mass-produced chips, and this is likely to produce affordable multi-processor-based workstations but not HPC. Most of the hardware manufacturers currently offer symmetric multi-processors (SMP) based on this approach. At the same time as processor speeds increase it becomes feasible to create monster MPP machines based on many thousands of fast but cheap processors *if* the parallel programming tasks are sufficient coarsely grained (a definition is given later) or *if* more efficient hardware architectures can be invented.

4.2.4 *Three basic machine architectures*

A final typology is to think in terms of what types of parallel hardware are most commonly used in practice. Demmel (1996) reminds us that basically there are only two main approaches to parallelism: SIMD and MIMD. Remember that SIMD involves applying the same operation on multiple pieces of data in parallel, while with MIMD different instructions are applied to different data at the same time. This results in a threefold typology of parallel machine architectures:

1 SIMD array or vector processors;
2 MIMD–SM (shared memory) machines with a small number of very fast processors; and
3 MIMD–DM (distributed memory) machines with larger numbers of fast but mass-produced components.

Examples of SIMD parallelism include a confusingly broad range of different hardware types, ranging from single-processor vector supercomputers such as the Cray I or multi-processor Cray J90 to extinct machines such as Thinking Machines CM-2, various array processors (i.e. ICL DAP) and also any pipelined floating-point accelerator attached to a mainframe (an approach that was fairly common in the 1980s but is now totally defunct). MIMD parallelism can be found in every type of parallel machine where separate processors can be individually programmed. The complication is that many machines now exhibit both SIMD and MIMD parallelism at different levels; for example, the multiprocessor Cray J90 (making it an MIMD machine) has multiple SIMD CPUs, each capable of vector processing). Equally, an MIMD machine can be programmed to function as an SIMD! However, the future may be simpler, as the HPC hardware world seems to be heading towards two generic hardware configurations, maybe only one. It is already possible to imagine hybrid MIMD–DM machines being built from MIMD–SM, SIMD and SISD subcomponents. In the long run, SIMD hardware seems likely to disappear inside processor technology as part of the floating-point unit. MIMD–DM parallel hardware is by far the more flexible and general type, and maybe it is this style of hardware that will eventually take over. No one really knows and, as geographers, does it much matter anyway? The key observation must be that the future of HPC is parallel, and the most future-proof way of programming these machines is via message passing and/or some suitable HPC language (such as HPF), but more about these aspects in subsequent chapters.

4.3 The three principal types of HPC hardware

Despite all our attempts at simplification, it is nevertheless true that this is still a potentially highly confusing topic. One problem is that there are several different terms in popular currency which are widely regarded as being synonymous, e.g. SIMD, and vector and array processors. At the same time, these terms can have far more restrictive, distinct and sometimes different meanings to different people, vendors and authors of books on parallel computing. What follows is our attempt to demystify the tangle.

4.3.1 Vector processors, array processors and SIMD hardware

A vector processor is usually considered to be a pipelined machine that is designed to perform large amounts of floating-point arithmetic. Most floating-point arithmetic in code is based on very simple operations such as add, subtract, multiply and divide plus some basic functions such as SQRT on real-valued data. Many scientific and engineering computing applications involve performing many millions of these operations. The challenge for the early computer builders was how to speed up this floating-point arithmetic computation, and the notion of a supercomputer was born. It was recognised long ago that much scientific computing consisted of large amounts of calculation being repeated on different data. In essence, many programs consist of a series of do-loops within which the arithmetic is contained. This is a natural reflection of the use of matrix algebra. Clearly, if it was possible to exploit any opportunities for local parallelism then it would be worth while. It is noted that the opportunities for parallelism also exists at three different levels: in the loop, in the sequence of arithmetic operations that is repeated many times and in the bit manipulations that constitute an individual add or multiply operation at the finest level of detail. Over time, attempts have been made to exploit all three.

Pipelining it seems was originally developed as a cost-saving engineering compromise method of implementing the SIMD computational model. Almasi and Gottlieb (1989) write that pipelining 'is an engineering technique that trades off performance for lower cost by executing the data items for each instruction in *overlapped* fashion on shared hardware, rather than in fully parallel fashion on fully replicated hardware' (p. 301). It is less efficient than a 'proper' parallel approach but was more than good enough to form the basis for HPC for about two decades. Maximum speeding up is related to degree of instruction overlap and on most machines was less than a factor of 10, whereas on a parallel machine speeding up is broadly equivalent to the number of processors. In essence then, a Cray I vector supercomputer (one of the first to be so designated) with an eleven-stage pipeline might be considered broadly comparable to an eleven-processor parallel machine with shared memory for some types of floating-point operation. The speeding up in vector processing comes from pipelining, where the processor can work on several instructions in different stages of completion at the same time. In essence, the processors form an assembly line, each unit performing a task and then passing the result on to the next one. Some may regard vector processors as merely serial machines with very fast floating-point units; however, that is being somewhat churlish!

Consider for example the operation SQRT ($2X_i + 5$) applied to a vector of N values of X. In serial Fortran, this would involve

```
DO I = 1, N
X (I) = SQRT(2.0*X(I)1 5.0)
ENDDO
```

the DO loop is repeated N times, with a considerable amount of performance loss because the time taken to perform the different operations varies tremendously; for instance, fetching a value for X(I) from memory would probably be 1000 or more times slower than the time taken to multiply it by 2.0. On a sequential computer, this loop would be equivalent to N statements of the form

```
X(1) = SQRT(2.0*X(1)+5.0)
X(2) = SQRT(2.0*X(2)+5.0)
X(3) = SQRT(2.0*X(3)+5.0)
```

etc.

```
X(N) = SQRT(2.0*X(N)+5.0)
```

Note also that each statement has to be processed completely before work can start on any of the next one. The idea in pipelining is to overlap these operations in order to keep the arithmetic units as busy as possible.

As can be seen from Table 4.3, it takes N + 3 steps to calculate a vector of N values using a three-step pipeline. If N is very much larger than 3 this represents a considerable speeding-up over the sequential model, which requires N × 3 steps to complete the operation. However, this assumes that each step of the pipeline takes the same amount of time and that it does not require a branch or subroutine call, both of which would break the pipeline and force it to operate in a serial way.

Now each of the N loops could be sent off to a separate processor, but in the late 1970s it was much easier to use a pipeline approach, whereby different stages in the computation could function independently and arrangements made so that they could be processed in an overlapped manner; see Hockney and Jesshope (1981) or Kogge (1981) for fuller descriptions if you are at all interested in any of this aspect of palaeo-computing technology.

Table 4.3 Example of vector processing.

Time step		*Processor*	
	1	*2*	*3*
1	$2{*}X_1$	–	–
2	$2{*}X_2$	$2{*}X_1 + 5$	–
3	$2{*}X_3$	$2{*}X_2 + 5$	$\mathrm{sqrt}(2{*}X_1 + 5)$
4	$2{*}X_4$	$2{*}X_3 + 5$	$\mathrm{sqrt}(2{*}X_2 + 5)$
.			
.			
.			
N	$2{*}X_N$	$2{*}X_{N-1} + 5$	$\mathrm{sqrt}(2{*}X_{N-2} + 5)$
N+1	–	$2{*}X_N + 5$	$\mathrm{sqrt}(2{*}X_{N-1} + 5)$
N+2	–	–	$\mathrm{sqrt}(2{*}X_N + 5)$

The SIMD approach has been operationalised at two different levels. The vector supercomputers overlapped the floating-point computation within DO loops. The array processors started off by exploiting the parallelism involved at the bit level within individual floating-point operations, also within a DO loop. This is a much finer degree of parallelism, but it could be applied to far greater numbers of data items simultaneously than with a vector processor.

To summarise, then, it is apparent that SIMD hardware is a far less general-purpose parallel computing than MIMD, but it offers a number of advantages: it is easier to build, it is inherently synchronous, it offers performance advantages for suitable problems, there is one common instruction memory for the whole machine, and as all the processors do the same thing at the same time they are easier to debug and very easy to program. However, this type of HPC hardware is really only suitable for data parallel problems; see Chapter 6.

4.3.2 MIMD–SM hardware

The shared-memory MIMD hardware is a natural evolution from a uniprocessor. Indeed, during the 1970s many mainframe and minicomputer manufacturers started to produce versions with two and sometimes more CPUs. They may be regarded as the ancestors of today's shared-memory MIMD machines. As the component speeds increased so too did the attractions of parallel processing via this more coarsely grained route. There were also good commercial reasons. If a user wanted twice the performance and no uniprocessor hardware could deliver it, then why not double the number of processors! The idea of scaleable computing is not a new one; it is just that manufacturers are slowly getting better at it.

Multiprocessors with global shared memory, whether based on mainframe, minicomputer or workstation hardware, have some very useful aspects. In particular:

1 it is the easiest parallel hardware to program, with a close resemblance to sequential programming;
2 some serial programs may run unchanged;
3 there is a simple approach to data movement using shared memory as the data-transmission mechanism;
4 uniform memory access time; and
5 all the processors share the same memory, thereby reducing storage redundancy.

However, shared-memory machines have some problems apart from the obvious ones relating to software (e.g. how to spread the work of a program efficiently and safely over more than one processor), especially:

1 memory contention restricts the maximum number of processors that can be used;

2 the hardware has historically tended to be expensive because of the use of customised components, which are not mass produced;
3 coarse-grained parallelism problems are best suited because of the relatively small number of fast processors; and
4 performance may suffer if users run hard to parallelise serial code with many data dependencies through automatic parallelising compilers rather than restructure or rethink their algorithms.

Shared-memory machines can be programmed in various ways. You could run a separate serial program on each processor concurrently, which would always yield maximum levels of performance efficiency, but this is not really parallel programming. You could use message passing to allow the processors to communicate and share the work of a single program (as in distributed-memory systems) or, more usually, you can use multiple processors on a single program using compiler directives which specify which sections of the program can be performed in parallel (*viz.* automatic parallelising compiler options) or by directly specifying threads of execution (somewhat harder to do).

MIMD–SM hardware is likely to become the dominant workstation architecture of the near future, once the annual speeding-up of the microchip starts to diminish and the case for multiple processors in the same box becomes important and cheap. At best, it will probably only ever yield medium-power HPC unless the individual processors are themselves leading-edge vector supercomputers. The jury is still out, but the economic arguments may soon defeat this shared-memory approach to HPC unless radical new architectures or clever software are devised that allow the memory contention problems to be avoided. Currently, there is some evidence of both.

4.3.3 MIMD–DM hardware

The key HPC hardware type for the future is almost certainly some variant of MIMD–DM machine. Almasi and Gottlieb (1989: p. 355) have a very nice way of describing this form of parallel computing that is worth reproducing. They write:

> Suppose we change the rules of the game and let each processor march to its own drummer rather than the single drum of an SIMD architecture by expanding each processor's memory so that it can now hold a substantial number of instructions as well as data. Suppose, too, that the problem to be computed is broken into substantial programlets (processes or tasks) that can be distributed to the processors for execution. <u>The existence of these programlets suddenly creates some new concerns that we didn't have to worry about before.</u>

The last sentence is a nice understatement, and we have taken the liberty of underlining it to emphasise its importance. The so-called programlets

simultaneously cause problems and create opportunities to exploit parallelism better due to the immense increase in flexibility they provide.

Chalmers and Tidmus (1996: pp. 23–24) provide a more basic definition: 'Distributed memory MIMD systems consist of a collection of processors, each with its own memory and connected together by some form of network. Processors communicate by passing messages via the interconnection network'. This type of MIMD machine is generally thought of being as very useful because:

1 it is very flexible;
2 it can handle heterogeneous processing (mixtures of processors of different types and speeds);
3 there is no memory contention;
4 other forms of parallel programming exist as special cases;
5 it offers the basis for highly parallel computing using up to several thousand processors built from mass-produced components; and
6 most leading-edge HPCs are now of this broad family type.

MIMD–DM therefore seems to be the future of HPC. It is perhaps the ultimate integrative system framework into which MIMD–SM and SIMD hardware components can be plugged if necessary or thought to be useful. It is also possible to cluster MIMD–DM systems.

In HPC, the benefits tend to come with 'costs' attached to them. The principal problems here are:

1 the greater difficulty of programming MIMD–DM machines so that codes run efficiently;
2 the need to explicitly change serial codes;
3 the need to redesign algorithms;
4 memory access times are uneven;
5 the processors communicate by sending and receiving messages; and
6 the greater risks of both bug creation and bug survival because parallelism adds an extra layer of complexity.

So how exactly do the processors communicate? How are they interconnected? How do messages go from processor A to processor B? What are the data transfer speeds? What are the principal network geometries? Well, as a geographer or social scientist, you really do not need to know or worry about much or any of this, other than to note that messages do not go from processor A to processor B instantly, and they travel many times slower than the speed at which the hardware can do arithmetic. So minimising network communications traffic is a universally good idea! Basically, if you are to avoid Amdahl's law you need to design algorithms that scale well so that as problem sizes increase and/or the number of processors increases, the time each processor spends doing computation rather than communicating with other processors also increases. The problem here is that this feature has to be built into your algorithms, and in practice this is not always easy to achieve. Other than this bit of common sense you can fortunately escape from most, if not all, of the details of the associated computer science.

4.4 Levels of parallelism and identifying them

The only other bit of computer science techno-speak that is really relevant here is the concept of granularity. Granularity is the relative amount of computation that can be performed in parallel. Granularity or parallel graininess is therefore an important concept when considering different types of parallelism and where to find it. You do not have to be a rocket scientist to realise that parallel computers feed on parallel tasks. However, if you are unable to split your work (i.e. code) up into pieces that can be run concurrently then you will not gain any benefits from running on a parallel processor other than access to a very large memory space. What is less obvious is where you can find parallelism. Hockney and Jesshope (1988) provide a list of the different levels of parallelism that have been used since the earliest days of computing. This is summarised in Table 4.4. It would appear that over time the focus of active parallel programming has moved upwards from (4) to (2) and maybe even (1). Baker and Smith (1996: p. 76) write: 'The graininess of a parallel application is a measure of how small a unit we can partition problems into'. However, it is important to note that maximum graininess (i.e. smallest size) is not best and that optimal granularity is dependent on the machine being used. Over time, the optimum size of granularity has been increasing. Today, you need to think in terms of large chunks of parallel computing with a chunk size that can automatically be increased (by your algorithm) as processors become faster.

Fine-grained parallelism is the smallest imaginable grain (size) of parallel processing activity; for instance, this could involve storing a single value on each processor and have each processor perform various calculations on that value. It could also involve distributing the work involved in a single arithmetic operation. So a problem that can be broken down into many small parallel tasks can be described as 'fine-grained'. At the other extreme is *coarse-grained parallelism*. This involves breaking a problem into the largest possible size of parallel components. Sawyer (1998: p. 35) writes: 'Fine-grained problems can have a greater maximum degree of parallelism than coarse-grained problems, but it is perfectly possible for coarse-grained problems to be implemented efficiently'. Indeed, there is an argument that as the processors used in distributed-memory machines become faster so it becomes important to increase the granularity of the parallelism to gain maximum benefit from faster processors and reduce potential communications bottlenecks.

Table 4.4 Levels of parallelism.

1 **at the job level**
- between jobs
- between phases of a single job

2 **at the program level**
- between parts of a program
- within do-loops

3 **at the instruction level**
- between phases of instruction execution

4 **at the arithmetic and bit level**
- between elements of a vector operation
- within arithmetic logic circuits

Source: Hockney and Jesshope (1988: p. 54).

Table 4.5 Levels of granularity.

Granularity	*Where to find it*	*Programming style*
very coarse-grained	job	
coarse-grained	program level	multi-tasking
medium-grained	subroutine level outer DO loop	MIMD
fine-grained	inner DO loop expression level	SIMD
very fine-grained	operation level bit level intra-instruction level	SISD

Source: based on Kober (1988).

Kober (1988) offers a useful classification of parallelism at different levels within a program; see Table 4.5. It is very useful to think in terms of where to find parallelism and what hardware and/or programming paradigm is most appropriate for handling it, although this does tend to be rather theoretical. With current HPC, if you want maximum performance you often have no real choice. In general, coarse- and medium-grained parallelism now provides the best performance, especially if the number of processors is limited or if the processors are fast ones. The key in all parallel design decisions is to try to maximise processor utilisation while minimising the time the processors are idle or doing or waiting for communications with other processors. What this means in practice is that you need to think in terms of decomposing or breaking up the work of your program so that each parallel task takes several seconds (or ideally far more) to perform. If it takes a few milliseconds then you are unlikely to achieve good levels of performance. The faster the processors become in relation to communications network speed, the greater the amount of work they need to perform per unit of communication unless the speed and bandwidth of the interconnection network increases at the same rate. The difficulty here is that processor speed-ups continue to outperform network improvements. If you fail to achieve a good balance then either your job's execution time will not scale linearly with the number of processors being used or it will reach a limit and then start to become worse, and you can easily achieve very poor levels of performance to the extent that more processors actually make it even worse! Fine-grained parallelism is harder to handle as in most cases it requires more inter-processor communication. Twenty years ago, this did not much matter because the bottleneck was the computation rather than memory or network access times. Today the reverse is true. The difficulty is that not all problems can be expressed in a coarsely grained fashion. Most problems can involve both coarse- and medium-grained parallelism, and it is most useful to be able to cover both of these on the same system via the same language.

4.5 Programming models

In practice, the parallelism available in the HPC hardware is hidden from the user and is presented by what is commonly termed a programming model. Demmel (1996: p. 2) defines a programming model as 'the interface provided to the user by the programming language, compiler, libraries, run-time system, or anything else that the user directly programs. Not surprisingly, any programming model must provide a way for the user to express parallelism, communication and synchronisation in his or her algorithm'. Historically, programming models were totally tied to particular hardware architectures. This was really bad news for users! When the hardware became obsolete or the company that made it went bankrupt, the code had to be rewritten for another machine because it was probably not portable. None of this helped to popularise parallel programming!

Today's programming models are designed to be largely independent of the hardware, which is great because it allows user code to survive machine obsolescence and aids portability. This is achieved via a layered approach, in which the user's program is separated from the hardware via libraries and compilers that map the parallel-programming model on to whatever the machine offers, hiding most of the machine-specific details. The potential drawbacks are:

1 not all programming models work equally well on all machine architectures: the 'horses for courses' argument still holds good;
2 not all problems are suited to all programming models;
3 there is a potential loss of performance due to loss of access to the internal functions of a particular machine; and
4 what Demmel (1996: p. 3) calls '*caveat* programmer', by which he means that the technology used to create the layers (compilers, libraries and run-time systems) is often incomplete, buggy, inefficient or some combination of all three.

There are a number of different parallel programming models and paradigms. Three basic types of interest to geographers are (1) vector parallel programming; (2) multi-tasking, data parallel and shared DO loops; and (3) message passing. Originally, each had its own architecture, but increasingly hybrid architectures

are able to handle more that one of them. However, they all require different programming principles. What these are and how they work is the subject of the next three chapters.

4.6 Examples of each type of parallelism

Well that is enough about computer hardware for the present. We end this chapter with a brief review of the different types of parallelism offered by the principal types of computer in Flynn's classification discussed in Section 4.2.1. Imagine a problem involving marking 100 examination scripts, each with five questions. Now let us consider how different computer hardware would handle it. In this example, you have to imagine that each marker is equivalent to a processor and each script to a chunk of data.

4.6.1 SISD hardware

One marker works through each script in the pile in sequential order. Well, that is obviously serial and this is essentially how conventional computers (and examination markers) operate. The time taken depends on the number of scripts and the average time for each script. The bottleneck is the marker's speed.

4.6.2 SIMD vector processor

Now take five markers: a supervisor gets the first script and sends it to Marker 1. Marker 1 marks question 1 and passes the script to marker 2. Then marker 1 starts the next script, while marker 2 marks question 2 and passes it to marker 3 and so on. You would almost but not quite achieve a fivefold speed-up, depending on how many scripts were to be marked because of idle time at the start and end. This assumes that it takes the same time to mark each question, otherwise the pipeline would stall and the speed-up would diminish to less than the theoretical best.

4.6.3 SIMD data parallel machine

A supervisor sends each marker some scripts. He (or she) then says 'mark question 1', and each marker does that. Only when all markers have finished question 1 can the supervisor record the marks and then announce that the markers all start question 2, etc. All questions are marked one at a time, exactly synchronised with all other markers. If some questions take longer to mark then everyone has to wait for all the marking to be completed. Optimum performance would be reached when the number of markers equalled the number of scripts or the number of markers could be exactly divided into the number of scripts; otherwise, you would have idle processors and slightly poorer performance.

4.6.4 MISD

Each marker is sent a copy of question 1 on script 1. They all mark it according to different criteria and end up with a different mark for it. This would not be very useful, although seemingly some examinations are sometimes conducted in something akin to this approach! This may also be regarded as a model post-modernist approach to parallel computing; *viz.* total chaos.

4.6.5 MIMD shared memory

Each marker is allocated a unique share of the scripts that they are to mark. All scripts are stored in a single pile at the end of the room to which all markers have access. The time taken to complete the marking process is the time taken by the slowest marker to complete his share of scripts. Note that performance could be improved by what is termed 'load balancing', so that the speed of the marker determines how many scripts are sent to be marked.

4.6.6 MIMD distributed memory

There are two alternative work distribution strategies. Each marker is sent a parcel containing a pile of scripts to mark in their own time. The parcel is sent by courier (simulating a message on an interconnection network). Once received, the scripts are marked in parallel by each processor and sent back to the central store in another parcel (by courier) only when the task is finished. Depending on the speed of the courier, a more efficient solution is for a controller to distribute a new script to each marker as they finish one and it is received, so if some scripts take longer than others or some markers work faster than others, no one is idle for long. This greater flexibility in work distribution is an important feature of this type of hardware.

4.7 Conclusions

There is not the slightest doubt that writing software for parallel computers is harder than for sequential machines. You need to understand the types of parallelism that the hardware can handle, where to find it in your code and how to create it when writing new or changing old algorithms. It is not largely or just a computer science problem but also a design and software engineering challenge for you. In fact, you do not need to know much about the computer science, although Flynn's taxonomy serves as a useful introductory guide as it is helpful to know what different types of computer exist and broadly how they work.

However, the most important HPC parallel machines are now MIMD computers or some variant of them with distributed memory, and it is these machines that provide the fastest and biggest HPCs. Less powerful HPC is offered by shared-memory machines. Algorithms that work well on distributed-memory machines will also tend to work well on these machines, but not *vice*

versa. If you wish to play safe, then learn the programming techniques needed for distributed-memory MIMD. How to program these machines (and other HPCs) is the subject of most of the rest of the book. Fortunately, this task has recently been made far easier by standardisation of the principal parallel programming tools.

5 Programming vector supercomputers

For over twenty years, most HPC was based on vector supercomputers. This chapter describes in simple terms how to program this type of very simple-minded parallel hardware. It is the easiest of HPC to handle but offers the smallest performance advantages. This chapter also tells a good HPC story about how a program that used to vectorise and would have taken nine days to run can be made to run in less than one hour.

5.1 Introduction

Until fairly recently (*circa* early to mid-1990s), nearly all the world's supercomputing was based on vector supercomputing hardware. They are referred to as vector computers because they provide instructions for manipulating vectors or arrays of data. Instead of doing one arithmetic operation at a time they can do many. Indeed, this type of machine still represents a very important part of the HPC scene. Programming vector supercomputers is probably the easiest form of parallel programming paradigm so far invented but also one that offers the most limited speed-up possibilities. Nevertheless, it is one of the oldest forms of parallel programming and the most established, with over twenty-five years of practical experience.

One of the first computers to offer pipelining was the CDC 6600. In the foreword to the documentation for the CDC 6600 by J. Thornton, Seymour Cray wrote in 1970 that the CDC 6600 was 'one of the early machines attempting to explore parallelism in electrical structure without abandoning the serial structure of the computer programs. Yet to be explored are parallel machines with wholly new programming philosophies in which the serial execution of a single program is abandoned'. The 'awesomely fast' CDC 6600 of 1970 was rated at about 1 megaflop, which was at that time a really terrific speed. However, it also needs to be said that programming vector supercomputers with few processors is probably the least future-relevant, because clock speed limitations will sooner or later put performance strangleholds on this approach to HPC. However, all is not lost, as it is likely that the strategy of building multi-processor parallel machines will increasingly be based on CPUs that are themselves vector processors or have vector-processing components within them. Indeed, pipelining is appearing

increasingly in microprocessor hardware as a means of speeding up floating-point operations. So knowing how to write efficient vector-processing code is also of benefit on many other types of hardware as chips increasingly use vector-processing procedures. Finally, most vector supercomputers now offer modest numbers of multiple processors. Discovering the principles of programming these machines is a good place to start your career as a parallel programmer.

Like much of parallel programming, if you are going to be able to utilise vector hardware properly you need to understand in general terms how it works, since you are in the driving seat. Only you are really smart enough to know what your program is meant to be doing (as distinct from what it appears to be doing due to various historical algorithmic improvements now lost in the mists of time) so that you are the best-placed person to alter it, if necessary, to improve its performance further but without changing the results it produces. The last point is quite fundamental. Your vectorised program has to produce the same results as the unvectorised or scalar version else your attempts to vectorise it have broken it! Fortunately, with vector hardware the single most important cause of 'broken programs' is you with an algorithmic or logic error rather than due to the hardware or software provided by others. As we shall see later, this is a nice feature (since you know who is to blame if it does not work, i.e. yourself). This feature is not shared by other more advanced forms of parallel hardware, where the causes of failure can be much more varied and harder to identify (maybe you or the system or the compiler, or more than one). Vector programming is a comparatively straightforward form of parallel programming. You should be able to master the principles, if not instantly, within a small number of hours.

5.2 The secrets of vector processing

Vector processing is a finely grained form of parallelism. It exploits parallelism that exists within a program at the level of the DO loop. If there are multiple sets of nested DO loops then usually only the innermost ones are vectorised. DO loops are a very useful place to optimise performance because it is here where nearly all the arithmetic computation takes place within most programs. The basic idea is to express the important DO loops as vector statements whenever and wherever possible. It is only these loops that vector hardware attempts to speed up. The definition of a vector in this context is a one-dimensional array. It is a very simple-minded approach. If the array has more than one dimension then the innermost DO loop will reference a one-dimensional slice (or vector) of it. It is this that the vector processor optimises. For example, consider the following fragment of Fortran code. Let us assume that there are four one-dimensional arrays (or vectors) x, y, z, p of dimension n, then

```
DO I=1,N
X(I)=Y(I)+Z(I)*P(I)
ENDDO
```

Here the arithmetic statement

```
X(I)=Y(I)+Z(I)*P(I)
```

is repeated n times. A serial processor would do something like the following:

- set i = 1
- load register 1 with value of y(i)
- load register 2 with value of z(i)
- load register 3 with value of p(i)
- multiply contents of register 2 by contents of register 3 and put result in register 4, i.e. z(i)*p(i)
- add contents of register 4 to contents of register 1 and put result in x(i)
- Now increment i = i + 1
- Repeat the above n − 1 times

Note that a computer operates just like a calculator in that each register (or memory on a calculator) contains only a single number. This serial execution is not very clever, since the same instructions are being repeated over and over again. In practice, reality is seldom as bad as this example would suggest, because most compilers would optimise register assignments and memory access would be handled very efficiently via a cache and attempts made to overlap some of the processing and memory reading and writing. However, the same code run on a vector processor would do something quite different. Vector hardware has vector registers, hence the name. Each operation now involves not one value in a register but up to 64 (on Cray vector hardware). So in the previous example the vector computer would do the following:

- load vector register 1 with up to 64 elements of array y
- load vector register 2 with up to 64 elements of array z
- load vector register 3 with up to 64 elements of array p
- multiply contents of vector register 2 by vector register 3 and put the results in vector register 4 i.e. z(i)*p(i) for up to 64 values at a time
- add vector register 4 to vector register 1 and put up to 64 sets of results into array x

Usually, the code runs up to ten times faster on Cray vector hardware than corresponding scalar code as each vector operation produces one result per clock cycle, while chaining between the function units allows overlapping of the load, multiply and add operations. On Cray hardware, the arrays are split up into blocks of up to 64 elements; on other machines, this may be larger (1024 elements on Fujitsu vector hardware) or smaller (ten on IBM hardware). The speeding-up results from the provision of special vector registers and vector operations (such as add, multiply, sqrt, perhaps exp and log) that operate not on a single number at a time but on many. If you are confused, then try rereading Chapter 5.

A nice feature is that the Fortran code is the same for both scalar and vector processors. However, it is handled by the compiler and hardware very differently. The fact that the Fortran code in both cases is identical makes vector processing very convenient and easy to use as no massive code rewrite is needed, at least in the first instance, although there are some statements that you need to avoid if good performance is to be achieved.

So vector parallel programming is all about exploiting parallelism at the DO loop level. This needs to be emphasised, since vector hardware only does *vectorised* loops quickly. Any scalar arithmetic tends to be unremarkable by comparison; it is still fast, but much slower, while integer arithmetic is often best avoided altogether; i.e. recode integers as reals. Likewise, vector processors are designed for floating-point arithmetic, so text processing might not be a good idea. Various restrictions or rules also apply that define whether a DO loop will vectorise. Once upon a time, you had to fiddle around with your code to force or re-express it in a suitable vector form. Today, compilers are much better at doing the restructuring for you, more or less automatically, while retaining the original logic inherent in your code. However, be aware that these compiler modifications may not be as efficient as if you redesigned the offending non-vectorisable loops by changing the logic of the underlying algorithm to emphasise its vectorisable features. The compiler only translates the code you give it and produces revisions that are logically identical to the original. As a result, this may not always yield optimal performance, but the answer will be the same. With a little algorithmic rethink you may be able to do much better yourself. However, it depends on how much time you wish to invest in code optimization and whether the effort is likely to be worth while. So it is very useful to know what to avoid and what to concentrate on when writing code for vector processors (or indeed for most modern serial processors).

Vectorisation on most machines only applies to one of a set of nested DO loops. It usually only works at the innermost DO loop level, and the DO loops have to conform to certain conditions. Typically, a DO loop will vectorise if the following conditions are met:

1. all calculations within the loop for one value of the loop index i have no implicit or explicit dependency on any previous value of i; and
2. all calculations involving variables which depend on i can be performed simultaneously.

Another way of expressing these conditions is to ask the question, can the arithmetic operations within a DO loop be performed in random order and still provide the same result. If the answer is 'yes' then it will vectorise, and it will also be recognised as a parallel DO loop on a shared-memory machine. If the answer is 'no' then it is neither parallel nor vectorisable, and maybe you should consider changing it.

There are several other aspects that may inhibit or stop vectorisation, depending on the compiler and the hardware. These include:

1 backward recursion or dependencies on previous values of a vector nearly always cause problems, although the compiler and/or you may be able to fix them by changing the code; examples of this type of problem are as follows:

```
x(i) = x(i)/x(i − 5) + x(i)/x(i − 4) + x(i)/x(i − 3)

x(i) = x(i + 10)/y(i)

x(i) = 1.5*x(i − 1) + 0.7*x(i − 2) + 0.5*x(i − 3)
```

2 conditional statements (i.e. ifs) may cause problems, although less so than a decade ago due to improved compilers, but they may still slow down execution speed because extra work is involved;
3 branching out of loops tends to cause major difficulties despite its obvious utility;
4 complex subscript calculations can also cause problems and slow down memory access, which in turn reduces the amount of arithmetic being performed;
5 scatter and gather operations on sparse arrays are at best inefficient and at worst will ruin performance even if they vectorise; for example, try to avoid code such as

```
x(i) = y(ip(i))*45.6

x(ip(i)) = y(j)*45.6
```

6 loops with FUNCTION and SUBROUTINE calls within are not vectorised, although many compilers may resolve the problem for you by expanding the code in-line, although this may then change the definition of the innermost DO loops;
7 some mathematical library functions may inhibit vectorisation, so check the documentation; and
8 integer arithmetic often tends to be done slowly as there may be no vector integer registers, in which case you may have to recode integers as reals if they are used in arithmetic statements instead of just for addressing arrays or in DO loops.

In addition to these requirements, there will also be several other characteristics that may well enhance vector performance. Particularly important are:

1 sufficient arithmetic work in the loop to keep the vector-processing units busy;
2 a high ratio of computation to memory access is helpful if it can be arranged, for example by merging DO loops;

3 you need a large fraction of the computationally intensive part of the program to vectorise, or run times will be dominated by the serial parts in accordance with Amdahl's law;
4 long DO loops are better than short ones, and maybe you will need to restructure your code to ensure that the longest loops are the innermost ones; and
5 look for and use any locally optimised versions of standard libraries; e.g. the BLAS library of matrix operations is widely available and is often optimised by vendors for their hardware.

Vectorising code is also worth while because it has a hidden benefit! Many microprocessors now use pipelines to speed up their arithmetic operations. These pipelines 'stall' for most of the same reasons that causes difficulties to a vector supercomputer. Code that is altered so that it vectorizes on vector hardware will also often run faster on all types of machines including workstations and PCs. So the effort may well be very worth while; see Dowd (1993).

5.3 Vectorising your code

Landau and Fink (1993: p. 303) offer this advice: 'The easiest path is to change nothing yourself and let the compiler do its job. The next easiest is to remove your own personal utility routines and used vectorized library routines (written by people who are paid to do that kind of stuff)'. So look for the BLAS library optimised for your hardware. However, you do need to review what the compiler has done to your code in case you can identify further improvements, or maybe it has missed some vectorisable loops for whatever reason. Alternatively, maybe the levels of performance are now sufficient and you can stop. There is no point in spending weeks of effort that gain you another 5 per cent off a run time that is already acceptable. Only if a more dramatic speeding-up is needed should you seriously consider fiddling with your code. However, many GIS applications will probably not use many standard functions included in the BLAS library and equivalents, so the onus is more firmly put on the programmer to learn enough to perform most of the vectorising and tuning themselves. The first area to look for improvement is in your use of subscripted variables, vectors or arrays, bearing in mind that not every subscripted calculation can benefit from vector hardware and there are limits to what vectorisation can deliver in terms of speeding up.

Remember also that Amdahl's law applies to vector processing. Total program execution time (T) is decomposed as follows:

$$T = T_{\text{serial}} + T_{\text{vector}}$$

The larger T_{vector} becomes relative to T_{serial} the greater the potential for improvement in speed. However, the potential is limited; e.g.

for 2 times the scalar performance then T_{vector} needs to be 50 per cent of total T
for five times the scalar performance then T_{vector} needs to be 85 per cent of total T

But remember that it is not possible to escape completely from Amadhl's law by making the problem size larger, as the maximum speed-up is limited to five–ten times by the nature of the vector-processing hardware. So a speeding-up of 5–10 times is about the best you can expect from single-processor vector hardware.

Now a speeding-up of five times does not seem much, but historically it has been very significant. It is useful to note that most other components on a vector supercomputer are designed for speed, and in the early workstation era (before about 1994) these relative speed improvements were extremely useful and of great practical significance in many areas of computational science. For example, in the early 1980s a Cray I vector supercomputer would be offering a performance gain of up to two orders of magnitude compared with a fast mainframe. In the mid-1990s, your workstation is probably running at a faster speed than a Cray J90 single processor without recourse to any visible vectorisation. In short, the era of the HPC based on vector supercomputing with a single or a small number of expensive high-performance vector hardware CPUs has probably ended. Soon clock speeds limit will reach some maximum beyond which it will be either uneconomic or infeasible to proceed. It is, however, only fairly recently (mid-1990s) that this has occurred and the era of highly parallel MIMD supercomputing has matured sufficiently to become a viable practical alternative to vector processors. The problem, as ever, is that those with the largest investment in vector code (built up over many years) will still want more vector hardware to preserve their software investment and to avoid having to port and thus rewrite and often redesign many of their algorithms for a purely MIMD world. It is hoped that this changeover will be short-lived. Fortunately, it is far less of a problem in geographical applications of HPC, due to a lack of previous vectorised software.

5.4 Optimisation of performance

5.4.1 A thirst for speed

It is useful to remember the reasons behind the use of vector parallel programming, which are shared by the rest of HPC. It is presumed that:

1 you have a problem of such computational intensity that it will not run to completion on a PC or workstation in a reasonable length of time (or at all); this is a moving target as workstations become faster and reflects the extent of your patience;
2 that the problem is sufficiently important or significant to justify the expenditure and investment of considerable programming effort to resolve it; and
3 that the science is of sufficient quality (i.e. internationally competitive alpha-plus-rated science as judged by relevant peer reviewers) and urgent enough to be allocated time on an expensive and typically scarce national supercomputing resource, which will usually be over-allocated.

Having met these criteria, the next stage in your HPC adventure is obtaining the maximum possible speed. Some equate faster computers with better science. Others might argue that without faster computers you cannot solve some problems at all. You should realise that program performance optimisation is one way of simulating a faster computer without really having to buy one! HPC is a drug. Its junkies are usually fascinated by the challenge of gaining access to more supplies of HPC and then code tweaking to induce a state of intellectual euphoria and self-satisfaction. Well, maybe this is a slight exaggeration, but there is certainly some truth in it. We know because we have been there! The good news is that the effects are not lasting, and it certainly delivers some good science and innovative research as a by-product. It can also be intense fun as well as hard work!

5.4.2 Tales about going faster

So a prime purpose of vector processing and HPC is the need for more computing power or speed. You want the best possible performance. If you are a geographer or social scientist you probably also need to demonstrate to curious onlookers that (1) you know what you are doing and (2) you can obtain levels of performance at least as good as other sciences and ideally better than average. So *if* you have a problem of *sufficient* importance to *require* a vector supercomputer then clearly you will want it to perform at the highest possible level.

A common rule of thumb is that in a typical program 10 per cent of the code will consume 90 per cent of the computing or CPU time; or so you hope! Others talk of an 80–20 per cent rule. If the program is complex and long then maybe you will have to hope that the critical parts are small. It is this 5–20 per cent of code that is worth streamlining. Kernighan and Plauger (1974) suggest some additional maxims: (1) do not bother fiddling around with minor changes; instead, consider improving the program by finding a better algorithm; and (2) let the compiler do the simple optimisations of expressions, etc. for you so that all you need to do is to make it easy for these to be recognised.

Bentley (1986) reinforces this point by way of an example. He quotes Appel (1985), who succeeded in reducing the run time of a program from a year to a day by speeding up his serial code by a factor of 400; see Table 5.1. However, speeding up is not free, and this task took several months of Appel's time; but seemingly it was very worth while, otherwise the program could not have been run. However, as hardware becomes faster the benefits of a few orders of magnitude speed-up become less. So it is possible that Appel's original code would now run 200 times faster on a workstation even without his tuning efforts, because hardware has improved by about this amount. However, it would have been of no use saying to Appel in 1985, 'well why not wait ten years before running your program!' This is being too wise after the event, and also a code that

Table 5.1 Appel's speed-up of a program.

Nature of changes	*Speed-up factor*
new algorithm	12.0
algorithm tuning	2.0
data structure reorganisation	2.0
single precision arithmetic	2.0
recoding a routine in assembler	2.5
faster hardware	2.0
Total Speed-up	400

Source: Bentley (1986).

would have taken a year to run on a minicomputer in 1985 was clearly unusable unless the task being performed could justify a dedicated computer and the users were amazingly patient in waiting for the result.

The simplest problem of all to speed up is where a complete algorithm can be replaced in its entirety by a vastly superior alternative. The classic example is sorting large arrays of numbers. The best methods may take seconds, the worst weeks! There is a danger of being HPC-myopic. If, for example, a point-in-polygon method takes on average 0.01 seconds per point and you have 10 million points to assign to 200 polygons, do you:

1 think of applying for time on a faster computer so that the same method can be made to run five–ten times faster; or
2 apply for time on a parallel processor with 512 processors so you can gain a speed-up of 512 times; or
3 think of using a different algorithm and/or rethink how it is being used so that the problem can be run on a workstation in much less time; or
4 decide that you can afford to wait for the original method to run to completion.

Options (1) and (2) should only be seriously considered if (4) and possibly (3) turn out to be impracticable. In the case of the point-in-polygon problem there are some very fast routines that can be made to run even faster by pre-classifying the data by minimum area rectangles before performing a point-in-polygon retrieval. HPC is no substitute for a better algorithm or for thinking about the problem. Only if there is no obvious simpler alternative should you consider it.

Code tuning can also easily become an all-consuming obsession because of the intellectual challenge that is involved. You can measure progress and the goal is clearly if rather fuzzily identified (i.e. shorter run times), but there is no way of knowing when to stop, because the best performance that can be obtained is an unknown. Nevertheless, it is important. Turton and Openshaw (1996) report even better results for a geography code than Appel (1985). They started with a spatial optimisation model code that used an embedded spatial interaction model to optimise a retail profit function; see, for example, Birkin *et al.* (1995).

The quality of the result depended on calculating this profit function for as many alternative solutions as could be computed in a reasonable length of time. The original code was run on a workstation using an old algorithm that may not have produced the best result but was computationally feasible as it did not take too long to run. Birkin *et al.* used a parallel supercomputer (the CM-200) to speed it up, allowing use of a potentially improved optimiser. Turton and Openshaw (1996) ported the problem on to the Cray T3D and then sought to optimise its performance on a single processor before running it on many. The results are very relevant here, especially as they managed to achieve a total speed-up of 2.8 million times, of which a factor of only 256 came from using a Cray T3D parallel computer (with 256 processors) and a massive factor of 1096 by changing the way the embedded spatial interaction model was calculated. Clearly, the performance gained depends on the initial performance benchmark and the extent to which specific application knowledge can be applied to the problem. Nevertheless, this work required three person-months of effort because the improvements necessary to achieve a dramatic result are seldom self-evident. In this case, most of the model's computations turned out to be unnecessary as many of the intermediate calculations could be updated rather than recomputed each time. However, such an observation often requires a good chunk of luck or a lot of effort. The results though can be spectacular. They serve to demonstrate, yet again, that algorithm changes are by far the most effective way of speeding up code. Only then do you start to consider the additional benefits to be gained from HPC. In this example, vectorisation would not have worked on the optimised code, because the performance gain resulted from a dramatic reduction in the amount of floating-point arithmetic being performed. What was left was no longer vectorisable, yet the performance of the revised program was broadly what a teraflop vector processor (if one could be built) would have managed with the original code. We will return to this point later.

5.4.3 Ossifying the past

In essence, porting is seldom a pure translation task, although this is often how it is regarded. There is a danger in ossifying inefficient algorithms by porting them unchanged on to faster hardware. Porting should be an opportunity to rethink the critical chunks of code that matter most. Imagine how embarrassing it would be to ask for two weeks of time on the world's fastest supercomputer only to discover that it could have been run in 10 minutes on your workstation! Or, putting it another way, you may be able to emulate the performance of inefficient code on a not-yet-built multi-teraflop machine costing many tens of millions of pounds by writing efficient code for the workstation that you already own! However, this quest for speed and performance is best justified when better science or more science or even some science can be performed because of it. The speeding-up in the performance of the spatial optimisation problem allowed a different optimiser to be used and thus obtained better-quality results (i.e. better science). In those areas where faster means better science the case is more easily proved. In many potential geography applications, the challenge is different in that high performance is needed purely to make the application possible on hardware that is typically not quite up to the task being presented to it or via the use of algorithms that are highly inefficient. However, if you succeed too well in improving algorithmic performance then the case for using HPC either disappears or is changed! This is of course an ideal outcome, because it then allows you to make greater use of the code on many more machines and also to consider using some of the 'saved' computing time to improve the quality of the results by doing more computation, which was previously completely out of the question. For example, if you can now run a model ten thousand times in the same time that you could once only run it once, then use some of the ten thousand possible model calculations to compute confidence intervals (via a bootstrap, see Chapter 2) or to investigate issues such as error propagation via Monte Carlo methods (this would use up probably all the saving but yield a better quality of result). It is important not to stand still. Faster computing runs previous tasks more quickly, but it also creates new demands for even more computation to improve the quality of the results, to handle larger problems and to allow the use of better or more robust algorithms.

However, you should not be using supercomputers for problems that do not need it, and problems that need HPC now may not need it in five years time, when your workstation will run that much faster. That is great, because HPC allows you to research, develop and prototype applications now that soon will not be so dependent upon it. The problem is that supercomputing usage can easily become an exclusive club for macho science. You are special because you use a supercomputer, but in reality this may also be a sign that your code is grossly inefficient and your algorithms are totally pathetic! The national supercomputing centres tend not to dwell much or at all on these latter aspects, emphasising instead the need for more computational power to remedy in at least some cases undiscovered (or hushed-up) inefficiencies in the codes being used.

5.5 A case study in vector processing using the Mark 1 geographical analysis machine as an example

5.5.1 Some background to the GAM

The geographical analysis machine of Openshaw *et al.* (1987, 1988) might be regarded as a first-generation geographical data-mining tool. It is a spatial database explorer based on a very simple principle. It uses a brute force search method to make up for the ignorance of not knowing where to look or what to look for by looking everywhere for evidence of localised clustering in two-dimensional spatial data. This exhaustive map search strategy made it very computationally intensive; indeed, it is a very fragile technology from a computing feasibility perspective. The search is exhaustive, so if you increase the search load by increasing the number of locations being examined or use higher-resolution

spatial data with more data points then the computing times may become unbearable even though it is a naturally parallel process. The GAM is of interest here because it was one of the earliest geographical applications that ran on Cray vector processors.

The GAM also cause many problems, not all of a HPC nature. It was an attempt to compute a solution to a problem that many people did not even appreciate existed. In 1986, the notion of a geographer running a seemingly powerful mainframe computer for weeks at a time on the same problem was difficult for some to accept, particularly if the underlying justification was in terms of a seemingly discreditable inductive technology – an early case of spatial data mining, or what some critics at that time called data dredging! Additionally, the concept of an 'analysis machine' was a highly emotive issue that was difficult for some people to relate to; maybe it still is! An 'analysis machine' implies an automation of rare statistical analysis skills, and this may appear to be a threat to those who possess such skills. It is also important to avoid undue excitement when reporting the results of such analysis machines, especially when applied to disease data as the results could be spurious, for example created by errors in the data being processed. There is still a need for independent validation no matter what method is used and regardless of how much computation is performed. The results of this type of search are essentially descriptive, focusing on patterns rather than process. However, the original GAM results were more or less corroborated, although subsequently described by some critics as being extremely obvious! There is now little doubt that the GAM was a major advance and that there are many different heuristic searches that could be developed to handle this problem; see, for example, Besag and Newell (1991) and Openshaw *et al.* (1989). Yet the GAM can be credited with the discovery of the putative 'Gateshead cancer cluster', and this may well be one of the few real achievements of spatial epidemiology since Snow's work on cholera in the nineteenth century; see Openshaw (1990, 1991). So it may be claimed that the GAM did provide useful insights into the database being investigated. Subsequent versions of the GAM are more intelligent. The latest GAM/K version has cluster recognition rules built into it; see Openshaw and Craft (1991). Subsequently, the International Agency for Research on Cancer (IARC) conducted a blind test of various cluster-hunting methods, including a GAM using fifty synthetic data sets. The results showed that the latest GAM version (GAM/K, described in Openshaw and Craft, 1991) worked best of all; see Alexander and Boyle (1996).

The problem is that the original version of GAM/K ran on a Cray Y-MP. It required a large amount of computing resource, it was non-portable and the code was never made widely available. The GAM needed vector supercomputers such as the Cray to cope with the computational task. It would run even better on a parallel supercomputer because of the implicitly parallel search process, but these machines were not then either sufficiently powerful or stable enough to be considered as serious contenders. The GAM is therefore a good case study on which to practise parallel programming skills. It is also still a most useful technology for exploratory spatial analysis in a purely geographical space.

5.5.2 GAM/1 algorithm and Fortran code

The GAM searches for evidence of localised clustering in point data. Assume you have a set of points (or small areas) located on a map with *x*, *y* coordinates and that for each point you have two counts representing the incidence of a disease (or something else of interest) and a measure of the population at risk. The original Mark 1 GAM, *circa* 1985–87, was used to analyse childhood leukaemia. It was based on the following algorithm:

Step 1: Define a study region
Step 2: Define an arbitrarily small circle, radius *r*, taking into account the spatial resolution of the data
Step 3: Cover the study region with a two-dimensional grid with a mesh size set at 0.2*r* so that the circles overlap by a large amount in an attempt to incorporate a sensitivity analysis to handle edge effects and data uncertainties
Step 4: For each circle located on a grid intersection retrieve a population at risk count (i.e. children) and incidence data (i.e. cancers) from a spatial database
Step 5: Compute the probability of an excess cancer rate under a Poisson assumption
Step 6: Display 'interesting' circles on the map, *viz.* those with a small probability of being a chance event
Step 7: Repeat for all circle locations defined by the two-dimensional grid
Step 8: Repeat Steps 3–7 for a range of circle radii (e.g. 1, 2, 3, 4, 5 . . . 50 km) considered to be of potential interest.
Step 9: Map the significant circles

The GAM/1 algorithm is really quite simple, and when programmed simply in Fortran (or any other language) it consists of a series of nested DO loops; see the program listing in Appendix 5.1. The critical DO loops are:

Loop 1: Do RADIUS = RADMIN, RADMAX, RADINC
Loop 2: Do CY = MIN_Y, MAX_Y, Y_INC
Loop 3: Do CX = MIN_X, MAX_X, X_INC
Loop 4: Compute population and cancer count for all X,Y points inside circle at (X,Y) of size (RADIUS) and write it out if there is an indication of an unusual number of cases according to some significance test.

Typical DO loop ranges if data for all the UK are used based on 1991 census enumeration districts are as follows:

Loop 1: Do RADIUS = 1,50,1
Loop 2: Do CY = 1,1209,RADIUS*0.2
Loop 3: Do CX = 1,650,RADIUS*0.2

Loop 4: Compute population and cancer count for all X,Y points inside circle at (X,Y) of size (RADIUS) and write it out if there is an indication of an unusual number of cases according to some significance test.

It is quite clear then that Loop 4, will be executed a very large number of times. Also in Loop 4, the easiest way of determining whether a data point lies within a circle located at centroid CX,CY of size RADIUS involves the following computation to calculate the distance of each of the N points from the centre of the circle. So within Loop 4 the following code is executed a very large number of times:

```
OBSP = 0.0
OBSC = 0.0
DO I = 1,145,716
DIS = (X(I)-CX)**2 + (Y(I) - CY)**2
IF(DIS.GT.0.0) DIS = SQRT(DIS)
IF(DIS.LE.RADSQ)THEN
      OBSP = OBSP + P(I)
      OBSC = OBSC + C(I)
      ENDIF
ENDDO
```

In this example, X(I), Y(I) contain the *x,y* map coordinates used to represent the location of the Ith census enumeration district: P(I) is the population at risk for the Ith point, and C(I) is the cancer count for the Ith point. Note that there is an assumption here that the cancer data have been aggregated to census enumeration districts. In the UK, 145,716 census enumeration districts are represented as point data. If higher-resolution postcode data had been available then this would have increased to about 1.6 million, or 32 million points if the Ordnance Survey's address point data for individual addresses had been used.

5.5.3 Code tuning and performance optimisation

The program is vectorisable but only within Loop 4, where 145,716 distance calculations would be performed. The problem here is that the amount of arithmetic being performed is not particularly high compared with the amount of memory addressing going on. For example, the statement

```
DIS = (X(I) - CX)**2 + (Y(I) - CY)**2
```

has only two subtractions, one addition and two squares for five memory accesses. This is a typical feature of many geography and GIS codes, and essentially there is not much that can be done about changing it. The only feasible alternative is to change the algorithm.

Let us start with the code in Appendix 5.1. How do you go about tuning it? Well you begin by timing it, i.e. measuring the performance of the program on a realistic but reduced size or small problem, one small enough to allow you to experiment with various changes to the code while still being representative of the program. The process is described by the following sequence of steps.

Step 1: Select a representative but simplified version of your problem that does not take too long to run.
Step 2: Decide whether the code is worth improving or needs replacing. Does the run time need to be substantially reduced or just fine-tuned here and there?
Step 3: Time it as it is, examine profiling details, identify slow loops and think about how performance might be improved by changing the code.
Step 4: If it seems that only fine tuning is needed then start to fiddle with the composition of DO loops and experiment with various code changes in an attempt to find better results on given hardware. Dowd (1993) offers many useful optimisation suggestions.

You should stop as soon as the predicted full production run times can be regarded as acceptable, otherwise you may end up wasting vast amounts of time and effort for no useful level of benefit. Landau and Fink (1993: p. 277) also offer some useful general advice: 'A survival tool known as common sense dictates that you concentrate your effort vectorizing those loops that matter most'. The rule is to do as little as possible beyond that which is necessary. Otherwise, you end up with a program that no longer works because you fiddled around with it so much that you introduced one or more bugs. More seriously, a law of diminishing returns sets in whereby the last few percentage points of improvement require a mammoth amount of effort. There is little point in worrying about sections of code that absorb very little amounts of CPU time or are only done once (i.e. input or output). Instead, go for the biggies first! Finally, you only do this tuning if you have to in order to allow the program to run to completion on real problems and/or if the code is important and of interest to multiple users. Otherwise, it is a monumental waste of your effort and some one else's supercomputer time while you fiddle. Also remember that today's supercomputer is tomorrow's workstation, so you should try to ensure that the fine tuning of the code is not too hardware-specific and is portable.

One useful suggestion is to look for opportunities in time-critical sections whereby you can replace DO loops by locally optimised subroutines in the BLAS library (optimised for a particular machine) or any other generally available performance library. However, avoid altering your code solely so that BLAS can be used if it involves extra work, since typically you do not gain much from BLAS (probably less than a factor of two over your original code). Additionally, avoid using any routine that is unique to one vendor's hardware; it just causes problems later when you wish to move the software to a different machine or you want versions to run simultaneously on multiple machines and PCs.

It is also worth noting that a highly complex code often depresses performance on vector machines and make the compiler's job harder. Simple code is often best and is also quickest to program and debug. So take out the structured programming you once learned! Consider moving subroutine bodies in-line if they are called millions of times. Look carefully at highly complex nested if-then-else statements. Make the code simpler and ease the compiler's task of optimising it for you.

Step 5: If fine tuning is insufficient then consider how to improve the algorithm or what other alternatives may exist. You only really discover that fine tuning is insufficient when it fails to deliver the performance gains that you believe you need! To do this algorithmic change you need to switch your gaze from a local tactical DO loop level to a more strategic view of what is going on.

On this GAM/1 problem, most of the CPU time is in Loop 4, which does the spatial data retrieval. A typical run would do this data retrieval loop 50 × 6045 × 3250 times, each involving 145,716 distance calculations. This is a lot, even for an HPC! Openshaw *et al.* (1987) knew this and used a KDB tree to optimise spatial data retrieval times on a serial mainframe. Unfortunately, KDB trees are complex recursive data structures that do not vectorise at all! As a result, a completely different spatial data retrieval method had to be invented. The other difficulty at that time was the restricted memory size of the Cray X-MP, which prevented the data being stored as simple arrays. The problem was not just the storage of X, Y, P and C vectors (each of 145,716 elements) but it also needed to store 501 randomised set of C values, which were used in the GAM/1 to assess multiple testing effects. Accordingly, the Cray X-MP version of the GAM/1 used a complex hash look-up to define the data located in the region of the circle. The data was then packed to conserve memory using Cray subroutines PACK and UNPACK. All this produced a highly complex code that was very efficient, but as Cray Y-MP memories increased in size much of the compression work may no longer be necessary. It also made the program Cray-specific and totally non-portable. It was even harder, a decade later, to remember all the intricacies of such a complex algorithm.

It is sensible to start again and to write the simplest possible GAM/1 coding, which is what we do here. After all, to work well with a vector processor you only need to ensure that the code is in a highly vectorised form where most of the calculation occurs. The GAM/1 code is already highly vectorised, so not much effort is needed to port it to a vector supercomputer. In fact, by far the hardest part is understanding how to use a remote alien machine, but even here the universality of the Unix operating system now helps tremendously. The GAM/1 code in Appendix 5.1 was written on a Sun workstation and ported without change on to a Cray J90 vector supercomputer at the Edinburgh Parallel Computing Centre in the UK. It worked first time.

Table 5.2 Vector processor results for the initial GAM/1 code on a single Cray J90 processor.

	Vectorisation off	*Vectorisation on*
CPU time in seconds	2416.5	246.7
Mflops	9.4	93.9

A sample representative problem for the tuning work was arbitrarily defined as one which involved 6000 circle calculations for a 10 km radius circle. This was small enough to be run on a Sun workstation and seemed reasonably representative. This is important, because you need to be able to verify the results to ensure that code changes do not alter the answers being obtained by running on both a workstation and a supercomputer. The code in Appendix 5.1 produced the results shown in Table 5.2 for the Cray J90 using a single processor.

The code achieves 93.9 mflops (million floating-point calculations per second), which is just less than 50 per cent of the theoretical maximum of a single J90 processor. However, this is probably about as much as you will achieve with many geography codes because of the typically high ratio of memory activity to computation. For instance, it has already been pointed out that inside Loop 4, which is vectorised, there are six memory reads and six floating-point operations. The sqrt subroutine is not included in the mflop count (for some reason). Nevertheless, this is a level of performance that would be regarded as good. But is it good enough? Certainly, the biggest DO loop is the innermost one. There are no data dependencies, it vectorises very well, and the loop length is at the maximum for this application. All the hot spots are vectorised. So is this good or bad? There is probably nothing else in the code listed in Appendix 5.1 that is worth changing. The subroutine call to calculate the Poisson probabilities is unimportant as it is called infrequently, while the input and output parts are one-offs.

It is certainly very tempting to stop here on the grounds that 93.9 mflops is good for a single Cray J90 processor in the mid-1990s, and this level of performance is not embarrassing. However, the problem is twofold:

1 a full GAM/1 run would take an estimated 8.9 days of CPU time on Cray J90 which might be difficult for geographers to obtain; and
2 there is concern that the program merely succeeded in vectorizing a very inefficient algorithm for spatial data retrieval by a supercomputer?

Following the advice offered above in Step 4, it is possible to set about tuning the code. Now the secret is not necessarily to vectorise everything in sight. Here there is nothing left to be vectorised in any of the hot spots. So all that is left to do is to try to minimise the amount of arithmetic work being carried out. This is helpful, because not all floating-point operations take the same amount of CPU time to perform. Typically, addition and subtraction are three times faster than multiplication or division (and division is often slower than multiplication), while mathematical library functions such as SQRT or EXP or LOG can be fifty or more times slower than multiplication and 150 times or more slower than addition or subtraction. So an obvious modification is to remove the SQRT from the code, which also gets rid of an IF statement. This is easily done by using RADIUS squared and distance squared instead of square-rooting it. The results are identical, and it should produce more efficient code. So Version 2 was produced, which has the following changes:

Loop 4: Compute population and cancer count for each circle

```
RADSQ = RADIUS*RADIUS
OBSP = 0.0
OBSC = 0.0
DO I = 1,145, 716
DIS = (X(I)-CX)**2 + (Y(I) - CY)**2
IF(DIS.LE.RADSQ)THEN
      OBSP = OBSP + P(I)
      OBSC = OBSC + C(I)
      ENDIF
ENDDO
```

The result is a factor of about 4 reduction in CPU time. This is quite a lot and very helpful, as it reduces the total run time to an estimated 2.3 days! A useful move, but it is still 2.3 days of CPU time on a supercomputer. What happens if you want to run the GAM on fifty different data sets? So what else can be done?

Well, using 32-bit rather than 64-bit precision on a workstation will typically speed up the run time by less than a factor of 2. It is useful to remember this trick, especially if 64-bit precision is not needed, as indeed is the case here. It is possible here because all the points affected by round-off error in the distance calculation are so far away from any circle that the lack of precision does not matter in the GAM. Single precision offers two kinds of speeding-up, assuming that it is done by the hardware:

1 faster computation; and
2 a doubling of memory bandwidth, thus improving cache efficiency because only 32 bits are being used instead of 64.

This single-precision GAM is called Version 3. Unfortunately, this does not help on a J90, since on this machine single precision is 64 bits! It would, however, do better on a 64-bit word size workstation.

We can consider altering the code to remove the last IF statement and use BLAS routines, which will have been optimised for the J90 (by now!) Unfortunately, this involves performing additional arithmetic. Nevertheless, it might just be worth investigating in case the last surviving IF statement is causing some slowdown in execution. Version 4 involves the following changes:

Loop 2:

```
DO CY = 1,1200,RADIUS*0.2
      DO I = 1,145716
      YY(I) = RADSQ - (Y(I) - CY)**2
      ENDDO
```

Loop 3:

```
DO CX = 1,650,RADIUS*0.2
```

Loop 4: Calculate distance of ED at X(I), Y(I) from grid point CX,CY but set WEI to 1 or 0 depending on whether point is inside circle. This avoids an IF and creates a 0,1 membership function

```
DO I=1,145716
    WEI(I)=DMIN1(DMAX1((YY(I)-(X(I)-CX)**2),0.0),1.0)
      ENDDO
C* form cancer count as DOT product of WEI and C
      OBSC=SDOT(N,WEI,1,C,1)
C* SKIP if count is too small
      IF(OBSC.LT.CANMIN)GOTO 200
C* form population count in the same way
      OBSP=SDOT(N,WEI,1,P,1)
```

Unfortunately, while there is an improvement compared with the original code of 2.5 times, it is poorer than the previous best version. The extra arithmetic improved the vector performance, but it took longer to run. There is an interesting lesson here of more general application. Vector performance increased, but so too did the CPU time! This is not uncommon but is not the desired result, although it may well look good to outside viewers. Herein is another more general vector computing problem. There is a degree of a 'How fast does your code run in Mflops' syndrome! It is important to stop thinking only in terms of Mflops and to consider the amount of science being done, and this is seldom directly related to Mflops; indeed, it can often be inversely related! It is all too easy to become what may be termed 'vector-blinded' or 'Mflop-obsessed' and to concentrate all one's tuning effort on getting maximum-sized vector DO loops to provide maximum Mflops almost without thinking whether there are better ways of doing it by other means which may actually reduce both Mflops and CPU times. Maybe CPU times are a better measure of the amount of science being done than Mflops.

The next step is to try to reduce the amount of arithmetic being performed by rethinking some more of the algorithm. Have another look at Loops 2 to 4. You will spot (sooner or later) that there is a bit of computational redundancy that the compiler failed to spot for us, but maybe it was too well hidden. In particular, half of the arithmetic effort in calculating distance is being wasted since that part which depends on CY is constant for the entire CX loop. Provided there is sufficient memory to hold another copy of Y then you can reduce the amount of arithmetic by about one-third to produce Version 5:

Loop 2:

```
DO CY = 1,1200,RADIUS*0.2
   DO I = 1,145716
   YY(I) = (Y(I) - CY)**2
   ENDDO
```

Loop 3:

```
DO CX = 1,650,RADIUS*0.2
```

Loop 4:

```
OBSP = 0.0
OBSC = 0.0
DO I = 1,145716
      DIS = (X(I) - CX)**2 + YY(I)
      IF(DIS.LE.RADSQ)THEN
      OBSP = OBSP + P(I)
      OBSC = OBSC + C(I)
      ENDIF
ENDDO
```

This is also helpful as it reduces the original CPU time by a factor of 4.7 to produce an estimated total run time of only 1.9 days. A useful improvement, but it is still not enough to make routine applications of the GAM convenient. So what else can be done?

The answer is to redesign the algorithm to further reduce the amount of arithmetic being performed. So let us re-examine what is going on in Loop 2 and see if some further arithmetic reduction may be possible. In Version 4, arithmetic was reduced in Loop 4 by storing the invariant results from this loop but the dimensions of Loop 4 remained unchanged. Clearly this is inefficient, since those points in Loop 2 that already lie outside the circle radius cannot possibly be worth considering in Loop 4. The result is Version 6:

Loop 2:

```
DO CY = 1,1200,RADIUS*0.2
      DO I = 1,145716
      YY(I) = (Y(I) - CY)**2
      ENDDO
```

Loop 3:

```
DO CX = 1,650,RADIUS*0.2
```

Loop 4:

```
DO I = 1,145716
IF(YY(I).LE.RADSQ)THEN
       DIS = (X(I)-CX)**2 + YY(I)
       IF(DIS.LE.RADSQ)THEN
       OBSP = OBSP + P(I)
       OBSC = OBSC + C(I)
       ENDIF
ENDIF
ENDDO
```

This reduces the amount of computation in Loop 4 and offers a speeding-up of 7.2 times and an estimated run time of only 1.2 days, but this code is probably still inefficient, because it soon becomes apparent that much of Loop 4 is really not needed at all. If only there was some way of removing most of it. Well there is, and after a few changes Version 7 is produced. This merely restricts attention to those points which stand some chance of being inside the circle. Version 6 did this but failed to reduce the DO loop size for the innermost loop. This new version is as follows:

Loop 2:

```
DO CY = 1,1200,RADIUS*0.2
      L = 0
      DO I = 1,145716
      DIS = (Y(I) - CY)**2
      IF(DIS.LE.RADSQ) THEN
       L = L + 1
       YY(L) = DIS
                  PP(L) = P(I)
      CC(L) = C(I)
      ENDIF
ENDDO
```

Loop 3:

```
DO CX = 1,650,RADIUS*0.2
```

Loop 4:

```
DO I = 1, L
IF((X(I)-CX)**2 + YY(I).LE.RADSQ)THEN
        OBSP = OBSP + PP(I)
        OBSC = OBSC + CC(I)
ENDIF
ENDDO
```

This change requires at worst 3 × 145,716 additional words of memory, but it dramatically reduces the size of the computation in Loop 4 because typically L will be a small fraction of 145,716. This is because the search region is very large in relation to the circle sizes of interest to the GAM, which will always tend to be relatively small in relation to the size of the study region. There is an additional very nice feature here in that when the circle radii are small there are many circles, but each retrieval is focused on a very small part of the data. When the circle radii are very large, there are few of them and it no longer matters that a large part of the data may now be involved in the spatial data-retrieval process. The result is a massive 122 times reduction in CPU times compared with the original code and a projected total run time of only 1.8 hours!

However, greed or enthusiasm to do better has now set in and this results in Version 8, which seeks to restrict further the range of Loop 4 from 1 to L to something much smaller, if we can define the minimum limits that need to be examined. This can be done as follows:

Loop 2: as before
Loop 3: as before
Loop 4: now restricts the range of the DO loop to between K1 and K2, which are maximum limits on the search. This involves putting all the data into an ascending X value sort order, calculating minimum and maximum X values (*viz.* CX – RADIUS, CX + RADIUS) and then doing a binary search on X to find start and end locations K1 and K2 in the data subset. This further reduces the amount of computation by 2.5 times on the test data. Version 8 becomes:

```
DO I = K1,K2
      IF((X(I)  - CX)**2 + YY(I).LE.RADSQ)THEN
      OBSP = OBSP + PP(I)
      OBSC = OBSC + CC(I)
    ENDIF
ENDDO
```

The estimated run time for the full problem is now only 1.3 hours.

Table 5.3 Summary of effects of tuning changes to GAM/1 code.

Algorithm	*Workstation CPU time (hours)*	*Cray J90 CPU time (hours)*	*Cray Mflops/s*
version 1	275	195.0	93.9
version 2	156	49.3	22.0
version 3	140	49.3	39.0
version 4	206	77.2	38.9
version 5	107	41.3	24.0
version 6	50	27.1	18.0
version 7	3	1.6	9.5
version 8	1	1.2	8.6

Note: workstation is a Sun Ultra Sparc 170. Times exclude data input.

Table 5.3 summarises the tuning experience for both the Cray J90 and a Sun Ultrasparc 170 workstation. Note how Mflops rates fall as CPU times decrease, which here reflects algorithmic improvements that dramatically reduced the amount of vector arithmetic being performed. Yet the programs are all highly vectorised, and none of the principal loops fail to vectorise. Note also how the workstation speeds are now comparable with that of a single J90 processor. All that vectorising effort needed to speed up the J90 is not present on the workstation, yet the CPU times are almost the same, and the workstation hardware is a small fraction of the cost of a J90.

Table 5.4 provides counts of the number of flops (floating-point operations) performed in the critical loops. How do you do that, you may wonder? Well, you add extra code to the program to count them! It is a crude but very effective form of code instrumentation. The effects of the algorithmic improvements are reflected in a massive reduction of the amount of arithmetic computation going on, and this is why the run times speed up to the extent they do.

Finally, you need to ensure that the results do not change as the time is reduced! With GAM they no longer do so, but it is quite common to find that code and algorithm improvements wreck your algorithm and produce different

Table 5.4 Counts of floating-point operations (flops).

Algorithm	*Count of flops (millions)*	*Workstation CPU time*
version 1	10498	275
version 2	5106	156
version 3	2957	140
version 4	4572	206
version 5	2957	107
version 6	114	50
version 7	114	3
version 8	46	1

results. Sometimes the error may have been in the original code, so do not automatically rule this possibility out. More likely, it will have been introduced by your code modifications. Less frequently, both programs may produce identical wrong results, but that is a different and far more difficult matter to detect. It is always useful to have some benchmark standard results to compare all subsequent modifications with. This requires a problem size small enough to run on a workstation. It always useful to remove any common mode errors due to the use of the same software, so different hardware is safest when creating standard results. This is what was done here.

A final problem is to avoid being misled by the test problem used in the tuning runs and thus producing production time estimates that are incorrect. This can be easily demonstrated. A full run of the GAM involves generating 23,690,246 circles and then evaluating them. According to the test problem, the final run time should be around 4680 seconds (1.3 hours), but in fact it was considerably less than this at 714 seconds (0.19 hours) due to (1) the speeding-up is greatest for the smallest circles, whereas the benchmark runs used quite large circles; and (2) the resolution of the clock induced rounding errors in the time estimates. This is a nice surprise! The very poor Mflop rate of 8.6 is really irrelevant because of the much improved performance of the revised code. If each GAM run can be regarded as delivering one scientific result, then the final code delivers about 300 times more science despite an average Mflop rate of 8.6. Finally, it should be noted that not all efforts at improving performance will succeed and that many are very time-consuming or may have little positive impact. However, it can be dramatic, although not all codes are worth improving and not all code changes will be as dramatic as those reported here.

Finally, you could ask why not use a more sophisticated spatial data point data-retrieval algorithm. For instance, would a quadtree or KB tree not speed things up further. The answer is almost certainly 'no' for two reasons: (1) the optimised retrieval takes into account the GAM spatial search being used, so it uses the fact that each circle is not independent of the previous ones nearby; and (2) neither quadtrees nor KB trees vectorise, whereas the GAM still does.

5.5.4 Loop 0!

Oh yes, we dared not mention this earlier, as it was rather theoretical when a single run of the GAM looked like taking nine days! Since it now takes only a few hundred seconds to run the GAM once, on the largest problem sizes that are probably applicable, you can now start to tackle some of the science problems that impact on the quality of the results. In short, there is an outer Loop 0, which you could now run! One way of handling the multiple testing problems implicit in the GAM is by Monte Carlo simulation. Quite simply, you could generate 501 random cancer distributions and run a separate GAM on each of them. This yields 501 sets of results, which can be used in a Monte Carlo significance test procedure to determine how difficult it is to obtain results similar to those observed with the real data in 501 (or 1000 or 10,000) random data sets.

You could not even dare to think about doing this when each run takes nine days, since 500 runs would require 12.2 years of Cray J90 CPU time. Now all this is within reach, and GAM becomes once more a real supercomputing problem. It is probably not possible (or worth while) to further improve the performance of the GAM. The single-processor CPU times will probably not be much bettered. However, it is still possible to gain factors of 10 to 512 reduction in wall clock time by using other parallel-processing techniques. The total CPU time being expended is more or less constant; the trick is to reduce dramatically how long in terms of wall clock or elapsed time you have to wait to get it. On a single processor with a single user the CPU time is the wall clock or elapsed time. On a multi-processor, the elapsed time will be some fraction of the total CPU time; crudely put, divide it by the number of processors. More about this in subsequent chapters, when the GAM is converted into a parallel format. However, in developing parallel GAMs the lessons learned here should be retained if at all possible.

5.5.5 *Summary of the basic code optimisation steps*

The following methods were employed here in optimising the GAM. They are:

1. ensure all loops vectorise;
2. ensure that the innermost loops have the largest counts and if necessary also unroll very short loops so that they become part of a larger loop;
3. try removing IF statements;
4. try removing expensive-to-compute functions (e.g. SQRT);
5. try using performance libraries to replace DO Loops;
6. try reducing the amount of arithmetic being performed by reorganising the algorithm;
7. localise scattered memory accessing by copying the data into consecutive locations if it is used repeatedly;
8. be prepared to use extra memory to reduce arithmetic work;
9. above all try to minimise the amount of arithmetic by redesigning the operations of the algorithm based on knowledge about the application but in a generic way so that it works on any data set; and
10. remember that memory accessing is much slower than arithmetic, so try to optimise caching by keeping memory access sequential; indeed, treat it like a slow disk file.

Not all these attempts were successful. It is not always obvious how to do (9) at the outset; often it emerges as desperation to improve the results sets in. So the best strategy is to start with the obvious and work progressively towards a better solution, interspersed with frequent timing runs. The aim is to gain large integer factors of improvement, not small percentages. However, once performance is good enough then stop. Also be careful to ensure that the improvements are generally applicable and not unique to either a specific test problem or to one application (if the code has more general applicability) or to particular hardware or to a particular operating system.

5.6 Case study 2: origin-constrained spatial interaction model

An exercise for you to do. Take the code for a very basic origin-constrained spatial interaction model and vectorise it. This model can be expressed as

$$P_{ij} = O_i\, A_i\, D_j \exp(-\beta C_{ij})$$

where

$$A_i = 1/\sum_j^N D_j \exp(-\beta C_{ij})$$

The notation can be explained as follows:

P_{ij}	is the predicted flow from an origin zone i to destination j
O_i	is the size of origin zone i
A_i	is a balancing factor designed to ensure that the total of the flows leaving origin zone i match O_i
D_j	is the distance or cost of travelling from origin i to destination j
β	is a parameter to be estimated
N	is the number of origins and destination zones.

This model in its modern form was derived by Wilson (1970, 1974). It is still widely used in one form or another in a large number of retail decision support systems; see Birkin *et al.* (1996) for examples. A simple introduction to the concepts and practice of spatial interaction modelling is contained in Openshaw *et al.* (1999). Here attention is focused on a very elementary form of spatial interaction model on the grounds that if you can cope with this model then other more complex ones will not cause you many additional problems.

You do not have to be a parallel-programming genius to realise that this model consists mainly of vector operations; for example, you could re-express this model as involving:

$$F_{ij} = \exp(-\beta C_{ij}) \quad \text{for all i and j} \tag{5.1}$$

$$A_i = 1/\sum_j D_j F_{ij}) \quad \text{for all i} \tag{5.2}$$

$$T_{ij} = O_i A_i D_j F_{ij} \quad \text{for all i and j} \tag{5.3}$$

So it should run well on a vector processor.

In the program given in Appendix 5.3, the data are created using a random number generator. This avoids the need to read any data with all the associated copyright, etc. issues, which are best avoided because they are not relevant here. Additionally, the computational performance of this model is not dependent on the data! Loop 1 deals with this task. Function RANF() is a Cray random number generator, but there is nothing special about it.

Loop 2 computes the origin and destination totals O_i and D_j

$$O_i = \sum_j T_{ij}$$

$$D_j = \sum_i T_{ij}$$

where T_{ij} is the observed (albeit random) flows created by the random number generator.

Loops 3 to 5 do most of the model calculation. Loop 3 is an outer loop, within which Loop 4 computes the balancing factor A_i

$$A_i = \sum_j D_j \exp(-\beta C_{ij})$$

while Loop 5 produces the model's predicted flows P_{ij}

$$P_{ij} = A_i O_i D_j \exp(-\beta C_{ij})$$

Clearly, all the DO loops vectorise and the code is also well designed in that Loop 3 is based on the best variable. If J had been used here instead, the program would be quite different and considerable additional computation would be required. You could try it out, if you wished. In general, then, the model is fully vectorised in its native form. Note that the PARAMETER statement sets a value for N, and this can be changed.

So how do you optimise this program's performance? There are no IF statements that can be removed; nor can the EXP function be dispensed with, even if it is expensive to compute. However, the latter can be optimised. As it stands, the program computes the code fragment

```
D(J) *EXP (-BETA*C(I,J))
```

$2N^2$ times. So why not store it, thereby saving N^2 additions, N^2 subtractions, $2N^2$ multiplications and more importantly N^2 EXP calls.

Declare a new array (F(N,N)) and use it to store the values

```
F(I,J)  =  D(J)*EXP  (-BETA*C(I,J))
```

and then refer to F(I,J) instead of recalculating it. This produces quite an improvement! See Appendix 5.4 for the code.

Another stage in optimisation would be to consider converting, by unrolling, the two-dimensional arrays into a longer one-dimensional one so that the N by N arrays become one-dimensional with N^2 elements. Unrolling the model is not too difficult, but it needs to be done consistently. The unrolling involves the following process for a model where N = 3 (i.e. three origins and three destination zones):

$T_1 = T_{11}$
$T_2 = T_{21}$
$T_3 = T_{31}$
$T_4 = T_{12}$
$T_5 = T_{22}$
$T_6 = T_{32}$
$T_7 = T_{13}$
$T_8 = T_{23}$
$T_9 = T_{33}$
C_{ij} looks similar.

Likewise for O_i, but this is a little more tricky because it is one-dimensional.

$O_1 = O_1$
$O_2 = O_2$
$O_3 = O_3$
$O_4 = O_1$
$O_5 = O_2$
$O_6 = O_3$
$O_7 = O_1$
$O_8 = O_2$
$O_9 = O_3$

and likewise for D_j

$D_1 = D_1$
$D_2 = D_1$
$D_3 = D_1$
$D_4 = D_2$
$D_5 = D_2$
$D_6 = D_2$
$D_7 = D_3$
$D_8 = D_3$
$D_9 = D_3$

Note that whether you unroll the rows before the columns (or *vice versa*) is arbitrary and does not matter much provided that it is done in a consistent manner.

The model looks similar to the original, except that the j subscript has been removed:

$$T_i = O_i D_i A_j \exp(-\beta C_i)$$

Note that i now runs from 1 to N^2. A slight complication is the need to compute the A_i term and then unroll it in the same manner as O_i.

All this unrolling dramatically increases vector lengths and simplifies address calculation. However, the effects are not that dramatic on the Cray J90, probably because of the vector length limit of 64. This may not be the case on different hardware. Appendix 5.5 contains the Fortran 77 code for this model.

5.7 Conclusions

Vector parallel programming is essentially easy, simple-minded and unlikely to yield erroneous results. However, it is often hard to get more than 50 per cent of theoretical performance of the vector machine, and even then this may only reflect a highly inefficient code. Additionally, not all algorithms can easily be re-expressed as vectorisable code without either doing great damage to them or requiring more rather than less computation. Other codes and algorithms may be too complex for vectorisation; for example, if they involve recursion. Yet others may be highly parallel but involve only short DO loops or multiple dependencies or many library calls or employ random memory accessing. Also, sometimes, parallelism at the DO loop level is too finely grained to fully exploit the parallelism present in the algorithm. Maybe these codes will benefit most from other forms of parallel processing.

There is also a danger in becoming Mflops-obsessed without realising that the relationship between Mflops, computing times and quantity of science need not be linear. The argument is made that vector processors still have a role to play, but by themselves they are no longer the future of HPC as supercomputing will become increasingly dominated by MIMD hardware (see subsequent chapters). Nevertheless, many of the code changes needed to improve the performance of a vector processor are unlikely to have been wasted, and the algorithmic redesign principles discussed and demonstrated in this chapter are also more generally applicable in parallel programming, as we later demonstrate.

Appendix 5.1: listing of GAM /1 Version 1

```
C* ~stan/gam/gam_1.f based on ~stan/mapex/program1.f
       PARAMETER (NCASE5150 000)
       IMPLICIT NONE
       DOUBLE PRECISION OVERAT,XMINE,XMAXN,XMINN,XMAXE,
     X OBSP,OBSC,RADIUS,DIS,CX,CY,PROB,
     X X(NCASE),Y(NCASE),P(NCASE),C(NCASE),
     X RADINC,RADMAX,RADMIN,POPMIN,CANMIN,THRESH
```

```
       CHARACTER*100 XYDATF,PCDATF,OUTFIL
       INTEGER I,LOOP,ICOL,IROW
       INTEGER TOTCAL,TOTDAT,TOTNCS,TOTNHY,STEP,ID,N,NCASE,
     X      MINN,MAXN,MINE,MAXE,NTIMES,NCALC,NDAT,NCALS,NHY,
     X      TOTCAL,TOTDAT,TOTNCS,TOTNHY,N2
C*
       WRITE(6,78001)
78001 FORMAT('*Geographical Analysis Machine GAM/1 (Feb 1997)'//)
C* Step 1. read data=====================================
C* set constants
C* read ini.file
       OPEN(UNIT=1,FILE='gamfiles.dat',FORM='FORMATTED',
     X STATUS='OLD')
C* read USER DATA file names
C.. this file contains X,Y data
       READ(1,10001) XYDATF
C.. this file contains Pop at Risk and Count of Real Cases
       READ(1,10001) PCDATF
10001 FORMAT(A)
C* get output results file name
       READ(1,10001) OUTFIL
       CLOSE(UNIT=1,STATUS='KEEP')

C* read X-Y data
       WRITE(6,6707) XYDATF
6707 FORMAT('*User Input X,Y File is;',A)
       OPEN(UNIT=1,FILE=XYDATF,
     X STATUS='OLD',
     X FORM='FORMATTED')
       DO I=1,NCASE
       X(I)=0.0
       Y(I)=0.0
       ENDDO
       N=0

       DO I=1,NCASE
       READ(1,*,END=999) ID,X(ID),Y(ID)
       N=I
       ENDDO
999    WRITE(6,123) N
123    FORMAT(5X,'*EOF at Case Number',I10)
       CLOSE(UNIT=1,STATUS='KEEP')
C* No data read?
       IF(N.EQ.0) STOP 1
C* read population and observed cancer data
       WRITE(6,6708) PCDATF
6708 FORMAT('*User Input Data File is;',A)
       OPEN(UNIT=1,FILE=PCDATF,
     X STATUS='OLD',
     X FORM='FORMATTED')
      DO I=1,NCASE
        P(I)=0.0
        C(I)=0.0
      ENDDO
      N2=0
      DO I=1,NCASE
      READ(1,*,END=199) ID,C(ID),P(ID)
      N2=I
      ENDDO
199    WRITE(6,123) N2
       CLOSE(UNIT=1,STATUS='KEEP')
C* No data read?
       IF(N2.EQ.0) STOP 2
C* files do not match
       IF(N.NE.N2) STOP 3
C* go through data and produce counts
      OBSP=0.0
      OBSC=0.0
      DO I=1,N
      OBSC=OBSC+ABS(C(I))
      OBSP=OBSP+P(I)
      ENDDO
      WRITE(6,8) N,OBSP,OBSC
8      FORMAT(
    X        ' *Number of input data records: ',I8/
    X        ' *Total population at risk: ',F10.0/
    X        ' *Total Cases  ',F10.0)
       IF(OBSP. EQ.0.0.OR.OBSC.EQ.0.0)STOP 3
       OVERAT=OBSC/OBSP
C* find Min and Max X,Y values to define search region
      XMINE=999999999.0
      XMINN=999999999.0
      XMAXE=0.0
      XMAXN=0.0
C* data are in 1 km units
      DO I=1,N
      XMINE=DMIN1(XMINE,X(I))
      XMINN=DMIN1(XMINN,Y(I))
      XMAXE=DMAX1(XMAXE,X(I))
      XMAXN=DMAX1(XMAXN,Y(I))
      ENDDO
      WRITE(6,7123) OVERAT,XMINE,XMAXE,XMINN,XMAXN
7123   FORMAT(
    X  ' *Global Incidence Rate per Population at Risk is ',F12.8/
    X'   *Minimum Easting is',F12.1,' Maximum is',F12.1/
    X'   *Minimum Northing is',F12.1,' Maximum is',F12.1)
      MINN=XMINN-1.0
      MINE=XMINE-1.0
      MAXN=XMAXN+1.0
      MAXE=XMAXE+1.0
C* Step 2. set search parameters=============================
C* circle radii are in KM
      RADMIN=10.0
      RADMAX=10.0
      RADINC=1.0
C* select probability threshold
      THRESH=0.005
C* set minimum circle size
```

```
      POPMIN=100.0
C* set minimum cancer count size
      CANMIN=2.0
C* write search parameters out
      WRITE(6,76541) RADMIN,RADMAX,RADINC,POPMIN
76541 FORMAT('*Minimum Circle radius is',F10.3,' 1 km'/
     X       '*Maximum Circle radius is',F10.3,' 1 km'/
     X       '*Circle increment set to',F10.3,' 1 km'/
     X       '*Minimum POPULATION size is',F10.0)
      WRITE(6,78234) THRESH
78234 FORMAT(' *Significance THRESHOLD set at',F12.6)
C* other global inits
      TOTCAL=0
      TOTDAT=0
      TOTNCS=0
      TOTNHY=0
C* convert all population counts into expected values
      DO I=1,N
      P(I)=P(I)*OVERAT
      ENDDO
C* reset minimum value
       POPMIN=POPMIN*OVERAT
C* set initial radius for circles
      RADIUS=RADMIN-RADINC
C* compute number of circle sizes to be exained
      NTIMES=(RADMAX-RADMIN)/RADINC+1.0
C* open output file
      OPEN(UNIT=9,FILE=OUTFIL,STATUS='UNKNOWN',
     X    FORM='FORMATTED')
C* Step 3. circle size loop=============================
C* *********circle size loop starts here **********************
       DO LOOP=1,NTIMES
C* set circle radius
       RADIUS=RADIUS+RADINC
CC        STEP=RADIUS*0.2+0.5001
        STEP=RADIUS
        IF(STEP.EQ.0) STEP=1
       NCALC=0
       NDAT=0
       NCALS=0
       NHY=0
C* Step 4. grid search: northing loop======================
     DO 100 IROW = MINN, MAXN, STEP
     CY=IROW
C* Step 5. grid search: easting loop=======================
       DO 200 ICOL = MINE, MAXE, STEP
       CX=ICOL
C* GET DATA WITHIN CIRCLE AT (IROW, ICOL)
      NCALC=NCALC+1
C* Step 6. get data for circle=========================
       OBSP=0.0
       OBSC=0.0
       DO I=1,N
C* calc distance of ED at X(I), Y(I) from grid point CX,CY
```

```
       DIS=(X(I)-CX)**2+(Y(I)-CY)**2
       IF(DIS.GT.0.0) DIS=DSQRT(DIS)
C* is point inside circle?
       IF(DIS.LE.RADIUS)THEN
C* yes so accumulate counts
         OBSP=OBSP+P(I)
         OBSC=OBSC+C(I)
       ENDIF
       ENDDO
C* Step 7. compute Poisson probability=====================
C* SKIP if population count is too small
        IF(OBSP. LT.POPMIN)GOTO 200
C* SKIP if too small to be of interest
        IF(OBSC.LT.CANMIN) GOTO 200
        NDAT=NDAT+1
C* CALCULATE SIGNIFICANCE LEVEL
        CALL POIS(OBSP,OBSC,PROB)
        NHY=NHY+1
        IF(PROB.GT.THRESH)GOTO 200
C* YES its significant so save
        NCALS=NCALS+1
C* Step 8. save circle info=============================
        WRITE(9,90001) CX,CY,RADIUS,OBSP,OBSC,PROB
90001 FORMAT(3F9.3,2F9.3,F10.7)
C* END OF EASTING
  200 CONTINUE
C* END OF NORTHING
  100 CONTINUE
C
C ************ end of search loop for given circle radius *************
C
      WRITE(6,78221)RADIUS,STEP,NCALC,NDAT,NHY,NCALS
78221 FORMAT(40(1H-)/' *RADIUS=',F12.2,'KM with STEP of',I6 ' KM'/
     X1H ,5X,'*Number of sites generated ',I10/
     X1H ,5X,'*Number of sites examined ',I10/
     X1H ,5X,'*Number of hypotheses tested ',I10/
     X1H ,5X,'*Number of significant circles',I10)
C* form global stats
       TOTCAL=TOTCAL+NCALC
       TOTDAT=TOTDAT+NDAT
       TOTNHY=TOTNHY+NHY
       TOTNCS=TOTNCS+NCALS
C* go back and do another circle size
         ENDDO
C*********************************************************************
C* END OF ALL RUNS***************************************************
C*********************************************************************
         WRITE(6,887) TOTCAL,TOTDAT,TOTNHY,TOTNCS
 887     FORMAT('0********** End of GAM Run ***************************'/
    X 1H ,'*Total sites generated is',I10/
    X 1H ,'*Total sites examined ',I10/
    X 1H ,'*Total hypotheses tested ',I10/
    X 1H ,'*Total significant circles ',I10)
     STOP
```

```
      END
C* CALCULATE POISSON PROBABILITY OF JA CANCERS
       SUBROUTINE POIS(OBSP,OBSC,PROB)
       IMPLICIT NONE
       DOUBLE PRECISION CUMPRB(3000),CONS(3000),
       X AMEAN,OBSP,OBSC,PROB
        INTEGER I,JA,J
        JA=OBSC
        AMEAN=OBSP
C* initialise
        DO I=2,JA
        CONS(I)=1D0/(I-1D0)
        ENDDO
C* calculate Poisson probability of JA cancers being observed
       IF(JA.GT.1) THEN
       CUMPRB(1) = EXP(-AMEAN)
       PROB = CUMPRB(1)
       DO J=2,JA
       CUMPRB(J) = AMEAN*CONS(J)*CUMPRB(J-1)
       PROB = PROB + CUMPRB(J)
       ENDDO
       PROB= 1.0 - PROB
       ELSE
C* 1 OR less cancers
        PROB = 1.0 - EXP(-AMEAN)
        ENDIF
        RETURN
        END
```

Appendix 5.2: listing of GAM/1 Version 8

```
C* ~stan/gam/gam_8.f DSQRT,N (**,-) and 1 IF removed, data filter
        PARAMETER (NCASE=150 000)
        IMPLICIT NONE
        REAL OVERAT,XMINE,XMAXN,XMINN,XMAXE,
     X  OBSP,OBSC,RADIUS,DIS,CX,CY,PROB,RADSQ,
     X  X(NCASE),Y(NCASE,P(NCASE),C(NCASE),
     X  RADINC,RADMAX,RADMIN,POPMIN,CANMIN,THRESH,
     X  YY(NCASE),XX(NCASE),PP(NCASE),CC(NCASE),
     X  LEFT,RIGHT
        CHARACTER*100 XYDATF,PCDATF,OUTFIL
        INTEGER I,LOOP,ICOL,IROW,L,K1,K2,INDEX (NCASE)
        INTEGER TOTCAL,TOTDAT,TOTNCS,TOTNHY,STEP,ID,N NCASE,
     X      MINN,MAXN,MINE,MAXE,NTIMES,NCALC,NDAT,NCALS,NHY,
     X      TOTCAL,TOTDAT,TOTNCS,TOTNHY,N2
C*
        WRITE (6,78001)
78001 FORMAT ('* Geographical Analysis Machine GAM/1 (Feb 1997) '//)
C* Step 1. read data ========================================

C* set constants

C* read ini. file
```

```
        OPEN (UNIT=1, FILE=' gamfiles.dat',FORM='FORMATTED'.
     X STATUS='OLD')
C* read USER DATA file names
C.. this file contains X,Y data
        READ (1,10001) XYDATF
C.. this file contains Pop at Risk and Count of Real Cases
        READ (1,10001) PCDATF
10001   FORMAT (A)
C* get output results file name
        READ (1,10001) OUTFIL
        CLOSE (UNIT=1,STATUS='KEEP')
C* read X-Y data
        WRITE (6,6707) XYDATF
6707    FORMAT ('*User Input X, Y File is; ',A)
        OPEN (UNIT=1, FILE=XYDATF,
     X  STATUS='OLD',
     X FORM='FORMATTED')

        N=0
        DO I=1, NCASE
        READ (1, *,END=999) ID,XX(I),YY(I)
        X (I)=XX(I)
        N=I
        ENDDO
999     WRITE (6,123) N
123     FORMAT (5X,'*EOF at Case Number',I10)
        CLOSE (UNIT=1,STATUS= 'KEEP')
C* No data read?
        IF (N.EQ.0) STOP 1
C* sort X values
        CALL SORT (X,INDEX,N)

C* re-order to reflect sort on X
        DO I=1,N
        ID=INDEX (I)
        X (I)=XX(ID)
        Y (I)=YY(ID)
        ENDDO
C* check sort
        DO I-2,N
        IF (X(I).LT.X(I-1)) STOP 55
        ENDDO

C* read population and observed cancer data
        WRITE (6,6708) PCDATF
6708    FORMAT ('*User Input Data File is; ',A)
        OPEN (UNIT=1,FILE=PCDATF,
     X  STATUS='OLD',
     X FORM='FORMATTED')
        N2=0
        DO I=1, NCASE
        READ (1, *,END=199) ID,XX(I),YY(I)
        N2=1
        ENDDO
```

```
199     WRITE (6,123) N2
        CLOSE (UNIT=1,STATUS='KEEP')
C* No data read?
        IF (N2.EQ.0) STOP 2
C* re-order to reflect sort on X
        DO I=1, n
        ID=INDEX (I)
        C (I)=XX(ID)
        P (I)=YY(ID)
        ENDDO

C* Files do not match
        IF (N.NE.N2) STOP 3

C* go through data and produce counts
        OBSP=0.0
        OBSC=0.0
        DO I=1,N
        OBSC=OBSC+ABS(C(I))
        OBSP=OBSP=P(I)
        ENDDO
        WRITE (6,8) n,OBSP,OBSC
8       FORMAT(
     X       '*Number of input data records:',18/
     X       '* Total population at risk:',F10.0/
     X       '*Total Cases',F10.0)
        IF (OBSP.EQ.0.0.OR.OBSC.EQ.0.0)STOP 3
        OVERAT=OBSC/OBSP

C* Find Min and Max X,Y values to define search region
        XMINE=999999999.0
        XMINN=999999999.0
        XMAXE=0.0
        XMAXN=0.0

C* data are in 1km units
        DO I=1, N
        XMINE=AMIN1 (XMINE,X(I))
        XMINN=AMIN1 (XMINN,Y(I))
        XMAXE=AMAX1 (XMAXE,X(I))
        XMAXN=AMAX1 (XMAXN,Y(I))
        ENDDO

        WRITE (6,7123) OVERAT,XMINE,XMAXE,XMINN,XMAXN
7123    FORMAT(
     X ' *Global Incidence Rate per Population at Risk is ',F15.9/
     X'  *Minimum Easting is',F12.1,' Maximum is',F12.1/
     X'  *Minimum Northing is',F12.1,' Maximum is', F12.1)
        MINN=XMINN-1.0
        MINE=XMINE-1.0
        MAXN=XMAXN+1.0
        MAXE=XMAXE+1.0

C* Step 2. set search parameters========================================
C* circle radii are in km
         RADMIN=10.0
         RADMAX=10.0
         RADINC=1.0

C* select probability threshold
         THRESH=0.005

C* set minimum circle size
         POPMIN=100.0
C* set minimum cancer count size
         CANMIN=2.0

C* write search parameters out
        WRITE (6,76541) RADMIN,RADMAX,RADINC,POPMIN
76541   FORMAT ('*Minimum Circle radius is',F10.3,' 1km'/
     X          '*Maximum Circle radius is',F10.3,' 1km'/
     X          '*Circle increment set to',F10.3,' 1km'/
     X          '*Minimum POPULATION size is',F10.0)
         WRITE(6,78234) THRESH
78234    FORMAT(' *Significance THRESHOLD set at',F12.6)

C* other global inits
         TOTCAL=0
         TOTDAT=0
         TOTNCS=0
         TOTNHY=0

C* convert all population counts into expected values
        DO I=1,N
        P(I)=P(I)*OVERAT
        ENDDO

C* reset minimum value
        POPMIN=POPMIN*OVERAT

C* SET INITIAL RADIUS for circles
      RADIUS=RADMIN-RADINC

C* compute number of circle sizes to be examined
       NTIMES= (RADMAX-RADMIN)/RADINC+1.0

C* open output file
        OPEN (UNIT=9,FILE=OUTFIL, STATUS='UNKOWN',
     X     FORM='FORMATTED')

C* Step 3. circle size loop========================================

C* *********circle size loop starts here**********************
       DO LOOP=1,NTIMES
C* set circle radius
       RADIUS=RADIUS+RADINC
       RADSQ=RADIUS*RADIUS
C*  STEP=RADIUS*0.2+0.5001
```

```
C*  STEP=RADIUS
       IF(STEP.EQ.0) STEP=1

       NCALC=0
       NDAT=0
       NCALS=0
       NHY=0

C* Step 4. grid search: northing loop======================================
      DO 100 IROW=MINN,MAXN,STEP
      CY=IROW
      L=0
      DO I=1,N
      DIS=(Y(I)-cy)**2
      IF(DIS.LE.RADSQ)THEN
        L=L+1
        YY(L)=DIS
        XX(L)=X(I)
        PP(L)=P(I)
        CC(L)=C(I)
        ENDIF
      ENDDO
      IF(L.EQ.0)GOTO 100

C* Step 5. grid search: easting loop=======================================
      DO 200 ICOL=MINE,MAXE,STEP
      CX=ICOL
C* GET DATA WITHIN CIRCLE AT (IROW,ICOL)
      NCALC=NCALC+1

C* establish search region
      LEFT=CX-RADIUS
      RIGHT=CX+RADIUS
      CALL FIND(XX,L,LEFT,K1,1)
      CALL FIND(XX,L,RIGHT,K2,2)
C* Step 6. get data for circle=========================================
      OBSP=0.0
      OBSC=0.0
      DO I=K1,K2
C* calc distance of ED at X(I),Y(I) from grid point CX,CY
      DIS=(XX(I)-cx**2+YY(I)
C* is point inside circle?
      IF(DIS.LE.RADSQ)THEN
C* yes so accumulate counts
        OBSP=OBSP+PP(I)
        OBSC=OBSC+CC(I)
      ENDIF
      ENDDO

C* Step 7. compute Poisson probability====================================

C* SKIP  if population count is too small
         IF(OBSP.LT.POPMIN)GOTO 200
C* SKIP  if too small to be of interest
```

```
        IF(OBSC.LT.CANMIN)GOTO 200
        NDAT=NDAT+1
C* CALCULATE SIGNIFICANCE LEVEL
        CALL POIS(OBSP,OBSC,PROB)
        NHY=NHY+1
        IF(PROB.GT.THRESH)GOTO 200
C* YES its significant so save
        NCALS=NCALS+1

c* Step 8. save circle info=========================================
        WRITE (9,90001) CX,CY,RADIUS,OBSP,OBSC,PROB
90001   FORMAT(F9.3,2F9.3,F10.7)
C* END OF EASTING
  200 CONTINUE
C* END OF NORTHING
  100 CONTINUE
C
C ************ end of search loop given circle radius ************
C
      WRITE(6,78221)RADIUS,STEP,NCALC,NDAT,NHY,NCALS
78221 FORMAT(40(1H-)/' *RADIUS=',F12.2,'KM with STEP of ',I6'KM'/
     X1H ,5X,'*Number of sites generated',I10/
     X1H ,5X,'*Number of sites examined',I10/
     X1H ,5X,'*Number of significant circles',I10/
     X1H ,5X,'*Number of significant circles',I10/
C* form global stats
        TOTCAL=TOTCAL+NCALC
        TOTDAT=TOTDAT+NDAT
        TOTNHY=TOTNHY+NHY
        TOTNCS=TOTNCS+NCALS
C*  go back and do another circle size
         ENDDO
C***************************************************************************
C* END OF ALL RUNS**********************************************************
C***************************************************************************
      WRITE (6,887) TOTCAL,TOTDAT,TOTNHY,TOTNCS
887   FORMAT('0********** End of GAM Run **************************'/
     X  1H ,'*Total sites generated is',I10/
     X  1H ,'*Total sites examined    ',I10/
     X  1H ,'*Total hypotheses tested ',I10/
     X  1H ,'*Total significant circles',I10)

      STOP
      END
C* CALCULATE POISSON PROBABILITY OF JA CANCERS
      SUBROUTINE POIS(OBSP,OBSC,PR)
      IMPLICIT NONE
      DOUBLE PRECISION CUMPRB(3000),CONS(3000),
     X AMEAN,PROB
      REAL OBSP,OBSC,PR
      INTEGER I,JA,J
      JA=OBSC
      AMEAN=OBSP
C* initialise
```

```
      DO I=1,JA
      CONS(I)=1DO/(I-1DO)
      ENDDO
C* calculate Poisson probablity of JA cancers being observed
      IF(JA.GT.1) THEN
      CUMPRB(1) = EXP(-AMEAN)
      PROB = CUMPRB(1)
      DO J=2,JA
      CUMPRB(J) = AMEAN*CONS(J)*CUMPRB(J-1)
      PROB = PROB + CUMPRB
      ENDDO
      PROB = 1.0-PROB
      ELSE
C*  1 OR less cancers
      PROB = 1.0-EXP(-AMEAN)
      ENDIF
      PR=PROB
      RETURN
      END
C* BINARY SEARCH
        SUBOUTINE FIND(X,N,VALUE,IPOS,FLAG)
        IMPLICIT NONE
        REAL X(N),VALUE,LAST
        INTEGER N,IPOS,MAX,MIN,K,FLAG


C* check range
        IF(VALUE.LT.X(1))THEN
              IPOS=1
              RETURN
              ENDIF
        IF(VALUE.GT.X(N))THEN
              IPOS=N
              RETURN
              ENDIF
C* BINARY SEARCH
           MAX=N
           MIN=1
           K=(MAX+MIN)/2

C* COMPARE
4000       CONTINUE
           IF(VALUE.EQ.X(K))THEN
                 IPOS=K
                 GOTO 100
            ENDIF

C* UPDATE SEARCH
            IF(X(K).GT.VALUE)THEN
                 MAX=K-1
            ELSE
                 MIN=K+1
            ENDIF

          IF(MAX.GE.MIN)THEN
```

```
            K=(MIN+MAX)/2
            GOTO 4000
         ENDIF

C*  NOT FOUND
         IPOS=MIN
C* find edge
100      CONTINUE
C* left edge========================
         IF(FLAG.EQ.1)THEN
C* check with target
         IF(IPOS,EQ.0) IPOS=1

25       IF(X(IPOS).GE.VALUE)THEN
            IF(IPOS.EQ.1)RETURN
            IPOS=IPOS-1
            GOTO 25
            ENDIF
C* X(IPOS) LT target
         LAST=X(IPOS)
24       IF(IPOS.EQ.1)RETURN
         IPOS=IPOS-1
         IF(X(IPOS).EQ.LAST)GOTO 24
         IPOS=MINO(IPOS+1,N)
         RETURN
         ENDIF
C* right edge========================
C* check with target
          IF(IPOS.EQ.0) IPOS=1
250       IF(X(IPOS).LE.VALUE)THEN
             IF(IPOS.EQ.N)RETURN
             IPOS=IPOS+1
             GOTO 250
          ENDIF
C* X(IPOS) GT target
         LAST=X(IPOS)
240      IF9IPOS.EQ.N)RETURN
         IPOS=IPOS+1
         IF(X(IPOS).EQ.LAST)GOTO 240
         IPOS=MAXO(IPOS-1,1)
         RETURN
         END
C* Sort routine
       SUBROUTINE SORT(A,INDEX,N)
       REAL A(N),B
       INTEGER INDEX(N)

       DO I=2,N
       INDEX(I)=I
       ENDDO

       IF(N.EQ.1)RETURN
       INC=N/2
100    CONTINUE
```

```
      LIMIT=M-INC
      DO 50 I=1,LIMIT
        J=I+INC
      IF(A(I).LE.A(J))GOTO 50
      B=A(I)
      A(I)=A(J)
      A(J)=B
      II=INDEX(I)
      INDEX(I)=INDEX(J)
      INDEX(J)=II
50    CONTINUE
      INC=(INC*3)/4
      IF(INC.GT.1) GOTO 100
      L=N-1
500   IF(L.LE.0) RETURN
      K=0
      DO 200 I=1,L
        J=I+1
      IF(A(I).LE.A(J)) GOTO 200
      B=A(I)
      A(I)=A(J)
      A(J)=B
      II=INDEX(I)
      INDEX(I)=INDEX(J)
      INDEX(J)=II
      K=I
200   CONTINUE
      IF(K.LE.0) RETURN
      L=K-1
      GOTO 500
       END
```

Appendix 5.3: listing of initial spatial interaction model

```
Program sim
!* si_1. f Spatial Interaction Model
      IMPLICIT NONE
      INTEGER, parameter: : N=1000
      REAL, parameter : : BETA=0.25
      REAL,T(N,N), C(N,N), O(N), D(N), P(N, N), SUMS (N), SS, SUMX, A (N)
      INTEGER I, J
      read *, T
      read *, C
      !* calculate Oi and Dj
      O=0.0
      D=0.0
      O=sum (T,1)
      D=sum (t,2)
      !* calculate model
      SS=0.0
      !*Calc A(I)
      SUMS=0.0
      do i=1,n
```

```
        A(i)0=1.0/sum(d*exp(-beta*c(i,:)))
        !* calc model
        ss=ss+sum(((A(1)*O(I)*D*EXP(-BETA*C(I,:)))-T(i,:))**2)
        enddo
        SUMX= N
        SS=SS/(SUMX*SUMX)
        PRINT '(F15.9)' , SS
        STOP
END program sim
```

Appendix 5.4: listing of spatial interaction model with calculation reduced by storing some results

```
C* si_1.f Spatial Interaction Model
        IMPLICIT NONE
        INTEGER N
        REAL BETA
        PARAMETER (N=1000, BETA=0.25)
        REAL T(N,N),C(N,N),O(N),D(N),P(N,N),SUM,SS,
   X RANf,A(N)
        INTEGER I,J
C* generate some random data (Loop 1)
        SS=RANf()
        DO I=1,N
        DO J=1,N
        T(I,J)=RANf()*10.0
        C(I,J)=RANf()*100.0
        ENDDO
        ENDDO
C* calculate Oi and Dj (Loop 2)
        DO I=1,N
        O(I)=0.0
        ENDDO
        DO J=1,N
        DO I=1,N
        O(I)=O(I)+T(I,J)
        D(J)=D(J)+T(I,J)
        ENDDO
        ENDDO
C* calculate model (Loop 3)
        SS=0.0
        DO I=1,N
C* Calc  A(I) (Loop 4)
          SUM=0.0
          DO J=1,N
                SUM=SUM+D(J)*EXP(-BETA*C(I,J))
                ENDDO
                A(I)=1.0/SUM
C* calc model (Loop 5)
        DO J=1,N
        P(I,J)=A(I)*O(I)*D(J)*EXP(-BETA*C(I,J))
        SS=SS+(P(I,J)-T(I,J))**2
        ENDDO
```

```
        ENDDO
        SUM=N
        SS=SS/(SUM*SUM)
        WRITE(6,123)SS
123     FORMAT(F15.9)
        STOP
        END
```

Appendix 5.5: listing of final version of spatial interaction model

```
C* si_3.f Spatial Interaction Model **unrolled
        IMPLICIT NONE
        INTEGER N
        REAL BETA
        PARAMETER (N=2000, BETA=0.25)
        REAL T(N*N),C(N*N),SUM,SS,
    X RANf,OO(N),DD(N),F(N*N)
        INTEGER I,J,K1,K2,L
C* generate some random data
        SS=RANf()
        DO I=1,N
        OO(I)=0.0
        DD(I)=0.0
        ENDDO
        L=0
        DO I=1,N
        DO J=1,N
        L=L+1
        T(L)=RANf()*10.0
        C(L)=RANf()*100.0
        OO(I)=OO(I)+T(L)
        DD(J)=DD(J)+T(L)
        ENDDO
        ENDDO
C* unroll Dj
        L=0
        DO I=1,N
        DO J=1,N
        L=L+1
        F(L)=DD(J)
        ENDDO
        ENDDO
C* calculate model
        SS=0.0
C* remove constant calculation
        DO J=1,N*N
        F(J)=F(J)*EXP(-BETA*C(J))
        ENDDO
C* calculate Ai terms
        K1=1
        K2=N
        DO I=1,N
C* Calc A(I)
```

```
      SUM=0.0
      DO J=K1,K2
      SUM=SUM+F(J)
      ENDDO
      SUM=OO(I)/SUM
      DO J=K1,K2
      SS=SS+(F(J)*SUM-T(J))**2
      ENDDO
      K1=K1+N
      K2=K2+N
      ENDDO

      SUM=N
      SS=SS/(SUM*SUM)
      WRITE(6,123)SS
123   FORMAT(F15.9)
      STOP
      END
```

6 Shared-loop and data parallel programming

This chapter considers three different ways of parallel programming: multi-tasking, sharing out DO loops over many processors and data parallel programming. The focus is on how to do it, identification of the problems and practical advice on how best to handle the difficult bits. This approach to parallel programming is fairly easy. Most readers will be able to master the skills in a day or two if they already know how to program and have access to appropriate hardware and software. There is an argument that fairly soon most workstations will be offering this type of parallel processing capability, which makes the task of discovering how to do it more than worth while. However, let us be realistic. This form of parallel programming is unlikely in the near future to be what leading-edge HPC systems can handle. If you need maximum HPC power you will need to read Chapters 7 and 8.

6.1 Introduction

A number of other approaches to parallel programming offer much greater flexibility than vectorisation. The simplest involves spreading either complete jobs or sets of DO loops over multiple processors so that they are run in parallel (at the same time as each other). This particular parallel-programming paradigm and parallelisation strategy comes in various forms. It may be called job farming, multi-tasking, macro-tasking, auto-tasking or micro-tasking. It may also be called data parallel programming, spreading DO loops or shared DO loops. The different names tend to reflect a mix of personal preference and different historical origins. Also, at one time each was associated with particular hardware and specific vendors, but it is all essentially the same broad type of parallel programming. It is an attempt to move on from the fine-grained parallelism that exists only *within* a DO loop that vector processing exploits to a medium and more coarsely grained form. In many ways, this requires more complex hardware and software and as a result it has had a long gestation period. Indeed, it is only really in the 1990s that this approach to parallel HPC has become a practical proposition. One of the authors remembers using an Encore Multimax with twenty processors in the mid-1980s. There were only two problems. First, the compiler was buggy and undeveloped, and second, there was no floating-point hardware, which made it very slow. However, it was without doubt a nice computer science plaything. Today, this sort of parallelism is widely available in workstations with multiple CPUs (for example, the Sun Ultra 60 has two processors) as well as in much more expensive parallel hardware boxes. Indeed, it is destined to become fairly commonplace as hardware costs fall and the technology diffuses downwards to affordable entry-level HPC. So this chapter is worthy of close study as a guide to how you may soon be programming your workstation as a low-end HPC.

6.2 Multi-tasking on shared-memory MIMD machines

6.2.1 Multi-tasking

It has already been noted that vector parallel programming has been the principal HPC tool for about twenty years and that there are real limits to what it can achieve because of the finely grained and special nature of the parallelism. Yet it may now be regarded as a historically successful approach to obtaining HPC at a time when there was no better alternative (or indeed any practical alternative). An obvious development was, and still is, to look for other opportunities to exploit parallelism at higher levels of abstraction within a program. Again this is not a new idea, but it is far harder to achieve from both a hardware and a software point of view. One reason often given for the slow take-up of parallel processing relates mainly to the apparently vast investment in non-parallel serial code or in vector parallel code, which will need to be totally rewritten for other types of parallel hardware. The fear aspect arises because this can be a non-trivial task involving considerable cost and time, *cf.* the costs of fixing the so-called millennium bug. So multi-tasking was invented by the vendors of vector supercomputers to offer an easy way of obtaining better performance while avoiding the potential problems of having to parallelise existing vector codes by rewriting them from scratch. Indeed, if you are already a vector supercomputer user then it is clearly not worth doing this rewriting if there is an easier alternative in the vector supercomputing world. This attraction was probably greater in the early 1990s, at a time when the promise of parallel processing had not fully materialised and parallel programming was not standardised. Indeed, even today there is still an ongoing need for replacement vector supercomputing hardware driven by user demands to continue using legacy codes, although in a geography and social science context this is much less of a problem, due to an absence of major legacy codes! When you are starting from scratch, the only baggage from the past is the conversion of serial code.

Multi-tasking or macro-tasking on MIMD hardware is simply defined as a computer set up so that it can allocate more than one processor to work on a single program or so that multiple programs can be run simultaneously. It is a shared-memory form of MIMD computing.

Multi-tasking opportunities can exist at different places within a program:

1 at the whole program level;
2 at the subroutine level; and
3 at the DO loop level.

Again, you are reminded that multi-tasking is a fairly coarse-grained form of parallelism. This reflects the fact that the number of processors used in the shared-memory machines that offer multi-tasking are limited, and hence best performance may be achieved by dividing the work up into fairly large chunks. Multi-tasking at the whole program level is clearly the most extreme case, but it can be very useful with very high levels of efficiency. For example, if you are running the same program a number of times (on different data), do you split the work load between multiple processors so that the time taken for a single program run is reduced or do you simply run an identical program on multiple processors with each processor working on different data? The latter is far more efficient (100 per cent) than the best of the former, but it is not always possible or relevant. Another common example occurs in Monte Carlo simulation work, where the aim might be to independently run a complete code 10,000 times. Such a code might be termed trivially parallel or embarrassingly parallel, but the only embarrassing aspect here is that it would do extremely well on this type of parallel machine offering optimal levels of performance and scaleability. Is that really a cause for embarrassment or rejoicing? Unfortunately, not many problems needing HPC can be dealt with in such a simple and efficient fashion, because the parallel parts are harder to define, occur within programs and are complex. Also, a whole-program parallelism would in practice probably be more likely to be converted into parallel subroutine calls because it gives you more flexibility (in storing the results) as well as looking better to outsiders!

Maybe it is easiest to think of multi-tasking as the equivalent to an ***outer loop*** form of parallelism and as such it is a natural complement to the vectorisation of the innermost loop. However, it is more flexible than this and can be applied to inner loops and intermediate loops as well, more or less depending on where the parallelism exists and, particularly, where the compiler thinks it can be most readily, safely and efficiently exploited. You are now liberated from the restrictions and finicky nature of vectorisation, so you can now use all manner of recursive data structures without any loss in performance, provided that the parallelism exists at a higher level than these complex data structures. Additionally, there is another attraction. The single nicest feature of multi-tasking is that typically the compiler will do most or all of the hard work for you. It will often offer an immediate performance gain without you having to do much or anything at all. Only occasionally will your program run slower than before. This form of parallel programming involves least effort, but it also provides possibly the least benefit. The maximum level of speeding-up is related to the number of available processors, and typically this will be in the range eight to 32 or perhaps 64, and even this assumes that a problem can be split into eight to 64 fairly large chunks of parallel computation. It is your responsibility to ensure that the algorithm is so conveniently parallel. This is not always easy or straightforward, and as processor speeds increase so the parallel regions have to become larger.

In theory, then, multi-tasking is transparent, fairly easy and can be automatic, provided that either the compiler or the user can identify where the 'best' or most productive parallel regions are located in the code. The hard part is ensuring that these independent parallel regions exist; that they are correctly identified as such by the compiler; that you have not accidentally or unwittingly inhibited parallelisation; that they cover the most computing-intensive sections; and that they absorb a sufficiently large fraction of total execution time to make multi-tasking worth while. Load-balancing problems (*viz.* spreading the work evenly over all the available processors) may limit efficiency as there is a need to ensure that each parallel region takes the same amount of time and that the number of such parallel regions is an integer multiple of the number of processors that are available.

To summarise, multi-tasking is a fairly crude but well-developed and mature form of simple parallel programming that only works really well on suitable problems and then offers a speed-up of less than some factor defined by the number of processors used. Remember also that on shared-memory machines there are limits to how many processors can be used because of memory conflicts. The hardest part about multi-tasking is identifying the parallel regions in the code with particular regard to the need to distinguish between those variables which are *local* to a parallel region and those which are *global* to all such regions. This problem is common to all of parallel programming and is one of the potentially most difficult aspects to master, as a code run on a single processor is executed quite differently when run on multi-processors! Fortunately, the compiler is usually sufficiently clever to do this for you, making multi-tasking straightforward and involving few code changes. It is therefore an excellent place to start considering the problems of real parallel programming. However, it always pays to examine your code to see where the compiler thought it was useful to multi-task. These are the places where, later on, you may like to consider DO loop sharing and other forms of parallel work distribution.

6.2.2 Converting or modifying programs for multi-tasking

Usually, little or no conversion will be needed for well-structured and efficiently vectorised code. However, if you believe that conversion is needed (or you think you can beat the compiler by identifying the parallel regions for it) then the following approach is one way of doing this conversion.

Step 1: Run the original serial code and keep the results. Make a note of the time taken. This is the target you need to beat.

Step 2: Vectorise the code (see the previous chapter for advice) but try to avoid hampering vectorisation by reducing the size of DO loops in anticipation of later parallelisation. Go for the best performance you can achieve on a single-processor vector machine.

Step 3: There are now three options: either let the compiler do the multi-tasking changes for you, or do them all yourself, or try a mixture of both approaches. Here it is assumed you are going to do it all, or at least consider what to do if you were going to do it yourself. You need to know this, because it is very relevant if you are writing new code from scratch and also later on when hunting for parallelism in your old serial code.

Step 4: When converting existing code, start by looking for any remaining major computational bottlenecks by profiling a typical run. Then for these 'hot spot' regions check for data dependencies and break the code up into chunks that are computationally independent and do not modify the same data during concurrent running. If you are writing new code, then try to avoid possible later problems due to data dependencies by anticipating the problems at the earliest possible stage. Unfortunately, efficiently written serial codes seldom translate easily into good parallel codes without some effort. In an extreme case, you may have to undo some of the code fiddling designed to optimise single-processor execution if, for example, it creates load-balancing problems. For optimum efficiency, each parallel task (chunk of code) needs to take the same length of time. You may also have to change your style of code writing to create or emphasise the notion of parallel regions. If in doubt then code up a simple version of your problem or a slow bit of code, run it through an automatic multi-tasking compiler, and see what problems it flags and what changes it did to your code. Then try to improve on it by responding to compiler warning messages and by thinking about exactly what you are trying to do.

Step 5: Split the computationally heavy sections of code into parallel subroutines that do a reasonable chunk of work and contain collections of other subroutine calls, various DO loops and scalar code. It is important to ensure that the logic of the program remains the same as previously despite quite major surgery and changes to structure.

Step 6: Insert calls to the parallel subroutines in your main program. Switch on all available subscript checking and debugging aids. Carefully examine compiler warning messages and alter code to respond to any compiler complaints that this or that statement has stopped parallelisation. There are some that you can never do anything about, but if there are any where you can, then do it!

Step 7: Carefully check the use of all COMMON data and if necessary remove it from the parallel regions. Separate local from global variables and handle them properly. Again the use of subroutines will help you to do this, since most variables used within a subroutine should be local to it and not available outside. The Fortran COMMON statements (and equivalents in other languages) cause great problems to compilers because the variables stored there can be updated anywhere in the program and this will often inhibit parallelisation. Fortran 90 is much better at its declarations of explicit local and global variables than Fortran 77, where it can be implicit and harder to spot.

Step 8: Debug and test the code. It is very important to ensure that the results for the original benchmark runs are identical and that the wall clock time taken is reduced! Ideally, an eight-processor run should be up to eight times faster, but even four times faster would be good. Unfortunately, when trying to debug your code many debugging aids tend not to be useful if you run into problems. The difficulty is that each processor can hold different local values of the same variables! This makes symbolic debugging very difficult, because you can easily become completely confused. As a result, the old-fashioned PRINT statement is still widely used to debug parallel code. Also, because of shared memory, different processors may be changing the same data concurrently, and it is very easy to make logical mistakes, and it is not uncommon to take a long time to detect them. The most frustrating thing about parallel bugs is that they usually go away when only one processor is used! Also, some parallel bugs can be intermittent: these are very hard to detect and once detected, to trace to the source. If the results do not match your gold standard based on a serial run then 'congratulations', you have broken the program and may need to return to the original version to detect those changes or enhancements that may be causing the problems. Finally, if you believe your parallel code to be bug-free, then think again – you could easily be wrong! So play safe and deliberately add extra code that only does checks on the accuracy of your logic.

To summarise, if you are doing the multi-tasking yourself then there are some standard procedures you can use. Landau and Fink (1993: p. 336) suggest that you look for or create:

1 independent or unrelated subroutines;
2 arrange loops with vector (inner) and parallel (outer) loops clearly defined;
3 split vector operations whereby long DO loops are divided up, allowing vectorisation on different processors; and
4 create long vectors by collapsing multi-dimensional arrays into a one-dimensional form. This is probably the only change that a modern compiler will not now do for you, because quite often it is a complete pain requiring large numbers of code and algorithmic changes. You only do it if you really feel you have to, because it is an excellent way to stop a perfectly good program working for at least a while. You probably also think that programming languages let you have multi-dimensional arrays to make your life easier. Well, that is true, but if you want to exploit this sort of HPC hardware and push it to the limits of what it can do, then maybe the 'gain' will justify the 'pain'; or else forget it!

Finally, remember that there are also many more mind-bogglingly new problems and potential bugs that lurk in a parallel world just waiting for you to come along. If you like a challenge, then start thinking in parallel. Likewise, when you find a parallel bug then write it down while the pain is still present, because there is a better than evens chance that you could manufacture the same bug again.

6.2.3 Programming in parallel

In general, parallel bugs are often much harder to squash than serial ones. Thinking in parallel can be tricky because of multiple instances of the same variable. For example, consider the apparently simple task of summing an array in the following fragment of code:

```
SUM=0.0
DO I=1,1000
  SUM=SUM+X(I)
ENDDO
```

This will work perfectly well on a single processor. Now assume that the 1000 data values in this loop are to be shared between two processors; index I values 1 to 500 are run on one processor and index values 501 to 1000 on the other. The question now is how to obtain the correct result in the variable SUM? On a Cray J90, you could do this as follows:

```
CMIC@ DO PARALLEL (I) SHARED(SUM)
    SUM=0.0
    DO I=1,1000
      SUM=SUM+X(I)
    ENDDO
```

The compiler shares out the work for you, which is nice. Unfortunately, SUM is a shared *global* variable (just as it would be if stored in COMMON) that the DO loops on each processor will want to change at the same time, leading to potential memory conflicts. In practice, the hardware or software may be clever enough to avoid this problem by ensuring that each access to the variable SUM is synchronised. This is very nice of it and it gets the answer right, but it can result in a horrendous loss of efficiency because it is done 1000 times, and each synchronisation forces one processor to wait until the other is finished. The result is a program that probably now runs more slowly on two processors than on one. Yes, it is parallel slowdown!

A better approach is as follows:

```
CMIC@ DO PARALLEL (I) SHARED(SUM) PRIVATE(LSUM)
          LSUM=0.0
          DO I=1,1000
          LSUM=LSUM+X(I)
        ENDDO
CMIC@ GUARD
        SUM=SUM+LSUM
CMIC@ END GUARD
```

The GUARD directive ensures that the local processor-specific results held in LSUM are added to the global shared variable SUM one at a time at the end of the shared DO loop. This version runs much faster, because this critical memory write only occurs outside the loop and not within it, so it occurs only twice instead of 1000 times. If all this seems difficult then you can let the compiler do it for you. At the very least it will provide many interesting suggestions as to how to parallelise your code by generating the code changes that it thinks are necessary. It is, therefore, actually a good way of teaching yourself some of the basics of parallel programming! However, a compiler is seldom as clever as the human programmer, because it has no innate intelligence or knowledge of the algorithm and sees only the code you give it. It will do the best it can with the code you have given it. However, it has to preserve the logic of the original code, but quite often it is only by modifying the logic of the algorithm that optimal levels of performance can be achieved. Only you are clever enough to perform that task. Remember that if the compiler gets confused it will do nothing and leave the code unchanged.

6.2.4 Case study 1: a multi-tasking GAM example

Let us start with the GAM. The full GAM in its fully optimised vectorised form is run on a ten-processor Cray J90. This produces the results in Table 6.1. Note that on the final highly optimised version of the GAM it took 73.5 seconds and ran at 5.5 Mflops to complete the full problem. An equivalent run on a single processor took 261 seconds and ran at 18.7 Mflops. This is somewhat less than the optimal speed-up, but it is not bad.

Table 6.1 GAM test problems on Cray J90 with ten processors using multi-tasking.

Algorithm	*Cray J90 CPU time*	*Cray Mflops/s*	*Multi-tasking time (10 processors)*
version 1	195.0	8.5	>3600
version 2	49.3	22.0	2001.1
version 3	49.3	39.0	1283.6
version 4	41.3	24.0	117.9
version 5	77.2	38.9	1394.4
version 6	27.1	18.0	101.4
version 7	1.6	9.5	73.5
version 8	1.2	8.6	55.3

It is useful to see what the compiler did to ensure safe execution on multiple processors and also where it found the parallelism to be exploited. Appendix 6.1 shows the coding for Version 1 of the GAM and Appendix 6.2 that for Version 8. This is also indicative of the sorts of changes to the code that are needed for other types of shared-memory computing.

6.2.5 Case study 2: a multi-tasking origin-constrained spatial interaction model

Here the single-processor time was 2.95 seconds at 65 Mflops. The ten-processor time was 1.44 seconds at 61.2 Mflops. Appendix 6.3 gives the modified spatial interaction model code after the multi-tasking compiler had finished with it. Note that the best performance was achieved with the original unrolled version. Far less modification is needed here, because the code is already in a data parallel form. The principal problem is the lack of sufficient computational work to keep the machine busy. The arithmetic load is N^2, but there is insufficient memory to allow the processing of problems which are large enough to keep the J90 processors fully loaded. This is another reason why this form of parallel computing has a restricted future.

6.3 Parallelisation strategies

6.3.1 Hunting for parallelism

Anything more complex than multi-tasking or data-parallel problems requires the user to start to look for or to create parallel regions within code. Landau and Fink (1993: p. 332) have this to say:

> The key to parallel programming is to identify where your program will benefit from parallel execution. To do that the programmer must understand the program's data structures at a level similar to that of vector processing, must know how to synchronise the results generated by different processors, and must assign tasks to different processors of approximately equivalent numerical intensity (balance the load).

The secret is to start thinking in a parallel but non-finely grained way! You need to look for large-scale parallelism, and far more opportunities exist than you may at first believe possible or than you found in the vectorisation world or by studying multi-tasking compiler reports. However, if you cannot find any or enough parallelism in your existing algorithm or code then you will have to change one or the other, or both, so that you can. Otherwise, your career as a parallel programmer will come to a very speedy end, or you will be forever limited to a relatively simple range of problems.

So you need to start structuring and restructuring algorithms and code for parallelism, and this involves much more than 'merely' tuning inner DO loops and finding shareable outer DO loops. Porting may now be much less of an automatic activity. It may well require considerable extra work as you redesign, rethink and rewrite code that once worked fine in a serial or in a vector or multi-tasking environment; but then the gains are also potentially far greater. The aim is to structure code in such a way that there is enough work to keep multiple processors as fully employed as is possible with a minimum of idle time due to lack of work or communications delays.

6.3.2 Problem decomposition

Baker and Smith (1996: p. 91) write, 'Identifying parallelism in an existing algorithm is usually done by examining the existing code in detail and determining which parts can be easily modified to run in parallel'. A key question is whether or not to preserve the algorithmic structure of the serial code. However, do not get too sentimental here! Convenience is relevant, but efficiency might be more important in an HPC context. Those parts of the algorithm that cannot be run in parallel need to be changed or redesigned and maybe even entirely new parallel algorithms devised. The job of identifying and splitting out the parallel parts for execution on multiple CPUs is called problem decomposition. Dowd (1993: p. 294) writes: 'Once a problem has been decomposed for multiple processors, people say it has been *parallelised*'. The challenge for algorithm writers is how to design new algorithms that have this property built in. The difficulty is that while some problems are clearly parallel and are easily coded for a parallel processor, others may be almost entirely serial in nature and require completely new algorithms before they will work well. Yet others are parallel in nature, but the parallelism has been destroyed by serial thinking and is now buried or obscured by the method used or by the way it was programmed. Thinking in parallel is not easy until you train yourself to do it; then it is much easier than you thought it would ever be.

There are some basic principles that may help here. There are two general approaches to problem decomposition for the distribution of work between multiple processors:

1 data decomposition; and
2 control or functional decomposition.

In data decomposition, the data are partitioned into pieces and each chunk is distributed to a separate processor. Each processor does the same calculation but on a different chunk of data. In this way, the distribution of data is effectively a sharing of the computational load. This is essentially a data parallel and multi-tasking route. This works well on regular problems but tends to work far less well on irregular problems, which result in unbalanced load distribution.

In control decomposition, different processors are given different tasks to perform or are repeatedly reassigned new tasks as soon as they become idle. This is a more useful and more general form of parallelism, since it requires that each

processor can act independently. However, it is more difficult, and it too may also have problems in scaling. Dowd (1993: p. 299) identifies the problem as follows: 'how do you take four processors doing four separate jobs and scale the computations up to eight processors?' Once your algorithm scales well to eight processors (i.e. wall clock times diminish linearly with increasing numbers of processors), then consider 64 or 256 or 1024! Ideally, you need to build scaleability into the algorithm. The approach used depends on the problem and the available hardware. Another way of viewing this design task is in terms of how to distribute the workload of an algorithm evenly so that it can be processed in parallel. If the parallelism is not self-evident, then you have to 'create it'.

Load balancing is a very important concept here. The work needs to be divided up fairly or evenly distributed, because the speed of the program will depend on the time taken by the processor that takes the longest time. This might also be termed the convoy principle in that the speed of a convoy of ships is determined by the speed of the slowest.

Imagine you have N cases to process, and N is fairly large. The cases might be aspatial, but they could also be spatially structured. Likewise, the computation to be applied to each case could involve just one case or its immediate spatial neighbours. The task now is how best to divide up the N cases into K chunks, where K is some number greater than, or better still, equal to, the number of available processors. Mappable data are often readily decomposed using various grid-based or two-dimensional mesh-based structures; see Trewin (1998).

More generally, Wilson (1995: pp. 46–49) identifies five broadly different approaches that can be used to parallelise applications or assist in developing new algorithms. They are:

1 geometric
2 iterative
3 recursive
4 speculative
5 functional

In *geometric decomposition*, the problem is disaggregated into tasks that reflect some physical subdivision of the system being modelled. Each processor is assigned a spatial domain plus a boundary. This offers good scaleability if it can be done and will make good use of distributed memory, but it is really only suitable for certain types of problem. Fortunately, a geometric decomposition approach is highly applicable to most of the two-dimensional map-based processing in GIS, which is naturally parallel. This will also work well when there is a strong locality or neighbourhood effect that can be exploited. For geographers, the map is a wonderfully parallel search space. Virtually any and all applications of search over a two-dimensional map, operations performed on a map or processing relating to the modelling of a spatial system will be parallel in one form or another. As Openshaw and Openshaw (1997) note, you can become smarter in many spatial analysis and modelling tasks not only by becoming more sophisticated in terms of the technology being used but also by engaging in a more finely grained spatial search. This will work provided that the parallelism in the search is fully exploited and that there are not more than about a few thousand million locations to examine! Additionally, the typically positively spatially auto-correlated nature of much map data brings other benefits. In particular, locality effects are very strong, so that when data dependencies exist they are localised. This can be a particularly nice aspect to have from a parallel programming point of view as it can reduce communications traffic.

However, the decomposition strategy need not only be based on data: most are based on dividing up algorithms into parallel chunks. *Iterative decomposition* involves breaking down an algorithm in which one or more operations is repeatedly applied by executing these operations in parallel, or at least redesigning the program so that this occurs. Simulation models, parallel simulated annealers and genetic algorithms can all benefit from this decomposition strategy. A *recursive approach* involves breaking a problem down into parts that can be handled directly or further broken down into yet smaller parts. However, there is often a very strong relationship between the degree of parallelism and the amount of work being done! *Speculative decomposition* involves performing many independent calculations concurrently and then using the results of the first one to complete, thereby stopping and discarding the remainder. Some optimisers and search methods work like this. The search continues until an improvement is found; then it restarts around the current best. This is more a method of search than a decomposition strategy and appears computationally wasteful, but it is mainly concerned with improving the quality of the results rather than speeding it up.

Finally, *functional decomposition* involves dividing up the functions of the program and running all the parts concurrently. This is very nice if it can be done at a sufficiently coarse-grained level as it promises maximum levels of performance. A very common method is the task farm or slave–master approach (Foster, 1995). A classical master–slave approach is for one processor (the master) to divide up the work and send a chunk to each processor (the slaves); it does not matter if the chunks are of uneven size. When a slave processor has finished its allotted task, it sends back the results and asks for more work. This approach is widely used in message passing; see Chapter 7. It tends to produce good load balancing. It works best when the number of tasks is large in relation to the number of processors (say a factor of several thousand or more) and each involves sufficient computation work so that the interconnection used to communicate between the master and slaves is not overloaded. How this works in practice is discussed later.

6.4 Data parallel programming

6.4.1 *The attractions*

An alternative approach to multi-tasking at the macro scale and of parallelising algorithms is to consider whether the problem can be handled in a simple data parallel manner. Data parallel processing is where one operation (or set of operations) is applied to all elements of the data simultaneously. It is termed 'data parallel' because the data (and hence the associated computation) is distributed out to the processors. This approach was once restricted to SIMD machines but is now far more widely applicable because of the appearance of array operations in high-level languages. Originally, these language extensions were vendor-specific and were used to program array processors of the ICL DAP and connection machines in the mid-1980s and early 1990s. However, more recently these array-processing extensions have been absorbed into standard programming languages such as Fortran 90 and more particularly High-Performance Fortran (HPF).

Data parallel programming is almost as easy (some would say easier) as vector parallel programming, but it is a more general model than the vectorisation process and thus offers more opportunity for parallelism to be exploited. It operates at or within the DO loop. In its simplest form, it requires that all DO loops vectorise and the entire program consists of array or vector operations or that it can be recast into that format. It is therefore extremely well suited to types of scientific programming that involve arithmetic operations performed on large arrays of data, *viz.* physics, seismic analysis, weather forecasting and weapons development.

So in data parallel programming the parallelism is defined by the distribution of the data arrays in the parallel memory space. At its simplest, this name is given to code that can be expressed as a sequence of array operations. The advantages are (1) this can make the programs easier to read and write; (2) high-level data parallel programming languages exist that leave the implementation on specific hardware to the compiler, allowing the programmer to concentrate on the code and algorithm; and (3) it should yield results that provide optimal performance for a particular machine while still being portable. The problem is that many algorithms are not suitable for this approach.

6.4.2 *Some examples of data parallelism*

Typically, in a data parallel programming language there is a close correspondence with array notation, but this is not the same as matrix algebra. Consider, for example, the statement:

```
A=B*C
```

where A, B and C are arrays. This is *not* the matrix algebra interpretation that matrix B is post-multiplied by matrix C. Instead, it specifies the multiplication of each element of B by each element of C, the result being stored element by element in A. So this single statement performs the equivalent of the following Fortran code:

```
DO I=1,N
DO J=1,N
  A(I,J)=B(I,J)*C(I,J)
ENDDO
ENDDO
```

where N is the dimension of arrays A, B and C. Your programs instantly become shorter and more compact. This is useful, because experience suggests that short programs are much easier to debug than long, complex ones. Note that the precise implementation of the code is left to the compiler, which is presumed to know how best to optimise the order of memory accessing, etc. for specific hardware. This could be very efficient or very poor, depending on the life stage and maturity of the compiler. Note that the inter-processor data communication is implicit and invisible to the user, since in practice the simple statement A = B * C may require access to elements of these arrays held on many different processors. Synchronisation is also implicit. Finally, data parallelism extends to include reduction operations (i.e. a global sum). A surprisingly large number of problems can be expressed in a simple data parallel form, which makes this a very easy path to parallel programming that is particularly appealing for many statistical and mathematical modelling applications. Monte Carlo simulation and other computationally intensive statistical methods are easily implemented.

Both Fortran 90 and HPF offer some nice array syntax. For example, to sum all elements of array X:

```
REAL : : C
REAL, DIMENSION(100,100) : : X
C=SUM(X)
```

There is also a parallel IF, called WHERE, that can also be used on arrays; for example,

```
WHERE (X .GE. 0.0) X=ALOG(X)
```

In both cases, array X has 100 rows and 100 columns, so this operation is performed 10,000 times. Programming with arrays in High-Performance Fortran is an even more useful approach. An international standard for HPF has now been defined, but there are as yet few fully compliant compilers. However, the benefits of simplicity in coding reflect a fairly restricted form of parallelism. Matrix operations and mathematical expressions are most easily handled, but more complex algorithms may create problems due to inflexibility. Additionally, you have to trust that the compiler will do its work well and correctly, and this at present may sometimes be premature.

These types of data parallel programming problems are often termed ***embarrassingly parallel***. However, this is a very useful attribute if you wish to exploit parallel hardware fully, since it offers a good prospect for a dramatic performance gain without any great need for algorithm redevelopment. Such problems are easily programmed, although not all or many applications can be expressed in this form. The words 'embarrassingly parallel' may imply triviality and a lack of challenge. However, this is both unfair and misleading. It is what you do with the code that is important, not how it appears. The nicest thing about embarrassingly parallel code is that it runs at an embarrassingly parallel speed! In fact, it could be suggested that it is other users who should be embarrassed by the absence or poor levels of parallelism in their own code.

However, even here parallel programming is seldom completely simple, and it is necessary to worry about how the arrays are defined, the nature of the operations being performed, etc. if you are to squeeze out the best performance. The main disadvantage is the difficulty of handling irregular or heterogeneous computation. Also, not all (or indeed many) problems can be expressed in a pure data parallel form. Some will require considerable effort to restructure the algorithms and remove data dependencies that would ruin performance. However, in two to five years time this situation may be quite different as compilers improve and become smarter in how they do their business. So watch this space!

6.4.3 Data parallel GAM

Let us return to the case studies used in the previous chapter. Unfortunately, there is an instant snag. The GAM is not a straightforward data parallel problem and cannot be expressed in an efficient data parallel form. So we shall move on to look at the spatial interaction model, which is easily expressed in a data parallel way.

6.4.4 Data parallel spatial interaction model

The spatial interaction model is fortunately a very straightforward data parallel problem. At its simplest, this is equivalent to multi-tasking and data parallel programming in which complete sets of DO loops are parallelised. The DO PARALLEL compiler directive was one of the earliest forms of parallel programming with a shared memory. It is very simple, but remember that the potential performance gain is limited to some factor less than the number of processors, and the number of processors on non-distributed-memory machines tends to be limited.

The spatial interaction model written in HPF is given in Appendix 6.4. Note how short this code is compared with previous versions, even allowing for the data generation step being replaced by reads. The only changes made to the serial source code presented earlier are to insert some compiler directives. These are:

```
!HPF$ distribute (BLOCK,*) : : T,C
```

Now that was not hard! Yet this instructs the compiler that the arrays T and C are to be distributed between the processors with the first dimension split in blocks. The directive !HPF$ INDEPENDENT tells the compiler that the following loop can be parallelised (or executed independently). Since both these directives look like a comment to a normal Fortran 90 compiler, the same code can be compiled and run on a serial machine without difficulty. The intrinsic function SUM can be either a serial subroutine on a uniprocessor machine or a parallel subroutine on a multi-processor machine; however, since it is provided by the compiler writer you do not need to know what the best method for summing an array on any particular machine is: you just use the intrinsic function and hope that the compiler gets it right (it usually will).

There are some performance issues to be considered about how you divide up the data between different processors. You as the programmer must make these decisions, since the compiler cannot tell how you intend to use the data. HPF provides two ways to divide up an array: cyclic and block. In a *cyclic data distribution*, element 1 of the array goes on processor 1, element 2 on processor 2 and so on until element n goes to the n^{th} processor, then element $n + 1$ goes to processor 1 and so on. In a *block distribution*, elements 1 to m go on processor 1, where m is the size of the array divided by n, the number of processors. Each processor is allocated m elements of the array, although processor n gets the remainder of the elements, which may be less than m; for instance, with four processors and ten elements, processors 1, 2 and 3 will be given three elements and processor 4 will be given one. Obviously, this will lead to some inefficiencies as processor 4 will always run out of work before the first three processors. If, however, a cyclic distribution is used then processors 1 and 2 would each receive three elements of the array and processors 3 and 4 would receive two elements, which while still not perfect is much better than the block method. See Figure 6.1 for an illustration of these storage patterns.

The most fundamentally important aspect of data distribution is that it affects the performance of the program. For example, if you have a two-dimensional array, A, with N rows and N columns, which of the two data-distribution strategies is likely to be the more efficient if you wish to sum the first column? Let us assume that $N = 1000$ and the number of processors is ten.

If you thought cyclic then you would be wrong, because block distribution would be better. The reason is that data are stored in Fortran arrays in column order. Cyclic data distribution scatters the data in each column over many processors, whereas block keeps it in contiguous chunks. Maybe you thought that the parallel processor could access any data more or less simultaneously. Well this is almost true on shared-memory machines, apart from the usual memory latency. However, it is certainly not true of a distributed-

array element	1	2	3	4	5	4	7	8	9	10
cyclic processor allocation	1	2	3	4	1	2	3	4	1	2
block processor allocation	1	1	1	2	2	2	3	3	3	4

Figure 6.1 Cyclic and block distribution of data elements.

memory machine, where data are distributed and results received as messages travelling along a processor interconnection network. In HPF, the code is identical, but on a distributed-memory machine the compiler is, unknown and invisible to you, generating vast amounts of inter-processor messages. Remember the golden rule of distributed-memory machines is that code which generates the fewest messages is probably best. So the layout of the data can be crucial, because if you distribute it badly you may cause your program to run far more slowly than otherwise. Incidentally, this is also true, albeit to a far lesser degree, of all computers. Processors prefer their memories to be accessed sequentially, and parallel machines with distributed memories slow down by large factors once they have to perform lots of memory reads and writes to non-local memory attached to other processors.

6.5 Conclusions

This chapter has described ways of parallel programming without too many tears or too much effort. These simple approaches may be sufficient for most of the problems of operating in parallel to emerge, if any are likely to do so. The main criticisms have been that the resulting code tends to be non-portable because of vendor-specific language add-ons to Fortran and C to handle the shared DO loops and a corresponding lack of universal standards. However, the standardisation of Fortran 90 is very important as it includes data parallel syntax. Additionally, the emergence of High-Performance Fortran seems likely to provide an easier way of programming both shared- and distributed-memory MIMD and SIMD machines with non-uniform memory access (NUMA) times. The attractions are that the resulting code is portable and far easier to write and considerably less complex than message passing. Also, right now NUMA architectures are becoming increasingly popular, so this broad category of hardware looks set to become quite popular in the early years of the twenty-first century. Additionally, the multi-tasking, shared DO loops and vectorisation models of parallelism can all be handled. A possible disadvantage is that there can be some concern about loss of performance, since there is an assumption of synchronisation. Sooner or later the processors have to stop and wait for the full set of concurrent tasks to be completed, and this assumes an evenness of the parallel tasks. Maybe a data parallel approach will always be regarded as being too simple by the parallel-processing connoisseur, but is this really significant? It depends on the problem. For some applications, nothing else will be needed and considerable complexity can be avoided. For other problems, it will not work at all.

There is an increasingly strongly held belief that this type of parallel programming will sooner rather than later become the dominant model via high-level languages such as HPF. Yet for the cautious, the safest advice would be to use message passing based on a standard library, as described in Chapter 8. In thirty years time this will probably still work, whereas HPF might well have been and gone. On the other hand, HPF seems to be attracting an increasing degree of vendor support. For some applications, HPF may well provide a very good and quick solution, and since it is a high-level language it is easily used. Maybe there is a whole class of geographical problems that are suitable for this approach. If you are a beginner, then using HPF would be a good place to start your parallel-programming career.

Appendix 6.1: multi-tasking code of GAM Version 1

The interesting sections are those flagged by the compiler as

PR	parallel region
P	parallel DO loops
V	vectorisable code
CritReg	critical regions where local values are copied to global storage

```
                  C* gam_1.f
                        IMPLICIT NONE
                        DOUBLE PRECISION D1V(3000)                                    Generated,Xref
                        REAL QQQ                                                      Generated,Xref
                        TASK COMMON/Z1FPP1 CM/ QQQ(8191)                              Generated,Xref
                        EQUIVALENCE (QQQ(1),D1V)                                      Generated,Xref
                        INTEGER NCASE
                        PARAMETER (NCASE=150 000)
                        DOUBLE PRECISION OVERAT,XMINE,XMAXN,XMINN,XMAXE,
                       X  OBSP,OBSC,RADIUS,DIS,CX,CY,PROB,
                       X  X(NCASE),Y(NCASE),P(NCASE),C(NCASE),
                       X  RADINC,RADMAX,RADMIN,POPMIN,CANMIN,THRESH

                        CHARACTER*100 XYDATF,PCDATF,OUTFIL
                        INTEGER I,LOOP,ICOL,IROW
                        INTEGER TOTCAL,TOTDAT,TOTNCS,TOTNHY,STEP,ID,N,
                       X  MINN,MAXN,MINE,MAXE,NTIMES, NCALC,NDAT,NCALS,NHY,
                       X  N2
                  C*
                        INTEGER I1X, JA, J                                            Generated,Xref
                        DOUBLEPRECISIONCUMPRB(3000),CONS(3000),AMEAN,OBSC1X,          Generated,Xref
                       X OBSP1X,X,MIN1X, XMIN2X, XMAX1X, XMAX2X, OBSP2X, OBSC2X       Generated,Xref
                        WRITE(6,78001)
                  78001 FORMAT('*Geographical Analysis Machine GAM/1 (Feb 1997)'//)

                  C* Step 1. Read Data==============
```

```

                  C* set constants

                  C* read ini.file
                        OPEN(UNIT=1,FILE='gamfiles.dat',FORM='FORMATTED',
                       X  STATUS='OLD')

                  C* read USER DATA file names
                  C.. this file contains X,Y data
                        READ(1,10001) XYDATF
                  C.. this file contains Pop at Risk and Count of Real Cases
                        READ(1,10001) PCDATF
                  10001 FORMAT(A)
                  C* get output results file name
                        READ(1,10001) OUTFIL
                        CLOSE(UNIT=1,STATUS='KEEP')


                  C* read X-Y data
                        WRITE(6,6707) XYDATF
                  6707  FORMAT('*User Input X,Y File is;',A)
                        OPEN(UNIT=1,FILE=XYDATF,
                       X  STATUS='OLD',
                       X  FORM='FORMATTED')

P--               CMIC@ DO ALL GUIDED(64) SHARED(X, Y) PRIVATE(I)             Generated
P                 CDIR@ IVDEP                                                 Generated
P v --------            DO I=1,NCASE
P v                     X(I)=0.0
P v                     Y(I)=0.0
P v ------->            ENDDO
                        N=0

         1 --------     DO I=1,NCASE
         1              READ(1,*,END=999) ID,X(ID),Y(ID)
         1              N=I
         1 ------->     ENDDO
                  999   WRITE(6,123) N
                  123   FORMAT(5X,'*EOF at Case Number',I10)
                        CLOSE(UNIT=1,STATUS='KEEP')
                  C* No data read?
                        IF(N.EQ.0) STOP 1

                  C* read Population and Observed Cancer data
                        WRITE(6,6708) PCDATF
                  6708  FORMAT('*User Input Data File is;',A)
                        OPEN(UNIT=1,FILE=PCDATF,
                       X  STATUS='OLD',
                       X  FORM='FORMATTED')

P--               CMIC@ DO ALL GUIDED(64) SHARED(P, C) PRIVATE(I)             Generated
P                 CDIR@ IVDEP                                                 Generated
P v                     DO I=1,NCASE
P v                     P(I)=0.0
P v                     C(I)=0.0
P-v                     ENDDO
                        N2=0
          1 ----        DO I=1,NCASE
          1             READ(1,*,END=199) ID,C(ID),P(ID)
          1             N2=I
          1 ----        ENDDO
                  199   WRITE(6,123) N2
                        CLOSE(UNIT=1,STATUS='KEEP')
                  C*No data read?
                        IF(N2.EQ.0) STOP 2

```

```
94                  C* Files do not match
95                        IF(N.NE.N2) STOP 3
96                  C* go thru data and produce counts
97                        OBSP=0.0
98                        OBSC=0.0
99 PR               CMIC@  PARALLEL IF(N.GT.1200)SHARED(C,P,N,OBSC,OBSP)       Generated
100 PR              CMIC@1 PRIVATE(I OBSC1X, OBSP1X)                           Generated
101 PR                    OBSC1X = 0                                           Generated,Xref
102 PR                    OBSP1X = 0                                           Generated,Xref
103 PR P-           CMIC@ DO PARALLEL GUIDED(64)                               Generated
104 PR P            CDIR@ IVDEP                                                Generated
105 PR P v ---            DO I = 1, N                                          Generated,Xref
106 PR P v                OBSC1X = OBSC1X + ABS@(C(I))                         Generated,Xref
107 PR P v                OBSP1X = OBSP1X + P(I)                               Generated,Xref
108 PR P-v -->            END DO                                               Generated,Xref
109 CritReg         CMIC@ GUARD                                                Generated
110 CritReg         OBSP = OBSP + OBSP1X                                       Generated,Xref
111 CritReg         OBSC = OBSC + OBSC1X                                       Generated,Xref
112 CritReg         CMIC@ END GUARD                                            Generated
113 PR              CMIC@  END DO                                              Generated
114 PR              CMIC@ END PARALLEL                                         Generated
                          DO I=1,N                                             Deleted,NoXref!
                          OBSC=OBSC+ABS(C(I))                                  Deleted,NoXref!
                          OBSP=OBSP+P(I)                                       Deleted,NoXref!
                          ENDDO                                                Deleted,NoXref!
115                       WRITE(6,8) N,OBSP,OBSC
116                 8     FORMAT(
117                     X   ' *Number of input data records: ',I8/
118                     X   ' *Total population at risk: ',F10.0/
119                     X   ' *Total Cases ',F10.0)
120                       IF(OBSP. EQ.0.0.OR.OBSC.EQ.0.0)STOP 3
121                       OVERAT=OBSC/OBSP
122
123                 C* Find Min and Max X,Y values to define search region
124                       XMINE=999999999.0
125                       XMINN=999999999.0
126                       XMAXE=0.0
127                       XMAXN=0.0
128
129                 C* data are in 1 km units
130 PR              CMIC@ PARALLEL IF (N .GT.900)SHARED(X,Y,N,XMINE,XMINN,XMAXE,
131 PR              CMIC@1 XMAXN)PRIVATE(I,XMIN1X,XMIN2X,XMAX1X,XMAX2X)       Generated
                          DO I=1,N                                             Deleted,NoXref!
                          XMINE=DMIN1(XMINE,X(I))                              Deleted,NoXref!
                          XMINN=DMIN1(XMINN,Y(I))                              Deleted,NoXref!
                          XMAXE=DMAX1(XMAXE,X(I))                              Deleted,NoXref!
                          XMAXN=DMAX1(XMAXN,Y(I))                              Deleted,NoXref!
                          ENDDO                                                Deleted,NoXref!
132 PR
133 PR              C* data are in 1 km units                                  Generated
134 PR                    XMIN1X = XMINE                                       Generated,Xref
135 PR                    XMIN2X = XMINN                                       Generated,Xref
136 PR                    XMAX1X = XMAXE                                       Generated,Xref
137 PR                    XMAX2X = XMAXN                                       Generated,Xref
138 PR P            CMIC@ DO PARALLEL GUIDED(64)                               Generated
139 PR P            CDIR@ IVDEP                                                Generated
140 PR P v                DO I = 1, N                                          Generated,Xref
141 PR P v                XMIN1X = DMIN1(XMIN1X,X(I))                          Generated,Xref
142 PR P v                XMIN2X = DMIN1(XMIN2X,Y(I))                          Generated,Xref
143 PR P v                XMAX1X = DMAX1(XMAX1X,X(I))                          Generated,Xref
144 PR P v                XMAX2X = DMAX1(XMAX2X,Y(I))                          Generated,Xref
145 PR P-v                END DO                                               Generated,Xref
146 CritReg         CMIC@ GUARD                                                Generated
147 CritReg               XMAXN = DMAX1(XMAXN,XMAX2X)                          Generated,Xref
148 CritReg               XMAXE = DMAX1(XMAXE,XMAX1X)                          Generated,Xref
149 CritReg               XMINN = DMIN1(XMINN,XMIN2X)                          Generated,Xref
```

```
150 CritReg               XMINE = DMIN1(XMINE,XMIN1X)                          Generated,Xref
151 CritReg         CMIC@ END GUARD                                            Generated
152 PR              CMIC@ END DO                                               Generated
153 PR              CMIC@ END PARALLEL                                         Generated
154
155                       WRITE(6,7123) OVERAT,XMINE,XMAXE,XMINN,XMAXN
156                 7123  FORMAT(
157                     X   ' *Global Incidence Rate per Population at Risk is ',F12.8/
158                     X  ' *Minimum Easting is',F12.1,' Maximum is',F12.1/
159                     X  ' *Minimum Northing is',F12.1,' Maximum is',F12.1)
160                       MINN=XMINN-1.0
161                       MINE=XMINE-1.0
162                       MAXN=XMAXN+1.0
163                       MAXE=XMAXE+1.0
164
165                 C* Step 2. Set Search Parameters=========
166
167                 C* Circle Radii are in KM
168                       RADMIN=10.0
169                       RADMAX=10.0
170                       RADINC=1.0
171
172                 C* select probability threshold
173                       THRESH=0.005
174
175                 C* set minimum circle size
176                       POPMIN=100.0
177                 C* set minimum cancer count size
178                       CANMIN=2.0
179
180                 C* write search parameters out
181                       WRITE(6,76541) RADMIN,RADMAX,RADINC,POPMIN
182                 76541 FORMAT('*Minimum Circle radius is',F10.3,' 1 km'/
183                     X   '*Maximum Circle radius is',F10.3,' 1 km'/
184                     X   '*Circle increment set to',F10.3,' 1 km'/
185                     X   '*Minimum POPULATION size is',F10.0)
186
187                       WRITE(6,78234) THRESH
188                 78234 FORMAT(' *Significance THRESHOLD set at',F12.6)
189
190                 C* other global inits
191                       TOTCAL=0
192                       TOTDAT=0
193                       TOTNCS=0
194                       TOTNHY=0
195
196                 C* convert all population counts into expected values
197 P--             CMIC@ DO ALL GUIDED(64)IF(N.GT.1800)SHARED(N,OVERAT,P) PRIVATE(I)    Generated
198 P               CDIR@ IVDEP                                                          Generated
199 P v ---               DO I=1,N
200 P v                   P(I)=P(I)*OVERAT
201 P v ---               ENDDO
202
203                 C* reset minimum value
204                       POPMIN=POPMIN*OVERAT
205
206                 C* SET INITIAL RADIUS for circles
207                       RADIUS=RADMIN-RADINC
208
209                 C* compute number of circle sizes to be examined
210                       NTIMES=(RADMAX-RADMIN)/RADINC+1.0
211
212                 C* open output file
213                       OPEN(UNIT=9,FILE=OUTFIL,STATUS='UNKNOWN',
214                      X   FORM='FORMATTED')
215
```

```
216                      C* Step 3. Circle Size Loop==============
217
218                      C* *********Circle SIZE loop starts here **
219           1 --              DO LOOP=1,NTIMES
220           1          C* set circle radius
221           1                 RADIUS=RADIUS+RADINC
222           1          CC STEP=RADIUS*0.2+0.5001
223           1                 STEP=RADIUS
224           1                 IF(STEP. EQ.0) STEP=1
225           1
226           1                 NCALC=0
227           1                 NDAT=0
228           1                 NCALS=0
229           1                 NHY=0
230           1
231           1          C* Step 4. Grid Search: Northing Loop====
232           1
233           12 --             DO 100 IROW = MINN, MAXN, STEP
234           12                CY=IROW
235           12
236           12         C* Step 5. Grid Search: Easting Loop===
237           123 -             DO 200 ICOL = MINE, MAXE, STEP
238           123               CX=ICOL
239           123
240           123        C* GET DATA WITHIN CIRCLE AT (IROW, ICOL)
241           123               NCALC=NCALC+1
242           123
243           123        C* Step 6. Get Data for Circle=========
244           123               OBSP=0.0
245           123               OBSC=0.0
246 PR        123        CMIC@ PARALLEL SHARED(N,CX,CY,RADIUS,OBSP,OBSC,X,Y,P,C)      Generated
247 PR        123        CMIC@1 PRIVATE(I, DIS, OBSP2X, OBSC2X)                       Generated
248 PR        123               OBSP2X = 0                                            Generated,Xref
249 PR        123               OBSC2X = 0                                            Generated,Xref
250 PR P--    123        CMIC@ DO PARALLEL GUIDED(64)                                 Generated
251 PR P      123        CDIR@ IVDEP                                                  Generated
252 PR P      123v -----DO I = 1, N                                                   Generated,Xref
                                DO I=1,N                                              Deleted,NoXref!
253 PR P      123v       C* calc distance of ED at X(I), Y(I) from grid point CX,CY
254 PR P      123v              DIS = (X(I)-CX)**2 + (Y(I)-CY)**2                     Generated,Xref
255 PR P      123v              IF (DIS .GT. 0.0) DIS = DSQRT(DIS)                    Generated,Xref
                                DIS=(X(I)-CX)**2+(Y(I)-CY)**2                         Deleted,NoXref!
                                IF(DIS.GT.0.0) DIS=DSQRT(DIS)                         Deleted,NoXref!
256 PR P      123v       C* is point inside circle?
257 PR P      123v              IF (DIS .LE. RADIUS) THEN                             Generated,Xref
                                IF(DIS.LE.RADIUS)THEN                                 Deleted,NoXref!
258 PR P      123v       C* yes so accumulate counts
259 PR P      123v              OBSP2X = OBSP2X + P(I)                                Generated,Xref
260 PR P      123v              OBSC2X = OBSC2X + C(I)                                Generated,Xref
                                OBSP=OBSP+P(I)                                        Deleted,NoXref!
                                OBSC=OBSC+C(I)                                        Deleted,NoXref!
261 PR P      123v              ENDIF
262 PR P--    123v---->         END DO                                                Generated,Xref
263 CritReg   123        CMIC@ GUARD                                                  Generated
264 CritReg   123               OBSC = OBSC + OBSC2X                                  Generated,Xref
265 CritReg   123               OBSP = OBSP + OBSP2X                                  Generated,Xref
266 CritReg   123        CMIC@ END GUARD                                              Generated
267 PR        123        CMIC@ END DO                                                 Generated
268 PR        123        CMIC@ END PARALLEL                                           Generated
                                ENDDO                                                 Deleted,NoXref!
269           123
270           123        C* Step 7. Compute Poisson Probability==
271           123
272           123        C* SKIP if population count is too small
273           123               IF(OBSP. LT.POPMIN)GOTO 200
274           123        C* SKIP if too small to be of interest
```

```
275           123               IF(OBSC.LT.CANMIN) GOTO 200
276           123               NDAT=NDAT+1
277           123        C* CALCULATE SIGNIFICANCE LEVEL
278           123        C***** Code Expanded From Routine: POIS                      Generated
279           123                                                                     Generated
280           123               JA = OBSC                                             Auto-inline,Xref|
281           123               AMEAN = OBSP                                          Auto-inline,Xref|
282           123        C* initialise                                                Generated
283 P         123        CMIC@ DO ALL GUIDED(64)IF(JA-1.GT.600)SHARED(JA,CONS)PRIVATE(I1X)  Generated
284 P         123        CDIR@ IVDEP                                                  Generated
285 P         123v              DO I1X = 2, JA                                        Auto-inline,Xref|
286 P         123v              CONS(I1X) = 1D0/(I1X - 1D0)                           Auto-inline,Xref|
287 P         123v ---->END DO                                                        Auto-inline,Xref|
288           123                                                                     Generated
289           123        C* calculate Poison Probability of JA cancers being observed Generated
290           123               IF (JA .GT. 1) THEN                                   Auto-inline,Xref|
291           123               CUMPRB(1) = EXP((-AMEAN))                             Auto-inline,Xref|
292           123               PROB = CUMPRB(1)                                      Auto-inline,Xref|
293           123               IF (JA - 1 .GE. 10) THEN                              Auto-inline,Xref|
294           123        CDIR@ IVDEP                                                  Generated
295           123v              DO J = 1, JA - 1                                      Auto-inline,Xref|
296           123v              D1V(J) = AMEAN*CONS(1+J)                              Auto-inline,Xref|
297           123v ---->END DO                                                        Auto-inline,Xref|
298           1234 -----DO J = 1, JA - 1                                              Auto-inline,Xref|
299           1234              CUMPRB(1+J) = D1V(J)*CUMPRB(J)                        Auto-inline,Xref|
300           1234 ---->END DO                                                        Auto-inline,Xref|
301           123        CDIR@ IVDEP                                                  Generated
302           123v -----DO J = 1, JA - 1                                              Auto-inline,Xref|
303           123v              PROB = PROB + CUMPRB(1+J)                             Auto-inline,Xref|
304           123v ---->END DO                                                        Auto-inline,Xref|
305           123               ELSE                                                  Auto-inline,Xref|
306           123        CDIR@ NEXTSCALAR                                             Generated
307           1234 ---   DO J = 2, JA                                                 Auto-inline,Xref|
308           1234              CUMPRB(J) = AMEAN*CONS(J)*CUMPRB(J-1)                 Auto-inline,Xref|
309           1234              PROB = PROB + CUMPRB(J)                               Auto-inline,Xref|
310           1234 ---->END DO                                                        Auto-inline,Xref|
311           123               ENDIF                                                 Auto-inline,Xref|
312           123               PROB = 1.0 - PROB                                     Auto-inline,Xref|
313           123               ELSE                                                  Auto-inline,Xref|
314           123        C* 1 OR less cancers                                         Generated
315           123               PROB = 1.0 - EXP((-AMEAN))                            Auto-inline,Xref|
316           123               ENDIF                                                 Auto-inline,Xref|
317           123        C***** End of Code Expanded From Routine: POIS               Generated
                                CALL POIS(OBSP,OBSC,PROB)                             Deleted,NoXref!
318           123               NHY=NHY+1
319           123               IF(PROB.GT.THRESH)GOTO 200
320           123        C* YES it's significant so save
321           123               NCALS=NCALS+1
322           123
323           123        C* Step 7. Save circle info===========
324           123               WRITE(9,90001) CX,CY,RADIUS,OBSP,OBSC,PROB
325           123        90001  FORMAT(3F9.3,2F9.3,F10.7)
326           123        C* END OF EASTING
327           123->      200    CONTINUE
328           12
329           12         C* END OF NORTHING
330           12-->      100    CONTINUE
331           1
332           1          C
333           1          C End of SEARCH LOOP for given circle radius
334           1          C
335           1
336           1                 WRITE(6,78221)RADIUS,STEP,NCALC,NDAT,NHY,NCALS
337           1          78221  FORMAT(40(1H-)/' *RADIUS=',F12.2,'KM with STEP of',I6 ' KM'/
338           1               X  1H ,5X,'*Number of sites generated ',I10/
339           1               X  1H ,5X,'*Number of sites examined ',I10/
340     1                X  1H ,5X,'*Number of hypotheses tested ',I10/
341     1                X  1H ,5X,'*Number of significant circles',I10)
342     1
343     1           C* form global stats
344     1                 TOTCAL=TOTCAL+NCALC
345     1                 TOTDAT=TOTDAT+NDAT
346     1                 TOTNHY=TOTNHY+NHY
347     1                 TOTNCS=TOTNCS+NCALS
348     1           C* go back and do another circle size
349     1--->             ENDDO
350
351                 C*************************************
352                 C* END OF ALL RUNS********************
353                 C*************************************
354                       WRITE(6,887) TOTCAL,TOTDAT,TOTNHY,TOTNCS
355                 887   FORMAT('0********** End of GAM Run **
356                      X  1H ,'*Total sites generated is',I10/
357                      X  1H ,'*Total sites examined ',I10/
358                      X  1H ,'*Total hypotheses tested ',I10/
359                      X  1H ,'*Total significant circles ',I10)
360
361                       STOP
362                       END

1                   C* ~stan/gam/gam_8.f DSQRT, N(**,-) and 1 IF removed, data filter
2                         IMPLICIT NONE                                               Generated
3                         DOUBLEPRECISION D1V(3000)                                   Generated,Xref
4                         REAL QQQ                                                    Generated,Xref
5                         TASK COMMON/Z1FPP1 CM/ QQQ(8191)                            Generated,Xref
6                         EQUIVALENCE (QQQ(1),D1V)                                    Generated,Xref
7                         INTEGER NCASE
8                         PARAMETER (NCASE=150 000)
9                         REAL OVERAT,XMINE,XMAXN,XMINN,XMAXE,
10                       X   OBSP,OBSC,RADIUS,DIS,CX,CY,PROB,RADSQ,
11                       X   X(NCASE),Y(NCASE),P(NCASE),C(NCASE),
12                       X   RADINC,RADMAX,RADMIN,POPMIN,CANMIN,THRESH,
13                       X   YY(NCASE),XX(NCASE),PP(NCASE),CC(NCASE),
14                       X   LEFT,RIGHT
15
16
17                        CHARACTER*100 XYDATF,PCDATF,OUTFIL
18                        INTEGER I,LOOP,ICOL,IROW,L,K1,K2,INDEX(NCASE)
19                        INTEGER TOTCAL,TOTDAT,TOTNCS,TOTNHY,STEP,ID,N,
20                       X   MINN,MAXN,MINE,MAXE,NTIMES,NCALC,NDAT,NCALS,NHY,
21                       X   N2
22                  C*
23                        INTEGER I1X, INC, LIMIT, J, II, L1X, K, I2X, JA, J1X        Generated,Xref
24                        REAL B, OBSC1X, OBSP1X, XMIN1X, XMIN2X, XMAX1X, XMAX2X,     Generated,Xref
25                       X   OBSP2X,OBSC2X                                            Generated,Xref
26                        DOUBLEPRECISION CUMPRB(3000), CONS(3000), AMEAN, PROB1      Generated,Xref
27                        WRITE(6,78001)
28                  78001 FORMAT('*Geographical Analysis Machine GAM/1 (Feb 1997)'//)
29                  C* Step 1. Read Data============================
30                  C* set constants
31
32                  C* read ini.file
33                        OPEN(UNIT=1,FILE='gamfiles.dat',FORM='FORMATTED',
34                       X   STATUS='OLD')
35
36                  C* read USER DATA file names
37                  C.. this file contains X,Y data
38                        READ(1,10001) XYDATF
39                  C.. this file contains Pop at Risk and Count of Real Cases
40                        READ(1,10001) PCDATF
41                  10001 FORMAT(A)
42                  C* get output results file name
```

```
43                        READ(1,10001) OUTFIL
44                        CLOSE(UNIT=1,STATUS='KEEP')
45
46
47                  C* read X-Y data
48                        WRITE(6,6707) XYDATF
49                  6707  FORMAT('*User Input X,Y File is;',A)
50                        OPEN(UNIT=1,FILE=XYDATF,
51                       X   STATUS='OLD',
52                       X   FORM='FORMATTED')
53
54                        N=0
55          1 ----        DO I=1,NCASE
56          1             READ(1,*,END=999) ID,XX(I),YY(I)
57          1            X (I)=XX(I)
58          1             N=I
59          1 ----        ENDDO
60                  999   WRITE(6,123) N
61                  123   FORMAT(5X,'*EOF at Case Number',I10)
62                        CLOSE(UNIT=1,STATUS='KEEP')
63                  C* No data read?
64                        IF(N.EQ.0) STOP 1
65                  C* sort X values
66                  C***** Code Expanded From Routine: SORT                              Generated
67                        CALL SORT(X,INDEX,N)                                         Deleted,NoXref!
68
69 P--              CMIC@ DO ALL GUIDED(64)IF(N.GT.1800)SHARED(N,INDEX) PRIVATE(I1X)   Generated
70 P                CDIR@ IVDEP                                                        Generated
71 P        v ----        DO I1X = 1, N                                                Generated,Xref
72 P        v             INDEX(I1X) = I1X                                             Generated,Xref
73 P-       v ----        END DO                                                       Generated,Xref
74                        IF (N .EQ. 1) GO TO 77055                                    Generated,Xref
75                        INC = N/2                                                    Generated,Xref
76                  77056 CONTINUE                                                     Generated,Xref
77                        LIMIT = N - INC                                              Generated,Xref
78 P--              CMIC@ DO ALL GUIDED(64)IF(LIMIT.GT.450.AND.(ABS@(INC).GE.LIMIT.OR. Generated
79 P                CMIC@1 INC.EQ.0))SHARED(LIMIT,INC,X,INDEX)PRIVATE(I1X,J,B,II)      Generated
80 P                CDIR@ IVDEP                                                        Generated
81 P        v ----        DO I1X = 1, LIMIT                                            Generated,Xref
82 P        v             IF (X(I1X) .GT. X(INC+I1X)) THEN                             Generated,Xref
83 P        v             B = X(I1X)                                                   Generated,Xref
84 P        v             X(I1X) = X(INC+I1X)                                          Generated,Xref
85 P        v             X(INC+I1X) = B                                               Generated,Xref
86 P        v             II = INDEX(I1X)                                              Generated,Xref
87 P        v             INDEX(I1X) = INDEX(INC+I1X)                                  Generated,Xref
88 P        v             INDEX(INC+I1X) = II                                          Generated,Xref
89 P        v             ENDIF                                                        Generated,Xref
90 P-       v ---         END DO                                                       Generated,Xref
91                        INC = INC*3/4                                                Generated,Xref
92                        IF (INC .GT. 1) GO TO 77056                                  Generated,Xref
93                        L1X = N - 1                                                  Generated,Xref
94                  77058 CONTINUE                                                     Generated,Xref
95                        IF (L1X .LE. 0) GO TO 77055                                  Generated,Xref
96                        K = 0                                                        Generated,Xref
97          1 ----        DO I1X = 1, L1X                                              Generated,Xref
98          1             J = I1X + 1                                                  Generated,Xref
99          1             IF (X(I1X) .LE. X(J)) GO TO 77059                            Generated,Xref
100         1             B = X(I1X)                                                   Generated,Xref
101         1             X(I1X) = X(J)                                                Generated,Xref
102         1             X(J) = B                                                     Generated,Xref
103         1             II = INDEX(I1X)                                              Generated,Xref
104         1             INDEX(I1X) = INDEX(J)                                        Generated,Xref
105         1             INDEX(J) = II                                                Generated,Xref
106         1             K = I1X                                                      Generated,Xref
107         1       77059 CONTINUE                                                     Generated,Xref
108         1 -----       END DO                                                       Generated,Xref
109                       IF (K .LE. 0) GO TO 77055                                    Generated,Xref
```

```
                               L1X = K - 1                                       Generated,Xref
                          GO TO 77058                                            Generated,Xref
                          77055  CONTINUE                                        Generated,Xref
                          C***** End of Code Expanded From Routine: SORT        Generated
                          C* re-order to reflect sort on X
P--                       CMIC@ DO ALL GUIDED(64)IF(N.GT.900)SHARED(N,INDEX,XX,X,YY,Y)   Generated
P                         CMIC@1 PRIVATE(I, ID)                                  Generated
P                         CDIR@ IVDEP                                            Generated
P          v ---               DO I=1,N
P          v                   ID=INDEX(I)
P          v                   X(I)=XX(ID)
P          v                   Y(I)=YY(ID)
P----      v ----              ENDDO
                          C* check sort
           v ----              DO I=2,N
           v                   IF(X(I).LT.X(I-1)) STOP 55
           v ----              ENDDO

                          C* read Population and Observed Cancer data
                               WRITE(6,6708) PCDATF
                          6708 FORMAT('*User Input Data File is;',A)
                               OPEN(UNIT=1,FILE=PCDATF,
                              X   STATUS='OLD',
                              X   FORM='FORMATTED')

                               N2=0
           1 ---               DO I=1,NCASE
           1                   READ(1,*,END=199) ID,XX(I),YY(I)
           1                   N2=I
           1 ---               ENDDO
                          199  WRITE(6,123) N2
                               CLOSE(UNIT=1,STATUS='KEEP')
                          C*No data read?
                               IF(N2.EQ.0) STOP 2
                          C* re-order to reflect sort on X
P--                       CMIC@ DO ALL GUIDED(64)IF(N.GT.900)SHARED(N,INDEX,XX,C,YY,P)   Generated
P                         CMIC@1 PRIVATE(I, ID)                                  Generated
P                         CDIR@ IVDEP                                            Generated
P          v ----              DO I=1,N
P          v                   ID=INDEX(I)
P          v                   C(I)=XX(ID)
P          v                   P(I)=YY(ID)
P-         v ---               ENDDO
                          C* Files do not match
                               IF(N.NE.N2) STOP 3
                          C* go thru data and produce counts
                               OBSP=0.0
                               OBSC=0.0
PR                        CMIC@ PARALLEL IF (N.GT.1200)SHARED(C,P, N,OBSC,OBSP) PRIVATE(I,   Generated
PR                        CMIC@1 OBSC1X, OBSP1X)                                 Generated
PR                             OBSC1X = 0                                        Generated,Xref
PR                             OBSP1X = 0                                        Generated,Xref
PR P--                    CMIC@ DO PARALLEL GUIDED(64)                           Generated
PR P                      CDIR@ IVDEP                                            Generated
PR P       v ---               DO I = 1, N                                       Generated,Xref
PR P       v                   OBSC1X = OBSC1X + ABS@(C(I))                      Generated,Xref
PR P       v                   OBSP1X = OBSP1X + P(I)                            Generated,Xref
PR P-      v --->              END DO                                            Generated,Xref
CritReg                        CMIC@ GUARD                                       Generated
CritReg                        OBSP = OBSP + OBSP1X                              Generated,Xref
CritReg                        OBSC = OBSC + OBSC1X                              Generated,Xref
CritReg                        CMIC@ END GUARD                                   Generated
PR                        CMIC@ END DO                                           Generated
PR                        CMIC@ END PARALLEL                                     Generated
                               DO I=1,N                                          Deleted,NoXref!
                               OBSC=OBSC+ABS(C(I))                               Deleted,NoXref!
```

```
                               OBSP=OBSP+P(I)                                    Deleted,NoXref!
                               ENDDO                                             Deleted,NoXref!
                               WRITE(6,8) N,OBSP,OBSC
                          8    FORMAT(
                              X   ' *Number of input data records: ',I8/
                              X   ' *Total population at risk: ',F10.0/
                              X   ' *Total Cases ',F10.0)
                               IF(OBSP. EQ.0.0.OR.OBSC.EQ.0.0)STOP 3
                               OVERAT=OBSC/OBSP

                          C* Find Min and Max X,Y values to define search region
                               XMINE=999999999.0
                               XMINN=999999999.0
                               XMAXE=0.0
                               XMAXN=0.0

                          C* data are in 1 km units
PR                        CMIC@ PARALLEL IF(N.GT.900)SHARED(X,Y,N,XMINE,XMINN,XMAXE,     Generated
PR                        CMIC@1 XMAXN) PRIVATE(I, XMIN1X, XMIN2X, XMAX1X, XMAX2X)      Generated
                               DO I=1,N                                          Deleted,NoXref!
                               XMINE=AMIN1(XMINE,X(I))                           Deleted,NoXref!
                               XMINN=AMIN1(XMINN,Y(I))                           Deleted,NoXref!
                               XMAXE=AMAX1(XMAXE,X(I))                           Deleted,NoXref!
                               XMAXN=AMAX1(XMAXN,Y(I))                           Deleted,NoXref!
                               ENDDO                                             Deleted,NoXref!
PR
PR                        C* data are in 1 km units                              Generated
PR                             XMIN1X = XMINE                                    Generated,Xref
PR                             XMIN2X = XMINN                                    Generated,Xref
PR                             XMAX1X = XMAXE                                    Generated,Xref
PR                             XMAX2X = XMAXN                                    Generated,Xref
PR P--                    CMIC@ DO PARALLEL GUIDED(64)                           Generated
PR P                      CDIR@ IVDEP                                            Generated
PR P       v ----              DO I = 1, N                                       Generated,Xref
PR P       v                   XMIN1X = AMIN1(XMIN1X,X(I))                       Generated,Xref
PR P       v                   XMIN2X = AMIN1(XMIN2X,Y(I))                       Generated,Xref
PR P       v                   XMAX1X = AMAX1(XMAX1X,X(I))                       Generated,Xref
PR P       v                   XMAX2X = AMAX1(XMAX2X,Y(I))                       Generated,Xref
PR P--     v ---               END DO                                            Generated
CritReg                        CMIC@ GUARD                                       Generated,Xref
CritReg                        XMAXN = AMAX1(XMAXN,XMAX2X)                       Generated,Xref
CritReg                        XMAXE = AMAX1(XMAXE,XMAX1X)                       Generated,Xref
CritReg                        XMINN = AMIN1(XMINN,XMIN2X)                       Generated,Xref
CritReg                        XMINE = AMIN1(XMINE,XMIN1X)                       Generated
CritReg                        CMIC@ END GUARD                                   Generated
PR                        CMIC@ END DO                                           Generated
PR                        CMIC@ END PARALLEL                                     Generated
                                                                                 Generated
                               WRITE(6,7123) OVERAT,XMINE,XMAXE,XMINN,XMAXN
                          7123 FORMAT(
                              X   ' *Global Incidence Rate per Population at Risk is '
                              X   ,F15.9/
                              X   ' *Minimum Easting is',F12.1,' Maximum is',F12.1/
                              X   ' *Minimum Northing is',F12.1,' Maximum is',F12.1)
                               MINN=XMINN-1.0
                               MINE=XMINE-1.0
                               MAXN=XMAXN+1.0
                               MAXE=XMAXE+1.0

                          C* Step 2. Set Search Parameters===========================
                          C* Circle Radii are in KM
                               RADMIN=10.0
                               RADMAX=10.0
                               RADINC=1.0

                          C* select probability threshold
```

```
                               THRESH=0.005

                         C* set minimum circle size
                               POPMIN=100.0
                         C* set minimum cancer count size
                               CANMIN=2.0

                         C* write search parameters out
                               WRITE(6,76541) RADMIN,RADMAX,RADINC,POPMIN
                         76541 FORMAT('*Minimum Circle radius is',F10.3,' 1 km'/
                             X   '*Maximum Circle radius is',F10.3,' 1 km'/
                             X   '*Circle increment set to',F10.3,' 1 km'/
                             X   '*Minimum POPULATION size is',F10.0)

                               WRITE(6,78234) THRESH
                         78234 FORMAT(' *Significance THRESHOLD set at',F12.6)

                         C* other global inits
                               TOTCAL=0
                               TOTDAT=0
                               TOTNCS=0
                               TOTNHY=0

                         C* convert all population counts into expected values
P--                      CMIC@ DO ALL GUIDED(64)IF(N.GT.1800)SHARED(N,OVERAT,P)PRIVATE(I)      Generated
P                        CDIR@ IVDEP                                                           Generated
P         v ----               DO I=1,N
P         v                    P(I)=P(I)*OVERAT
P-        v ----               ENDDO

                         C* reset minimum value
                               POPMIN=POPMIN*OVERAT

                         C* SET INITIAL RADIUS for circles
                               RADIUS=RADMIN-RADINC
                         C* compute number of circle sizes to be examined
                               NTIMES=(RADMAX-RADMIN)/RADINC+1.0

                         C* open output file
                               OPEN(UNIT=9,FILE=OUTFIL,STATUS='UNKNOWN',
                             X   FORM='FORMATTED')
                         C* Step 3. Circle Size Loop==================================
                         C* *********Circle SIZE loop starts here ******************
          1 -----              DO LOOP=1,NTIMES
          1              C* set circle radius
          1                    RADIUS=RADIUS+RADINC
          1                    RADSQ=RADIUS*RADIUS
          1              CC STEP=RADIUS*0.2+0.5001
          1                    STEP=RADIUS
          1                    IF(STEP. EQ.0) STEP=1
          1
          1                    NCALC=0
          1                    NDAT=0
          1                    NCALS=0
          1                    NHY=0
          1
          1              C* Step 4. Grid Search: Northing Loop=======================
          1
          12 ---               DO 100 IROW = MINN, MAXN, STEP
          12                   CY=IROW
          12
          12                   L=0
          12v ---              DO I=1,N
          12v                  DIS=(Y(I)-CY)**2
          12v                  IF(DIS.LE.RADSQ)THEN
          12v                  L=L+1
```

```
          12v                  YY(L)=DIS
          12v                  XX(L)=X(I)
          12v                  PP(L)=P(I)
          12v                  CC(L)=C(I)
          12v                  ENDIF
          12v -->              ENDDO
          12                   IF(L.EQ.0)GOTO 100
          12
          12             C* Step 5. Grid Search: Easting Loop========================
          123 -                DO 200 ICOL = MINE, MAXE, STEP
          123                  CX=ICOL
          123
          123            C* GET DATA WITHIN CIRCLE AT (IROW, ICOL)
          123                  NCALC=NCALC+1
          123
          123            C* establish search region
          123                  LEFT=CX-RADIUS
          123                  RIGHT=CX+RADIUS
          123                  CALL FIND(XX,L,LEFT,K1,1)
          123                  CALL FIND(XX,L,RIGHT,K2,2)
          123            C* Step 6. Get Data for Circle==============================
          123                  OBSP=0.0
          123                  OBSC=0.0
PR                         123   CMIC@ PARALLEL IF(K2-K1+1.GT.720)SHARED(K2,K1,CX,RADSQ,OBSP   Generated
PR        123            CMIC@1 , OBSC, XX, YY, PP, CC) PRIVATE(I, DIS, OBSP2X, OBS C2X)       Generated
PR        123                  OBSP2X = 0                                                      Generated,Xref
PR        123                  OBSC2X = 0                                                      Generated,Xref
PR P--    123            CMIC@ DO PARALLEL GUIDED(64)                                          Generated
PR P      123            CDIR@ IVDEP                                                           Generated
PR P      123v -               DO I = 1, K2 - K1 + 1                                           Generated,Xref
                               DO I=K1,K2                                                      Deleted,NoXref!
PR P      123v           C* calc distance of ED at X(I), Y(I) from grid point CX,CY
PR P      123v                 DIS = (XX(K1+I-1)-CX)**2 + YY(K1+I-1)                           Generated,Xref
                               DIS=(XX(I)-CX)**2+YY(I)                                         Deleted,NoXref!
PR P      123v           C* is point inside circle?
PR P      123v                 IF (DIS .LE. RADSQ) THEN                                        Generated,Xref
                               IF(DIS.LE.RADSQ)THEN                                            Deleted,NoXref!
PR P      123v           C* yes so accumulate counts
PR P      123v                 OBSP2X = OBSP2X + PP(K1+I-1)                                    Generated,Xref
PR P      123v                 OBSC2X = OBSC2X + CC(K1+I-1)                                    Generated,Xref
                               OBSP=OBSP+PP(I)                                                 Deleted,NoXref!
                               OBSC=OBSC+CC(I)                                                 Deleted,NoXref!
PR P      123v                 ENDIF
PR P--    123v ->              END DO                                                          Generated,Xref
CritReg   123                  CMIC@ GUARD                                                     Generated
CritReg   123                  OBSC = OBSC + OBSC2X                                            Generated,Xref
CritReg   123                  OBSP = OBSP + OBSP2X                                            Generated,Xref
CritReg   123            CMIC@ END GUARD                                                       Generated
PR        123            CMIC@ END DO                                                          Generated
PR        123            CMIC@ END PARALLEL                                                    Generated
                               ENDDO                                                           Deleted,NoXref!
          123
          123            C* Step 6. Compute Poisson Probability=====================
          123
          123            C* SKIP if population count is too small
          123                  IF(OBSP. LT.POPMIN)GOTO 200
          123            C* SKIP if too small to be of interest
          123                  IF(OBSC.LT.CANMIN) GOTO 200
          123                  NDAT=NDAT+1
          123            C* CALCULATE SIGNIFICANCE LEVEL
          123            C***** Code Expanded From Routine: POIS                               Generated        \
          123                                                                                  Generated
          123                  JA = OBSC                                                       Auto-inline,Xref|
          123                  AMEAN = OBSP                                                    Auto-inline,Xref|
          123            C* initialise                                                         Generated |
P--       123            CMIC@ DO ALL GUIDED(64)IF(JA-1.GT.600)SHARED(JA,CONS)PRIVATE(I2X)     Generated |
P       123         CDIR@ IVDEP                                                  Generated |
P       123v -            DO I2X = 2, JA                                         Auto-inline,Xref|
P       123v              CONS(I2X) = 1D0/(I2X - 1D0)                            Auto-inline,Xref|
P-      123v ->           END DO                                                 Auto-inline,Xref|
        123         C* calculate Poison Probability of JA cancers being observed Generated
        123               IF (JA .GT. 1) THEN                                    Auto-inline,Xref|
        123               CUMPRB(1) = EXP((-AMEAN))                              Auto-inline,Xref|
        123               PROB1X = CUMPRB(1)                                     Auto-inline,Xref|
        123               IF (JA - 1 .GE. 10) THEN                               Auto-inline,Xref|
        123         CDIR@ IVDEP                                                  Generated
        123v ---          DO J1X = 1, JA - 1                                     Auto-inline,Xref|
        123v              D1V(J1X) = AMEAN*CONS(1+J1X)                           Auto-inline,Xref|
        123v -->          END DO                                                 Auto-inline,Xref|
        1234 --           DO J1X = 1, JA - 1                                     Auto-inline,Xref|
        1234              CUMPRB(1+J1X) = D1V(J1X)*CUMPRB(J1X)                   Auto-inline,Xref|
        1234 -->          END DO                                                 Auto-inline,Xref|
        123         CDIR@ IVDEP                                                  Generated |
        123v -----        DO J1X = 1, JA - 1                                     Auto-inline,Xref|
        123v              PROB1X = PROB1X + CUMPRB(1+J1X)                        Auto-inline,Xref|
        123v -->          END DO                                                 Auto-inline,Xref|
        123               ELSE                                                   Auto-inline,Xref|
        123         CDIR@ NEXTSCALAR                                             Generated |
        1234 --           DO J1X = 2, JA                                         Auto-inline,Xref|
        1234              CUMPRB(J1X) = AMEAN*CONS(J1X)*CUMPRB(J1X-1)            Auto-inline,Xref|
        1234              PROB1X = PROB1X + CUMPRB(J1X)                          Auto-inline,Xref|
        1234 -->          END DO                                                 Auto-inline,Xref|
        123               ENDIF                                                  Auto-inline,Xref|
        123               PROB1X = 1.0 - PROB1X                                  Auto-inline,Xref|
        123               ELSE                                                   Auto-inline,Xref|
        123         C* 1 OR less cancers                                         Generated
        123               PROB1X = 1.0 - EXP((-AMEAN))                           Auto-inline,Xref|
        123               ENDIF                                                  Auto-inline,Xref|
        123               PROB = PROB1X                                          Auto-inline,Xref|
        123         C***** End of Code Expanded From Routine:  POIS              Generated
                          CALL POIS(OBSP,OBSC,PROB)                              Deleted,NoXref!
        123               NHY=NHY+1
        123               IF(PROB.GT.THRESH)GOTO 200
        123         C* YES it's significant so save
        123               NCALS=NCALS+1
        123
        123         C* Step 7. Save circle info=================================
        123               WRITE(9,90001) CX,CY,RADIUS,OBSP,OBSC,PROB
        123         90001 FORMAT(3F9.3,2F9.3,F10.7)
        123         C* END OF EASTING
        123 --->    200   CONTINUE
        12
        12          C* END OF NORTHING
        12 ---->     100  CONTINUE
        1
        1           C
        1           C ************ End of SEARCH LOOP for given circle radius *
        1           C
        1
        1                 WRITE(6,78221)RADIUS,STEP,NCALC,NDAT,NHY,NCALS
        1           78221 FORMAT(40(1H-)/' *RADIUS=',F12.2,'KM with STEP of',I6 ' KM'/
        1                X 1H ,5X,'*Number of sites generated ',I10/
        1                X 1H ,5X,'*Number of sites examined ',I10/
        1                X 1H ,5X,'*Number of hypotheses tested ',I10/
        1                X 1H ,5X,'*Number of significant circles',I10)
        1
        1           C* form global stats
        1                 TOTCAL=TOTCAL+NCALC
        1                 TOTDAT=TOTDAT+NDAT
        1                 TOTNHY=TOTNHY+NHY
        1                 TOTNCS=TOTNCS+NCALS
        1           C* go back and do another circle size
```

```
        1 ------->        ENDDO

              C************************************************************
              C* END OF ALL RUNS*******************************************
              C************************************************************
                    WRITE(6,887) TOTCAL,TOTDAT,TOTNHY,TOTNCS
              887   FORMAT('0********** End of GAM Run ****************'/
                  X  1H ,'*Total sites generated is',I10/
                  X  1H ,'*Total sites examined ',I10/
                  X  1H ,'*Total hypotheses tested ',I10/
                  X  1H ,'*Total significant circles ',I10)

                    STOP
                    END
```

Appendix 6.2: multi-tasking code of GAM Version 8

```
C*  ~stan/gam/gam_8.f DSQRT, N(**,-) and 1 IF removed, data filter
      IMPLICIT NONE Generated
      DOUBLEPRECISION D1V(3000)                                    Generated,Xref
      REAL QQQ                                                     Generated,Xref
      TASK COMMON/Z1FPP1 cm/ QQQ(8191)                             Generated,Xref
      EQUIVALENCE (QQQ(1),D1V)                                     Generated,Xref
      INTEGER NCASE
      PARAMETER (NCASE=150 000)
      REAL OVERAT,XMINE,XMAXN,XMINN,XMAXE,
     X OBSP,OBSC,RADIUS,DIS,CX,CY,PROB,RADSQ,
     X X(NCASE),Y(NCASE),P(NCASE),C(NCASE),
     X RADINC,RADMAX,RADMIN,POPMIN,CANMIN,THRESH,
     X YY(NCASE),XX(NCASE),PP(NCASE),CC(NCASE),
     X LEFT,RIGHT


      CHARACTER*100 XYDATF,PCDATF,OUTFIL
      INTEGER I,LOOP,ICOL,IROW,L,K1,K2,INDEX(NCASE)
      INTEGER TOTCAL,TOTDAT,TOTNCS,TOTNHY,STEP,ID,N,
     X MINN,MAXN,MINE,MAXE,NTIMES,NCALC,NDAT,NCALS,NHY,
     X N2
 C*
      INTEGER I1X, INC, LIMIT, J, II, L1X, K, I2X, JA, J1X          Generated,Xref
      REAL B, OBSC1X, OBSP1X, XMIN1X, XMIN2X, XMAX1X, XMAX2X,OBSP2X,OB  Generated,Xref
      .SC2X Generated,Xref
      DOUBLEPRECISION CUMPRB(3000), CONS(3000), AMEAN, PROB1        Generated,Xref
      WRITE(6,78001)
      78001 FORMAT('*Geographical Analysis Machine GAM/1 (Feb 1997)'//)
C* Step 1. Read Data===========================
C* set constants

C* read ini.file
      OPEN(UNIT=1,FILE='gamfiles.dat',FORM='FORMATTED',
     X STATUS='OLD')

C* read USER DATA file names
C.. this file contains X,Y data
      READ(1,10001) XYDATF
C.. this file contains Pop at Risk and Count of Real Cases
      READ(1,10001) PCDATF
      10001 FORMAT(A)
C* get output results file name
      READ(1,10001) OUTFIL
      CLOSE(UNIT=1,STATUS='KEEP')


C* read X-Y data
      WRITE(6,6707) XYDATF
      6707 FORMAT('*User Input X,Y File is;',A)
      OPEN(UNIT=1,FILE=XYDATF,
     X STATUS='OLD',
     X FORM='FORMATTED')

      N=0
1 --- DO I=1,NCASE
```

```
56 1             READ(1,*,END=999) ID,XX(I),YY(I)
57 1             X(I)=XX(I)
58 1             N=I
59 1 --> ENDDO
60 999   WRITE(6,123) N
61 123 FORMAT(5X,'*EOF at Case Number',I10)
62       CLOSE(UNIT=1,STATUS='KEEP')
63 C* No data read?
64       IF(N.EQ.0) STOP 1
65 C* sort X values
66 C***** Code Expanded From Routine: SORT                                   Generated \
67       CALL SORT(X,INDEX,N)                                                Deleted,NoXref!
68
69 P-- CMIC@ DO ALL GUIDED(64) IF (N .GT. 1800) SHARED(N, INDEX) PRIVATE(I1X)     Generated
70 P CDIR@ IVDEP                                                             Generated
71 P v ---- DO I1X = 1, N                                                    Generated,Xref
72 P v          INDEX(I1X) = I1X                                             Generated,Xref
73 P v ----> END DO                                                          Generated,Xref
74       IF (N .EQ. 1) GO TO 77055                                           Generated,Xref
75       INC = N/2                                                           Generated,Xref
76 77056 CONTINUE                                                            Generated,Xref
77       LIMIT = N - INC                                                     Generated,Xref
78 P-- CMIC@ DO ALL GUIDED(64) IF (LIMIT.GT.450 .AND. (ABS@(INC).GE .LIMIT .OR.   Generated
79 P CMIC@1 INC.EQ.0)) SHARED(LIMIT, INC, X, INDEX) PRIVATE(I1X , J, B, II)  Generated
80 P CDIR@ IVDEP                                                             Generated
81 P v --- DO I1X = 1, LIMIT                                                 Generated,Xref
82 P v             IF (X(I1X) .GT. X(INC+I1X)) THEN                          Generated,Xref
83 P v             B = X(I1X)                                                Generated,Xref
84 P v             X(I1X) = X(INC+I1X)                                       Generated,Xref
85 P v             X(INC+I1X) = B                                            Generated,Xref
86 P v             II = INDEX(I1X)                                           Generated,Xref
87 P v             INDEX(I1X) = INDEX(INC+I1X)                               Generated,Xref
88 P v             INDEX(INC+I1X) = II                                       Generated,Xref
89 P v             ENDIF                                                     Generated,Xref
90 P-- v -------> END DO                                                     Generated,Xref
91       INC = INC*3/4                                                       Generated,Xref
92       IF (INC .GT. 1) GO TO 77056                                         Generated,Xref
93       L1X = N - 1                                                         Generated,Xref
94 77058          CONTINUE                                                   Generated,Xref
95       IF (L1X .LE. 0) GO TO 77055                                         Generated,Xref
96       K = 0                                                               Generated,Xref
97 1 --- DO I1X = 1, L1X                                                     Generated,Xref
98 1     J = I1X + 1                                                         Generated,Xref
99 1     IF (X(I1X) .LE. X(J)) GO TO 77059                                   Generated,Xref
100 1    B = X(I1X)                                                          Generated,Xref
101 1    X(I1X) = X(J)                                                       Generated,Xref
102 1    X(J) = B                                                            Generated,Xref
103 1    II = INDEX(I1X)                                                     Generated,Xref
104 1    INDEX(I1X) = INDEX(J)                                               Generated,Xref
105 1    INDEX(J) = II                                                       Generated,Xref
106 1    K = I1X                                                             Generated,Xref
107 177059 CONTINUE                                                          Generated,Xref
108 1 --> END DO                                                             Generated,Xref
109      IF (K .LE. 0) GO TO 77055                                           Generated,Xref
110      L1X = K - 1                                                         Generated,Xref
111      GO TO 77058                                                         Generated,Xref
112      77055 CONTINUE                                                      Generated,Xref
113 C***** End of Code Expanded From Routine: SORT                           Generated
114 C* re-order to reflect sort on X
115 P-- CMIC@ DO ALL GUIDED(64) IF (N .GT. 900) SHARED(N, INDEX, XX, X, YY, Y)   Generated
116 P CMIC@1 PRIVATE(I, ID)                                                  Generated
117 P CDIR@ IVDEP                                                            Generated
118 P v - DO I=1,N
119 P v   ID=INDEX(I)
120 P v   X(I)=XX(ID)
121 P v   Y(I)=YY(ID)
122 P v - ENDDO
123 C* check sort
124 v --  DO I=2,N
125 v     IF(X(I).LT.X(I-1)) STOP 55
126 v --  ENDDO
127
128 C* read Population and Observed Cancer data
129      WRITE(6,6708) PCDATF
130      6708 FORMAT('*User Input Data File is;',A)
131      OPEN(UNIT=1,FILE=PCDATF,
```

```
132      X STATUS='OLD',
133      X FORM='FORMATTED')
134
135        N2=0
136 1 --   DO I=1,NCASE
137 1      READ(1,*,END=199) ID,XX(I),YY(I)
138 1      N2=I
139 1 --  ENDDO
140 199    WRITE(6,123) N2
141        CLOSE(UNIT=1,STATUS='KEEP')
142 C*No data read?
143        IF(N2.EQ.0) STOP 2
144 C* re-order to reflect sort on X
145 P-- CMIC@ DO ALL GUIDED(64) IF (N .GT. 900) SHARED(N, INDEX, XX, C, YY, P)   Generated
146 P CMIC@1 PRIVATE(I, ID)                                                  Generated
147 P CDIR@ IVDEP                                                            Generated
148 P v -           DO I=1,N
149 P v   ID=INDEX(I)
150 P v   C(I)=XX(ID)
151 P v   P(I)=YY(ID)
152 Pv -  ENDDO
153 C* Files do not match
154        IF(N.NE.N2) STOP 3
155        C* go thru data and produce counts
156        OBSP=0.0
157        OBSC=0.0
158PR - CMIC@ PARALLEL IF (N .GT. 1200) SHARED(C, P, N, OBSC, OBSP) PRIVATE(I,   Generated
159PR CMIC@1 OBSC1X, OBSP1X)                                                Generated
160PR      OBSC1X = 0                                                      Generated,Xref
161PR      OBSP1X = 0                                                      Generated,Xref
162PR P-- CMIC@ DO PARALLEL GUIDED(64)                                     Generated
163PR P CDIR@ IVDEP                                                       Generated
164PR P v- DO I = 1, N                                                    Generated,Xref
165PR P v          OBSC1X = OBSC1X + ABS@(C(I))                           Generated,Xref
166PR P v OBSP1X = OBSP1X + P(I)                                          Generated,Xref
167PR Pv -ENDDO                                                           Generated,Xref
168CritReg CMIC@ GUARD                                                    Generated
169CritReg OBSP = OBSP + OBSP1X                                           Generated,Xref
170CritReg OBSC = OBSC + OBSC1X                                           Generated,Xref
171CritReg CMIC@ END GUARD                                                Generated
172PR CMIC@ ENDDO                                                         Generated
173PRegion- CMIC@ END PARALLEL                                            Generated
174        DO I=1,N                                                       Deleted,NoXref!
175        OBSC=OBSC+ABS(C(I))                                            Deleted,NoXref!
176        OBSP=OBSP+P(I)                                                 Deleted,NoXref!
177        ENDDO                                                          Deleted,NoXref!
178        WRITE(6,8) N,OBSP,OBSC
179        8 FORMAT(
180       X ' *Number of input data records: ',I8/
181       X ' *Total population at risk: ',F10.0/
182       X ' *Total Cases ',F10.0)
183        IF(OBSP. EQ.0.0.OR.OBSC.EQ.0.0)STOP 3
184        OVERAT=OBSC/OBSP
185
186 C* Find Min and Max X,Y values to define search region
187        XMINE=999999999.0
188        XMINN=999999999.0
189        XMAXE=0.0
190        XMAXN=0.0
191
192 C* data are in 1 km units
193PR - CMIC@ PARALLEL IF (N .GT. 900) SHARED(X, Y, N, XMINE, XMINN, XMAXE,   Generated
194PR CMIC@1 XMAXN) PRIVATE(I, XMIN1X, XMIN2X, XMAX1X, XMAX2X)              Generated
195        DO I=1,N                                                       Deleted,NoXref!
196        XMINE=AMIN1(XMINE,X(I))                                        Deleted,NoXref!
197        XMINN=AMIN1(XMINN,Y(I))                                        Deleted,NoXref!
198        XMAXE=AMAX1(XMAXE,X(I))                                        Deleted,NoXref!
199        XMAXN=AMAX1(XMAXN,Y(I))                                        Deleted,NoXref!
200        ENDDO                                                          Deleted,NoXref!
201PR
202PR C* data are in 1 km units                                            Generated
203PR      XMIN1X = XMINE                                                  Generated,Xref
204PR      XMIN2X = XMINN                                                  Generated,Xref
205PR      XMAX1X = XMAXE                                                  Generated,Xref
206PR      XMAX2X = XMAXN                                                  Generated,Xref
207PR P-- CMIC@ DO PARALLEL GUIDED(64)                                     Generated
```

```
208PR P CDIR@ IVDEP                                                 Generated
209PR P v-DO I = 1, N                                               Generated,Xref
210PR P v             XMIN1X = AMIN1(XMIN1X,X(I))                   Generated,Xref
211PR P v             XMIN2X = AMIN1(XMIN2X,Y(I))                   Generated,Xref
212PR P v             XMAX1X = AMAX1(XMAX1X,X(I))                   Generated,Xref
213PR P v             XMAX2X = AMAX1(XMAX2X,Y(I))                   Generated,Xref
214PR P v-ENDDO                                                     Generated,Xref
215CritReg CMIC@ GUARD                                              Generated
216CritReg XMAXN = AMAX1(XMAXN,XMAX2X)                              Generated,Xref
217CritReg XMAXE = AMAX1(XMAXE,XMAX1X)                              Generated,Xref
218CritReg XMINN = AMIN1(XMINN,XMIN2X)                              Generated,Xref
219CritReg XMINE = AMIN1(XMINE,XMIN1X)                              Generated,Xref
220CritReg CMIC@ END GUARD                                          Generated
221PR CMIC@ ENDDO                                                   Generated
222PR- CMIC@ END PARALLEL                                           Generated
223                                                                 Generated
224        WRITE(6,7123) OVERAT,XMINE,XMAXE,XMINN,XMAXN
225        7123 FORMAT(
226       X ' *Global Incidence Rate per Population at Risk is '
227        ,F15.9/
228       X' *Minimum Easting is',F12.1,' Maximum is',F12.1/
229       X' *Minimum Northing is',F12.1,' Maximum is',F12.1)
230        MINN=XMINN-1.0
231        MINE=XMINE-1.0
232        MAXN=XMAXN+1.0
233        MAXE=XMAXE+1.0
234
235 C* Step 2. Set Search Parameters============================
236 C* Circle Radii are in KM
237        RADMIN=10.0
238        RADMAX=10.0
239        RADINC=1.0
240
241 C* select probability threshold
242        THRESH=0.005
243
244 C* set minimum circle size
245        POPMIN=100.0
246 C* set minimum cancer count size
247        CANMIN=2.0
248
249 C* write search parameters out
250        WRITE(6,76541) RADMIN,RADMAX,RADINC,POPMIN
251        76541 FORMAT('*Minimum Circle radius is',F10.3,' 1 km'/
252       X '*Maximum Circle radius is',F10.3,' 1 km'/
253       X '*Circle increment set to',F10.3,' 1 km'/
254       X '*Minimum POPULATION size is',F10.0)
255
256        WRITE(6,78234) THRESH
257        78234 FORMAT(' *Significance THRESHOLD set at',F12.6)
258
259 C* other global inits
260        TOTCAL=0
261        TOTDAT=0
262        TOTNCS=0
263        TOTNHY=0
264
265 C* convert all population counts into expected values
266 P-- CMIC@ DO ALL GUIDED(64) IF (N .GT. 1800) SHARED(N, OVERAT, P) PRIVATE(I)    Generated
267 P CDIR@ IVDEP                                                                   Generated
268 P v-   DO I=1,N
269 P v    P(I)=P(I)*OVERAT
270 Pv -   ENDDO
271
272 C* reset minimum value
273        POPMIN=POPMIN*OVERAT
274
275 C* SET INITIAL RADIUS for circles
276        RADIUS=RADMIN-RADINC
277 C* compute number of circle sizes to be examined
278        NTIMES=(RADMAX-RADMIN)/RADINC+1.0
279
280 C* open output file
281        OPEN(UNIT=9,FILE=OUTFIL,STATUS='UNKNOWN',
282       X FORM='FORMATTED')
283 C* Step 3. Circle Size Loop==================================
```

```
284 C* *********Circle SIZE loop starts here *******************
285 1 -- DO LOOP=1,NTIMES
286 1C* set circle radius
287 1      RADIUS=RADIUS+RADINC
288 1      RADSQ=RADIUS*RADIUS
289 1 CC STEP=RADIUS*0.2+0.5001
290 1      STEP=RADIUS
291 1      IF(STEP. EQ.0) STEP=1
292 1
293 1      NCALC=0
294 1      NDAT=0
295 1      NCALS=0
296 1      NHY=0
297 1
298 1 C* Step 4. Grid Search: Northing Loop=======================
299 1
300 12 --            DO 100 IROW = MINN, MAXN, STEP
301 12     CY=IROW
302 12
303 12     L=0
304 12v  --DO I=1,N
305 12v    DIS=(Y(I)-CY)**2
306 12v    IF(DIS.LE.RADSQ)THEN
307 12v    L=L+1
308 12v    YY(L)=DIS
309 12v    XX(L)=X(I)
310 12v    PP(L)=P(I)
311 12v    CC(L)=C(I)
312 12v    ENDIF
313 12v -            ENDDO
314 12     IF(L.EQ.0)GOTO 100
315 12
316 12 C* Step 5. Grid Search: Easting Loop========================
317 123 -- DO 200 ICOL = MINE, MAXE, STEP
318 123    CX=ICOL
319 123
320 123 C* GET DATA WITHIN CIRCLE AT (IROW, ICOL)
321 123    NCALC=NCALC+1
322 123
323 123 C* establish search region
324 123    LEFT=CX-RADIUS
325 123    RIGHT=CX+RADIUS
326 123    CALL FIND(XX,L,LEFT,K1,1)
327 123    CALL FIND(XX,L,RIGHT,K2,2)
328 123 C* Step 6. Get Data for Circle==============================
329 123    OBSP=0.0
330 123    OBSC=0.0
331PR- 123 CMIC@ PARALLEL IF (K2 - K1 + 1 .GT. 720) SHARED(K2, K1, CX, RADSQ, OBSP     Generated
332PR 123 CMIC@1 , OBSC, XX, YY, PP, CC) PRIVATE(I, DIS, OBSP2X, OBS C2X)            Generated
333PR 123            OBSP2X = 0                                                     Generated,Xref
334PR 123            OBSC2X = 0                                                     Generated,Xref
335PR P 123 CMIC@ DO PARALLEL GUIDED(64)                                            Generated
336PR P 123 CDIR@ IVDEP                                                             Generated
337PR P 123v ---- DO I = 1, K2 - K1 + 1                                             Generated,Xref
338                DO I=K1,K2                                                      Deleted,NoXref!
339PR P 123v C* calc distance of ED at X(I), Y(I) from grid point CX,CY
340PR P 123v DIS = (XX(K1+I-1)-CX)**2 + YY(K1+I-1)                                  Generated,Xref
341               DIS=(XX(I)-CX)**2+YY(I)                                          Deleted,NoXref!
342PR P 123v C* is point inside circle?
343PR P 123v IF (DIS .LE. RADSQ) THEN                                               Generated,Xref
344               IF(DIS.LE.RADSQ)THEN                                             Deleted,NoXref!
345PR P 123v C* yes so accumulate counts
346PR P 123v       OBSP2X = OBSP2X + PP(K1+I-1)                                     Generated,Xref
347PR P 123v       OBSC2X = OBSC2X + CC(K1+I-1)                                     Generated,Xref
348               OBSP=OBSP+PP(I)                                                  Deleted,NoXref!
349               OBSC=OBSC+CC(I)                                                  Deleted,NoXref!
350PR P 123v       ENDIF
351PR P-- 123v --->END DO                                                           Generated,Xref
352CritReg 123 CMIC@ GUARD                                                          Generated
353CritReg 123    OBSC = OBSC + OBSC2X                                             Generated,Xref
354CritReg 123    OBSP = OBSP + OBSP2X                                             Generated,Xref
355CritReg 123 CMIC@ END GUARD                                                      Generated
356PR 123 CMIC@ ENDDO                                                              Generated
357PR- 123 CMIC@ END PARALLEL                                                      Generated
358        ENDDO                                                                   Deleted,NoXref!
359 123
```

```
360 123 C* Step 7. Compute Poisson Probability=====================
361 123
362 123 C* SKIP if population count is too small
363 123   IF(OBSP. LT.POPMIN)GOTO 200
364 123 C* SKIP if too small to be of interest
365 123   IF(OBSC.LT.CANMIN) GOTO 200
366 123   NDAT=NDAT+1
367 123 C* CALCULATE SIGNIFICANCE LEVEL
368 123 C***** Code Expanded From Routine: POIS                                  Generated \
369 123                                                                         Generated |
370 123   JA = OBSC                                                             Auto-inline,Xref |
371 123   AMEAN = OBSP                                                          Auto-inline,Xref |
372 123 C* initialise                                                           Generated |
373 P-- 123 CMIC@ DO ALL GUIDED(64)IF(JA-1.GT.600)SHARED(JA,CONS)PRIVATE(I2X)   Generated
                            | P 123 CDIR@ IVDEP                                 Generated |
374 P 123v -- DO I2X = 2, JA Auto-inline,Xref | P 123v CONS(I2X) = 1D0/(I2X - 1D0) Auto-inline,Xref |
375 P-- 123v ----> ENDDO                                                        Auto-inline,Xref |
376 123 C* calculate Poison Probability of JA cancers being observed            Generated |
377 123  IF (JA .GT. 1) THEN                                                    Auto-inline,Xref |
378 123  CUMPRB(1) = EXP((-AMEAN))                                              Auto-inline,Xref |
379 123  PROB1X = CUMPRB(1)                                                     Auto-inline,Xref |
380 123  IF (JA - 1 .GE. 10) THEN                                               Auto-inline,Xref |
381 123 CDIR@ IVDEP                                                             Generated |
382 123v -DO J1X = 1, JA - 1                                                    Auto-inline,Xref |
383 123v  D1V(J1X) = AMEAN*CONS(1+J1X)                                          Auto-inline,Xref |
384 123v - ENDDO                                                                Auto-inline,Xref |
385 1234 - DO J1X = 1, JA - 1                                                   Auto-inline,Xref |
386 1234  CUMPRB(1+J1X) = D1V(J1X)*CUMPRB(J1X)                                  Auto-inline,Xref |
387 1234 ->ENDDO                                                                Auto-inline,Xref |
388 123 CDIR@ IVDEP                                                             Generated |
389 123v - DO J1X = 1, JA - 1                                                   Auto-inline,Xref |
390 123v  PROB1X = PROB1X + CUMPRB(1+J1X)                                       Auto-inline,Xref |
391 123v-> END DO                                                               Auto-inline,Xref |
392 123   ELSE                                                                  Auto-inline,Xref |
393 123 CDIR@ NEXTSCALAR                                                        Generated |
394 1234  DO J1X = 2, JA                                                        Auto-inline,Xref |
395 1234  CUMPRB(J1X) = AMEAN*CONS(J1X)*CUMPRB(J1X-1)                           Auto-inline,Xref |
396 1234  PROB1X = PROB1X + CUMPRB(J1X)                                         Auto-inline,Xref |
397 1234  ENDDO                                                                 Auto-inline,Xref |
398 123   ENDIF                                                                 Auto-inline,Xref |
399 123   PROB1X = 1.0 - PROB1X                                                 Auto-inline,Xref |
400 123   ELSE                                                                  Auto-inline,Xref |
401 123 C* 1 OR less cancers                                                    Generated |
402 123   PROB1X = 1.0 - EXP((-AMEAN))                                          Auto-inline,Xref |
403 123   ENDIF                                                                 Auto-inline,Xref |
404 123   PROB = PROB1X                                                         Auto-inline,Xref |
405 123 C***** End of Code Expanded From Routine: POIS                          Generated /
406       CALL POIS(OBSP,OBSC,PROB) Deleted,NoXref!
407 123   NHY=NHY+1
408 123   IF(PROB.GT.THRESH)GOTO 200
409 123 C* YES it's significant so save
410 123   NCALS=NCALS+1
411 123
412 123 C* Step 7. Save circle info====================================
413 123   WRITE(9,90001) CX,CY,RADIUS,OBSP,OBSC,PROB
414 123 90001 FORMAT(3F9.3,2F9.3,F10.7)
415 123 C* END OF EASTING
416 123 -> 200 CONTINUE
417 12
418 12 C* END OF NORTHING
419 12 --> 100 CONTINUE
420 1
421 1 C
422 1 C ************ End of SEARCH LOOP for given circle radius *
423 1 C
424 1
425 1     WRITE(6,78221)RADIUS,STEP,NCALC,NDAT,NHY,NCALS
426 1 78221 FORMAT(40(1H-)/' *RADIUS=',F12.2,'KM with STEP of',I6 ' KM'/
427 1    X1H ,5X,'*Number of sites generated ',I10/
428 1    X1H ,5X,'*Number of sites examined ',I10/
429 1    X1H ,5X,'*Number of hypotheses tested ',I10/
430 1    X1H ,5X,'*Number of significant circles',I10)
431 1
432 1 C* form global stats
433 1     TOTCAL=TOTCAL+NCALC
434 1     TOTDAT=TOTDAT+NDAT
```

```
435 1     TOTNHY=TOTNHY+NHY
436 1     TOTNCS=TOTNCS+NCALS
437 1 C* go back and do another circle size
438 1 --> ENDDO
439
440 C**************************************************************
441 C* END OF ALL RUNS*********************************************
442 C**************************************************************
443       WRITE(6,887) TOTCAL,TOTDAT,TOTNHY,TOTNCS
444 887 FORMAT('0********** End of GAM Run ****************'/
445      X 1H ,'*Total sites generated is',I10/
446      X 1H ,'*Total sites examined ',I10/
447      X 1H ,'*Total hypotheses tested ',I10/
448      X 1H ,'*Total significant circles ',I10)
449
450       STOP
451       END
```

Appendix 6.3: multi-tasking compilation of the spatial interaction model

```
 1               C* si_1.f Spatial Interaction Model
 2                      IMPLICIT NONE
 3               C...Translated by FPP 6.0 (3.06G26)                        Generated,Xref
 4                    REAL SUM1U
 5                    INTEGER N
 6                    REAL BETA
 7                      PARAMETER (N=2000, BETA=0.25)
 8                      REAL T(N,N),C(N,N),O(N),D(N),P(N,N),SUM,SS,
 9                   X RANF,A(N)
10                      INTEGER I,J
11
12               C* generate some random data
13                      REAL D1X, SS1X
Generated,Xref
14                      SS=RANF()
15          1 --        DO I=1,N
16          1v -        DO J=1,N
17          1v          T(I,J)=RANF()*10.0
18          1v          C(I,J)=RANF()*100.0
19          1v ->       ENDDO
20          1 ->        ENDDO
21               C* calculate Oi and Dj
22                                                                          Generated
23     P--       CMIC@ DO ALL GUIDED(64) SHARED(O) PRIVATE(I)               Generated
24     P         CDIR@ IVDEP
25     P  v --          DO I=1,N
26     P  v             O(I)=0.0
27     P-v ->           ENDDO
28                                                                          Generated
29PRegion-       CMIC@ PARALLEL SHARED(O, T, D) PRIVATE(I, D1X, J)
30PR       1 --        DO J=1,N                                            Generated,Xref
31PR       1           D1X =                                               Generated
32PR   P-1       CMIC@ DO PARALLEL GUIDED(64)                               Generated
33PR   P  1      CDIR@ IVDEP                                                Generated,Xref
34PR   P  1v -          DO I = 1, 2000                                      Generated,Xref
35PR   P  1v               O(I) = O(I) + T(I,J)                             Generated,Xref
36PR   P  1v               D1X = D1X + T(I,J)                               Generated,Xref
37PR   P-1v ->          END DO                                              Generated
38CritReg 1      CMIC@ GUARD                                                Generated,Xref
39CritReg 1             D(J) = D(J) + D1X                                   Generated
40CritReg 1      CMIC@ END GUARD                                            Generated
41PR      1      CMIC@ END DO                                               Geleted,NoXref!
                        DO I=1,N                                            Geleted,NoXref!
                        O(I)=O(I)+T(I,J)                                    Geleted,NoXref!
                        D(J)=D(J)+T(I,J)
42PR      1 ->          ENDDO                                               Generated
43PRegion-          CMIC@ END PARALLEL                                      Deleted,NoXref!
                         ENDDO
44                  C* calculate model
45                       SS=0.0                                             Generated
46PRegion-          CMIC@ PARALLEL SHARED(SS,D,C,A,O,P,T)PRIVATE(SUM,I,J,SS1X,SUM1U)  Generated,Xref
47PR                 SS1X = 0                                               Generated
48PR   P-           CMIC@ DO PARALLEL CHUNKSIZE(16)
```

```
49PR    P  1 —            DO I = 1, 2000                                     Generated,Xref
                          DO I=1,N                                           Deleted,NoXref!
50PR    P  1
51PR    P  1        C* Calc A(I)
52PR    P  1                    SUM1U = 0.0                                  Generated,Xref
53PR    P  1        CDIR@     IVDEP                                          Generated
54PR    P  1v —               DO J = 1, 2000                                 Generated,Xref
55PR    P  1v                     SUM1U = SUM1U + D(J)*EXP((-0.25*C(I,J)))   Generated,Xref
56PR    P  1v ->               END DO                                        Generated,Xref
57PR    P  1                   A(I) = 1.0/SUM1U                              Generated,Xref
                              SUM=0.0                                        Deleted,NoXref!
                              DO J=1,N                                       Deleted,NoXref!
                              SUM=SUM+D(J)*EXP(-BETA*C(I,J))                 Deleted,NoXref!
                              ENDDO                                          Deleted,NoXref!
                              A(I)=1.0/SUM                                   Deleted,NoXref!
58PR    P  1        C* calc model
59PR    P  1        CDIR@    IVDEP                                           Generated
60PR    P  1v —             DO J = 1, 2000                                   Generated,Xref
61PR    P  1v                   P(I,J) = A(I)*O(I)*D(J)*EXP((-0.25*C(I,J)))  Generated,Xref
62PR    P  1v                   SS1X = SS1X + ((P(I,J)-T(I,J))**2)           Generated,Xref
63PR    P  1v ->            END DO                                           Generated,Xref
                       DO J=1,N                                              Deleted,NoXref!
                       P(I,J)=A(I)*O(I)*D(J)*EXP(-BETA*C(I,J))               Deleted,NoXref!
                       SS=SS+(P(I,J)-T(I,J))**2                              Deleted,NoXref!
                       ENDDO                                                 Deleted,NoXref!
64PR    P  1
65PR    P–1 ->               END DO                                          Generated,Xref
66CritReg          CMIC@ GUARD                                               Generated
67CritReg              SS = SS + SS1X                                        Generated,Xref
68CritReg          CMIC@ END GUARD                                           Generated
69PR                CMIC@ END DO                                             Generated
70PRegion-         CMIC@ END PARALLEL                                        Generated
                       ENDDO                                                 Deleted,NoXref!
71
72                       SUM=N
73                       SS=SS/(SUM*SUM)
74                       WRITE(6,123)SS
75              123     FORMAT(F15.9)
76                       STOP
77                       END
```

Appendix 6.4: HPF spatial interaction model

```
Program sim
!* si_1.f Spatial Interaction Model
        IMPLICIT NONE
        INTEGER, parameter: : N=1000
        REAL, parameter : : BETA=0.25
        REAL T(N,N),C(N,N),O(N),D(N),P(N,N),SUMS(N),SS,
        REAL SUMX,A(N)
        INTEGER I,J
!HPF$ distribute (BLOCK,*) : : T,C
!HPF$ distribute (BLOCK) : : O,SUMS,A
!* Read in some data
        read *,T
        read *,c
!* calculate Oi and Dj

        O=0.0
        D=0.0

        O=sum(T,1)
        D=sum(t,2)
!* calculate model
        SS=0.0
!* Calc A(I)
        SUMS=0.0
!HPF$ INDEPENDENT
        do i=1,n
                A(i)=1.0/sum(d*exp(-beta*c(i,: )))
!* calc model
                ss=ss+sum(((A(I)*O(I)*D*EXP(-BETA*C(I,: )))- &                           T(i,:
))**2)
        enddo
        SUMX=N
        SS=SS/(SUMX*SUMX)
```

```
        WRITE(6,123)SS
123 FORMAT(F15.9)
        STOP
END program sim
```

7 Parallel programming using simple message passing

The next two chapters are essential reading if you are interested in a non-technical introduction to message passing. Message passing is the principal way by which parallel HPC machines are programmed. At first sight it may appear very difficult, but in essence there is an underlying simplicity that is wondrous to behold. This chapter introduces the subject by way of easy-to-understand examples. If you are really serious about either writing portable parallel code or squeezing the last drop of performance out HPC or making use of workstation farms then the next two chapters are what you need to concentrate on.

7.1 Introduction

It would appear that the distributed-memory MIMD machines have 'won' the architecture battle at the high end. Message passing is important because it is the natural programming approach for these machines. It is also the most basic, general and flexible approach to parallel programming. However, message passing is intended mainly for MIMD machines. An SIMD machine may be better programmed using a data parallel approach. Indeed, high-performance Fortran (HPF) can be regarded as a high-level language generator of message-passing code when run on a distributed-memory MIMD machine, but without the flexibility and complexity that you get when you have to do it all yourself. This is precisely the case an assembly programmer would make when arguing the case for assembly programming versus any high-level language. Fortunately, this analogy is a poor one, because message passing is nothing remotely as primeval as assembly programming. The message-passing programming model is probably the future of HPC in a parallel world. So at present it is definitely worth while learning how to do it.

The advantages of message passing include an ability to scale to very large numbers of processors, flexibility in writing efficient parallel code, a means of future-proofing programs, a high degree of portability, and vendor independence. The principal penalty you pay is the extent of the software changes and the effort needed to detect, enhance and express parallelisation so as to take maximum advantage of the potential of distributed-memory processors. In practice, this means that you have to rewrite your code, rethink the algorithm

and express it in a form suitable for message passing. This is often non-trivial, but it need not be too difficult, and it frequently leaves you with a feeling of euphoria and achievement when it finally works. However, learning how to write good code for message passing is something that you only do if you have to; or because of curiosity or as part of a new skills acquisition programme. Put even more bluntly, if you think that you can do it well in HPF then do it via that route, at least initially. If the form of parallelism is uneven, not obviously data parallel, or there are few DO loops that can be shared out, then you have (at present) no choice but to use message passing. Indeed, once over the initial shock it often turns out to be easier than you may at first have thought possible.

Landau and Fink (1993: p. 331) write: 'parallel programming becomes more demanding as the number of processors increases, and it may be best to rewrite a code written for one CPU rather than try to convert it to parallel'. This is sound advice. Sooner or later you may have to bite the bullet! Quite simply, your old cuddly serial code is probably no longer suitable. It is not merely a translation and porting job but an opportunity for a complete algorithm rethink. It should also be added that with message passing you no longer have any choice but to add new lines to at least some of the code, so you might as well take this opportunity to check and rethink the algorithms on which it is based. For instance, why not consider the following questions. Is it sufficiently parallel? Could it be expressed differently? Will the performance scale as the number of processors increases? Is it offering the right levels of parallel granularity? Is there any obvious mechanism for increasing the size of the parallel tasks as processor speeds increase? Is there a better parallel way of doing it? Is the code of sufficient importance to you (or others) to justify the effort to parallelise it? Do you really, really need the additional speed, or could you get away with leaving it running on your workstation for a week or two? Do you expect the program to be run many times? Is the science sufficiently good to justify the allocation of scarce HPC resources to it? This does not mean that message passing has to be performed on an HPC. Indeed, one of the nice features is that you can write message-passing code on your workstation and even use it to create your own virtual HPC using your local area network and your department's workstations or PCs at night, when they are idle. So you are by no means locked into and totally dependent on a world of very expensive HPC hardware. You can prove your algorithms work and even look likely to scale long before you log on to a parallel HPC.

As you gain experience, you will get better at it. It is all rather an immense intellectual challenge of considerable excitement. The hope is that the code you produce will simultaneously deliver the promise of parallel processing in your application area, work extremely well on current machines and continue to do well on new and as yet undefined parallel hardware of the future that has not yet been announced. It goes without saying that the effort is only justified by your needs and ability to justify large-scale parallel-processing power. We are not talking here about a Noddy code that runs in 5 seconds on a toy problem. Parallel processing is all about the top end of high-performance computing in the context of significant research of major interest. It is when you are on the edge of what is conveniently possible or just beyond that the real thrills and spills, the big challenge, kicks in. Any buzz, excitement or enthusiasm generated here is really useful as an aid in the forthcoming message-passing struggle, so let battle commence!

7.2 Message passing?

7.2.1 What is it?

Message passing is the main alternative to shared-data programming. In a message-passing program, processes do not communicate invisibly through shared data; instead, they send and receive data as explicit messages. It is as simple as that. Gropp *et al.* (1994) write, 'The message-passing model posits a set of processes that have only local memory but are able to communicate by sending and receiving messages. It is a defining feature of the message-passing model that data transfer from the local memory of one process to the local memory of another requires operations to be performed by both processors'.

The term 'message' is somewhat misleading, but equally it is hard to imagine a better word. It also reflects the historical origins of certain types of operating system now lost in the mists of time. An analogy with postal mail is a good one. Letters contain a message. Each letter has a destination address, and the sender often includes his or her address on it as well. The postal service knows where to send it and where (approximately or accurately) it came from. Now move on to a telephone system. The process is essentially the same, except that the origin and destination numbers are known and the 'letter' is now an analogue or digital version of an audio message. Now move on to a distributed-memory MIMD computer. The messages are now bursts of digital data sent from one processor at a given address to another at a given address over an interconnection network. The processors have to do this because in a distributed-memory machine each processor has its own local memory. As a result, the global memory space consists of *M* sets of local memory (where *M* is the number of processors). If a processor wants data stored elsewhere then it needs to know which processor has it and then ask for it by sending a message to that processor and, after a wait, receiving it in the form of another message. The tricky part is that the processor with the data has to expect to be sent a message and therefore be ready and waiting to receive these requests. This may sound difficult, but it is actually quite easy.

Keep telling yourself that the basic idea is very simple. Imagine a number of processors, each with its own program and local data. If it helps, then each processor can be viewed as a workstation linked via a network. Data are exchanged between pairs of processors. To run on such a multi-processor machine, an algorithm has to be decomposed into a set of data transfers and blocks of local computation that can be run on each processor in parallel (i.e. simultaneously). A minor complication is that messages sent by one processor to another can take different lengths of time to be performed, and all processors may not work at the same speed or be given identical amounts of work to do. Computation now becomes a set of processors communicating with messages.

7.2.2 Drawbacks

Message passing is a very powerful and general method of expressing parallelism. The principal drawback is the difficulty of designing message-passing programs or of altering existing programs for message passing. This requires more work than is involved in any other form of parallel programming. Indeed, message passing has been called the 'assembly language' of parallel computing, because you have to handle explicitly much of the complexity and detail. You also have to perform fairly major surgery on your code to put it into a message-passing form. Suddenly, the seemingly simple and straightforward assignment

```
X = Y
```

may now involve the processor with memory space for X to be stored on it sending a message to one (or more) explicitly named processor asking for the value of Y. This forces you to rethink how your algorithm works in order to make good use of *M* processors (where *M* is typically in the range ten to several thousand) so that data and/or computational tasks can be shared out efficiently. If that is not enough, you will probably find either that it does not work too well or that performance does not improve with increasing numbers of processors; or even worse on the scale of programming catastrophes, the wrong result is obtained and it is not at all obvious why or what went wrong because the code may have finished quite normally. At first sight, message passing looks like a major backward step into a previous era of computer programming when debugging was largely a mix of mind experiments, intuition, black magic and print statements, sometimes tinged with dashes of luck and good fortune! Indeed it is, but the effort is worth while because it constitutes an extremely effective programming methodology for making the most of distributed-memory HPC hardware. Pacheco (1997: p. 7) argues 'if the design process is sufficiently deliberate, it doesn't take an undue effort to design extremely sophisticated programs'. This task is also becoming easier all the time as more libraries are written for message-passing systems, thereby providing useful building blocks for new applications.

7.2.3 Advantages

According to Gropp *et al.* (1994), the following four advantages more than justify the effort.

1 *The universality of the concept.* Message passing fits well on separate processors connected by both fast and slow communication networks. These

include PCs, workstations, and vector and parallel supercomputers. It is a programming paradigm that will stand the test of time.

2 *Expressivity.* Message passing offers a complete environment in which to write parallel algorithms. Gropp *et al.* (1994: p. 8) note that 'some find its anthropomorphic flavour useful in formulating a parallel algorithm'. It is well suited to adaptive, self-scheduling algorithms. In many ways, it adds the control and flexibility that is missing from data parallel programming, HPF and other simpler programming paradigms.
3 *Debugging is easier.* The principal benefit is that message passing makes memory referencing more explicit, making it easier to spot erroneous reads and writes than it is on a shared-memory machine. This is not to say that debugging is actually easy (or easier) under message passing, but it is easier to spot some types of common logical error in the code because memory access is via explicit messages. A few well-placed PRINT statements may often do the trick if only you can determine where best to put them.
4 *Performance is the principal advantage.* Gropp *et al.* (1994: p. 8) write 'Message passing provides a way for the programmer to explicitly associate specific data with processors and thus allow the compiler and cache hardware to function fully'. It provides a means of exploiting highly parallel distributed-memory machines very effectively and to obtain linear scaleability. However, it is still possible to write slow code in message passing, and much depends on the skills of the parallel algorithm designer and coder.

Other benefits include:

5 *It offers a standard approach* for implementing parallel applications.
6 *It provides hardware and vendor independence* and hence portability now that there is an industry standard (MPI and MPI2).
7 *It is a useful means of safeguarding and future-proofing the investment* in algorithmic development because of the generality of the approach.
8 *The programming task rapidly becomes easier as you gain experience.* As ever, getting the first code to work is the hardest. Quite often, the same basic structure can be used over and over again in different programs.
9 *It is far easier than it looks,* because there are only a small number of subroutines that you will typically need to use.
10 *We managed it, so why not you?*

Message passing is therefore definitely worth getting to know.

7.3 Message-passing software

7.3.1 Its importance

The idea behind message passing dates from the 1970s, but the early systems were completely vendor- and hardware-specific. It is only since the late 1980s that portable systems have been developed. The best known are PVM and PARAMACS. PVM (parallel virtual machine) was developed in 1989 and has been freely available since 1991. It allows many different types of networked computer (both workstations and PCs) to be linked (via Ethernet) to form the equivalent of a single parallel machine. It provides a complete environment for parallel computing, but it will also run on dedicated parallel HPCs where the processors are connected by very high-speed networks (many times faster than Ethernet, ATM, etc.) based on proprietary vendor and highly customised technology. The same principles apply, but communication speeds and bandwidth are very different from a network of PCs linked by a LAN. PARAMACS (parallel macros) is another, albeit more restricted, basic message-passing system developed at Argone National Laboratory; see Boyle *et al.* (1987), Bomans *et al.* (1990). There are a number of other message-passing libraries. Some of the better-known ones are p4, EXPRESS, Schedule and VCR. However, you really should not be using, or starting to use, any of these, as there is now a standard (MPI). Equally, beware of various trendy and very appealing languages designed for parallel computing by computer scientists (such as Charm, Linda, pen, etc.) and stay with international standard-based languages such as HPF and MPI. Avoid being tempted by vendor-specific add-ons to your favourite programming language (i.e. previously Craft on the Cray T3D or parallel C++ or parallel Java) because their appeal is likely to be short-term (T3D Craft is now extinct), their future uncertain, and the resulting code is probably non-portable (although conversion costs may not be so great unless you use features that are unique to specific hardware). Parallel-processing HPC is here to stay, and the trick is to ensure that your programs are equally long-lived. You should therefore be aiming to write code that has a shelf-life of more than two or three years. Indeed, if the problem is important enough to justify parallel processing, then this will usually be the case! So think longer-term, or you could be wasting all the intellectual and programming effort being expended now.

7.3.2 The message passing interface forum

The principal problem with message passing concerned the fact that historically there were multiple different message-passing libraries, all in competition with each other. Additionally, many are (or were) specific to unique hardware. Portability was achieved at the expense of reduced functionality, and the performance advantages of non-portable systems were lost. This potentially disastrous situation resulted in the MPI forum being established in 1992. This forum was an *ad hoc* committee of industry experts that was formed to solve the problems caused by the lack of standards. Its goals were to define a portable standard, to operate in an open way that anyone could join and to be finished in one year. Indeed, a standard in the form of the message-passing interface (MPI) was published in 1994 based on work by over sixty people from forty organisations. It is now a global standard and has been widely adopted. MPI is stable, portable and

Table 7.1 The six most used MPI subroutines.

Subroutine	*Function*
MPI_INIT	initialises MPI
MPI_COMM_SIZE	specifies how many processors are available
MPI_COMM_RANK	identifies a processor
MPI_SEND	sends a message
MPI_RECV	receives a message
MPI_FINALISE	ends MPI

readily available. It is also likely to be long-lived, so we would strongly commend MPI and its successor, MPI2, to you.

So what is special about MPI? Gropp *et al.* (1994: p. 11) explain: 'The primary aim of the MPI specification is to demonstrate that users need not compromise among efficiency, portability and functionality. This means that one can write portable programs that can still take advantage of the specialised hardware and software offered by individual vendors'. This triple goal of efficiency, portability and functionality is a key design feature. MPI is an attempt to keep the best features of many existing message-passing systems by offering a form of layered message passing. Subroutines use lower-level libraries unique to specific hardware to perform the specified tasks. It is this feature that ensures the portability (because the specification of the interface is fixed) and moderate to good efficiency (since the low-level subroutines can be made specific to particular hardware and when locally optimised for specific hardware should be highly efficient).

7.3.3 What is MPI?

So MPI is a library of subroutine specifications and not a language. These subroutines can be called from C and Fortran programs (in MPI2 C++). The interface is the definition of the parameters or arguments used in the subroutines and defines what the subroutine will do. The task of producing subroutines for particular hardware that conform to the standard is left for others (i.e. vendors) to perform. The key attractions for users are, first, that the subroutine names and arguments are standardised and hence portable across different HPC systems and, second, that MPI exists for virtually all the world's parallel HPC hardware. However, the MPI library is large, and the standard runs to 228 pages. There are 125 subroutines, but in general only about ten that are used regularly. The six most basic functions are listed in Table 7.1.

7.4 How to use MPI for SPMD

More jargon! Message passing is not difficult, but you will need to modify serial code in order to express it in a suitable form. This is not only a matter of adding

subroutine calls but you probably also need to restructure your code. Fortunately, this task is not as serious as you may think. Also, the code can be tested on a workstation or PC and does not require a multi-million ECU (or dollar) machine for development work.

Probably the hardest part is coming to terms with the single-program, multiple-data (SPMD) programming model that MPI implements. This means exactly what it says. There is a single program that runs on multiple processors! You need to remember that each processor runs the same program, but this does not mean that each processor has to do exactly the same thing or use the same data. Confused? Well there is no need to be, because there are IF statements in the program that determine which processor does what and when.

Each processor is given a unique number, which is useful to identify it. However, the numbers start at 0, not 1. You must remember this! It is essentially a product of a C programming mentality; in C, arrays start at 0 and in Fortran at 1. So if you are a Fortran programmer you do need to remember this 'feature' of MPI. The numbering of processors is important, because in message passing you use the processor number to assign tasks to each processor via lots of IF...ELSE...ENDIF statements (in Fortran, or the equivalent in C).

Remember that the same program is run on all processors simultaneously! For example, let us assume that the variable *mype* contains the *number of the processor* assigned to it by the hardware that the program is being run on. If there are eight processors, then these will be numbered

0, 1, 2, 3, 4, 5, 6, 7

So *mype* has a different value depending on which processor the code is run on. Note how the numbering starts at 0, not 1.

So if you want to make a section of the program specific to a particular processor (so that it is executed only on that processor), you can do this as follows:

```
IF(MYPE.EQ.0) THEN
X=1.0
ELSE
X=2.0
ENDIF
```

The only really confusing part of message passing is realising that in this example the same program is run on hardware with eight processors. The following happens:

1 Each of the eight processors executes the same program (in parallel).
2 The processor numbered 0 does something different from processors 1, 2, 3, 4, 5, 6, 7 because of the IF statement.
3 There are eight processors but no processor number 8 because the

numbering starts at 0 (this is for Fortran programmers, who usually think that 1 is where you start).

4 The number of processors available can range from one to something large, so do not 'hardwire' in too many specific processor numbers (other than processor 0), because they may not exist on the machine than runs your program. However, there will always be a processor 0.

5 There are now eight separate storage allocations for the variable X. The layout is as follows:

    processor 0 has its local X set at 1.0
    processor 1 has its local X set at 2.0
    processor 2 has its local X set at 2.0
    processor 3 has its local X set at 2.0
    processor 4 has its local X set at 2.0
    processor 5 has its local X set at 2.0
    processor 6 has its local X set at 2.0
    processor 7 has its local X set at 2.0
    processor 8 DOES NOT EXIST!

6 If processor 0 wishes to add its value of X to those stored on other processors, then it has to obtain these values by sending messages to each processor asking for them. The complication here is that each of the processors (1 to 7) has to expect to receive a message. Since they are not psychic, you have to arrange the code so that this happens.

7.5 Example 1: probably the world's simplest MPI program that could be useful

Consider a simple program that sets a variable (called A) to a value of 27 and then calculates a square root. If you programmed this task in Fortran for a serial machine, then the following code would suffice:

```
REAL A, B, SQRT
A=27.0
B=SQRT (A)
STOP
END
```

Now consider a parallel system with two processors, and for some 'daft' but illustrative reason you decide that you want to compute the SQRT on the second processor. The algorithm becomes instantly more complicated to accommodate message passing. You now have to write a new version of the program to set A = 27 and then send A as a message from processor 0 to processor 1. Processor 1 has to expect to receive this message, compute a square root and send the result back to processor 0 as another message. This is a kind of ping-pong task only slightly better than the 'hello world' example that most other parallel programming books use.

Consider a step-by-step algorithm for doing this, written in some kind of abstract pseudo code:

Step 1: Wake MPI up.
Step 2: On processor 0, set local variable A = 27.0 and send it to processor 1 as a message.
Step 3: Expect to receive a message on processor 1, so wait until one arrives. When it does, the value (in this case 27.0) will be stored locally in variable A.
Step 4: Do something to B (B = SQRT (A)) on processor 1 and send the answer (B) back to processor 0 in another message.
Step 5: On processor 0, expect to receive a message containing a value from processor 1; wait until it arrives and store it in B.
Step 6: Close MPI down, end and go to pub to celebrate (this latter bit of the MPI standard got left out because of uncertainty as to who should pay!).

Written in pseudo code, this becomes

```
if (mype.eq.0) then (this is processor 0)
      A=27
      send A to processor 1
else (this is processor 1)
        wait until a value arrives from processor 0
        store it in A
endif

if (mype.eq.0) then (this is processor 0)
        wait to receive a value from processor 1 and
        store it in B
else (this is processor 1)
      B=SQRT (A)
      send value of B to processor 0
endif
```

Appendix 7.1 gives a Fortran version of this MPI program. At first sight, twenty-two lines of code for the MPI version of a five-line Fortran program does not seem like a good deal. However, the tenfold code expansion (the two useful lines expands into twenty) is a small number problem that is most unlikely to be repeated in real applications. A far less than twofold code expansion is more typical. Nevertheless, this example does illustrate a very simple MPI application and gives some clues as to the added complexity of message passing.

Indeed, it is worth examining the code in Appendix 7.1 closely, because most of the general sections will repeat in virtually every MPI application that you may write. The MPI_INIT initialises the MPI library, and MPI_COMM_RANK determines the processor number (here MYPE is 0 or 1) of the processor running the code. The meaning of all the arguments in these subroutine calls is described in the MPI standard. You can download it (free) from the World Wide Web; see http://www-unix.mcs.anl.gov/mpi/mpich or any local documentation or MPI books such as Gropp *et al.* (1994).

Remember that the same program is run on both processors. If it is processor 0 (MYPE = 0) then it sets A = 27.0 and sends a message to processor 1, else (MYPE = 1) the processor waits until it receives a message (a value 27.0), which is stored in B. The next IF statement says that if you are processor 0 then expect to receive a message from processor 1 and store it as B, else sqrt (b) and send the result to processor 0 as a message. Phew! A lot of effort here, but this is how MPI implements an SPDM programming model. It is very flexible. You could remove the second IF...ELSE block by moving it all into the first one. So if processor 0 then send out A and wait until a message is received from processor 1 containing B, else if processor 1 then wait for a message from processor 0, compute B and send it back. *The hardest part is matching the sends and receives.* If you get this wrong (and it is easy to get confused or 'lost'), then the program will not work but will be stalled by a processor waiting for a message that will never be sent. This is probably the easiest bug to make. Once this process of sending and receiving messages is mastered, then the rest is fairly easy.

It is still a lot of additional effort compared with the serial code, but this is the cost of running a single program on multiple processors using the processor number to determine what is being done on each of them. The alternative is to program each processor individually, but this would be many times worse and almost impossible to debug, because each program would be different. The SPMD approach is a useful rationalisation and transformation of multiple programs and multiple data implicit in MIMD hardware on to a single code. In practice, it is not much harder than the ping-pong example, hence this is a good place to begin. If you understand this example then continue; otherwise, read Section 7.5 again and again and again, or send the authors an e-mail complaining!

7.6 Example 2: sum M numbers

The simplest type of parallel programming involves sharing DO loops and hence the associated data and computation over multiple processors – a data parallel type of problem decomposition. Figure 7.1 shows how a set of M data values could be distributed across a serial and/or global-memory machine and a distributed-memory one. A master processor (this is best called processor 0, because there is always a processor 0) shares out the data (and thus implicitly also the computational load), keeping a share back for itself. This is the so-called 'master–slave' approach. Very unsound terminology here – you think! One processor (processor 0) is the master and sends out tasks to the slaves! There is a lot here to get the deconstructionist–feminist–postmodernist fraternity over-excited; however, we are merely using words in common currency! If you do not

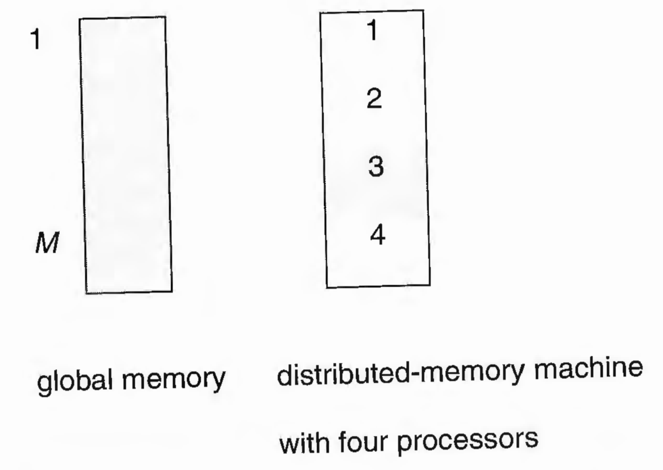

Figure 7.1 Layout of the M numbers on a global- and a distributed-memory machine.

like this terminology then you could try leader and consensual co-workers (but it is a bit of a mouthful, and spelling it correctly can sometimes be difficult).

This data parallel style of programming is easily implemented in MPI, albeit with much greater effort than it would be in HPF. Suppose now that the task is to use N processors to sum up M random numbers, where $M > N$. The following algorithm can be employed.

Step 1: Create M random numbers on the master processor (0) and send out M/N numbers to each slave processor from the master processor.
Step 2: Sum all M/N numbers on this processor.
Step 3: Pass the sum back to the master processor.
Step 4: Master processor collects all sums, adds them up and reports the result.

A serial version of this algorithm in Fortran is given below.

```
      PARAMETER (M=1 000 000)
      REAL X(M), SUM
C* generate data
      DO I=1, M
C* create random data using Cray random number
      generator
      X (I)=RANF ()
      ENDDO
C* sum numbers
      SUM=0.0
      DO I=1, M
      SUM=SUM+X (I)
      ENDDO
C* write out answer
      WRITE (6, 1) SUM
1     FORMAT ('SUM is', F15.3)
      STOP
      END
```

There are thirteen lines of executable Fortran here, and this is a very simple program. This code would run on a PC or workstation that could handle *M* = 1,000,000 words of memory. Note that the computing time is a linear function of the size of *M*.

The MPI program for implementing this algorithm is given in Appendix 7.2. There are now forty executable statements. A plain English description is as follows. There is the usual MPI wake-up call (MPI_INIT). A timing routine (MPI_WTIME) is used to keep note of the time. The routine MPI_COMM_SIZE returns the number of processors (as *npes*), so remember that these are assigned numbers from 0 to *npes*−1, while MPI_COMM_RANK returns the processor number (*myid*) of the particular processor being used here. Remember that this is the potentially confusing bit, since if *npes* = 8, then the processors will be numbered 0, 1, 2, 3, 4, 5, 6, 7. The same program runs on each of them, so the only way to make sections of the code specific to a particular processor is via this processor number, which is called *myid*.

The first DO loop calculates the share of the *M* numbers to be sent to each processor. It divides the *M* numbers by the number of processors and stores the result in *SIZE*, so that SIZE (I) contains the allocation of the data to be sent to the Ith processor. The second DO loop allocates the residual (if *M* is not exactly divisible by *npes*) between the processors to try to ensure that each has an equal amount of work to do. These two DO loops are a little tricky (because the processor numbering starts at zero), so let us test the logic on a small example, working it out by hand in order to be certain we get it right, because any errors here would be catastrophic and possibly very difficult to detect later as the program would run but produce the wrong answer.

In this worked example, let us assume *npes* = 4 and *M* = 8, so each processor will get exactly 2 numbers each. The results are

SIZE (0) = 8/4 = 2

SIZE (1) = 8/4 = 2

SIZE (2) = 8/4 = 2

SIZE (3) = 8/4 = 2

and the second DO loop will be ignored.

If *M* = 10 and *npes* = 4, then the following work allocation would occur:

SIZE (0) = 10/4 = 2

SIZE (1) = 10/4 = 2

SIZE (2) = 10/4 = 2

SIZE (3) = 10/4 = 2

Because in integer arithmetic as implemented in Fortran or C the decimal part is lost, 10.0/4.0 = 2.5 becomes 2 when converted to an integer (you round down to the nearest whole number). The second DO loop now allocates the residual, so the work allocation now becomes:

SIZE (0) = 2 + 1

SIZE (1) = 2 + 1

SIZE (2) = 2

SIZE (3) = 2

Note that processors 2 and 3 have slightly less to do than processors 0 and 1, and this will result in the total wall clock or elapsed time being determined by the time taken by processor 0.

Note here the importance of writing code that can handle the case where only one processor (processor 0) is available and also when there is a much larger number; indeed, the logic copes well with all of the possibilities. This is worth doing, because it will allow the parallel code to run on either a serial machine or parallel hardware with a variable number of processors. However, beware! There are three situations to be tested for:

1 only one processor;
2 when the number of processors is exactly divisible into *M*; and
3 when (2) does not hold.

It is important you obtain the same results with all three *and* that they match a gold standard based on a purely non-MPI, serial equivalent. Take no chances. Program and logic bugs of all kinds thrive extremely well in parallel worlds; they just love the complexity and possible confusion that parallelism causes serially minded humans! If you leave out some of the work, due to an error, the program will terminate normally. Everything will look fine, but the result will be wrong! The danger is that you may not know what the correct result is.

Returning to the program. It might be a good idea to sum the values in *SIZE* to ensure that it always equals *M*. It is better to be safe than sorry, because if you happen to get this processor–work allocation wrong it could take a very long time and much effort to determine why the result is wrong or why it works correctly with one number of processors but not with others. When it can be done easily, it is often worth while adding extra lines of code to check the logical accuracy or consistency of your algorithm's code. This is essential, because processor–work allocations are both critical and potentially error-prone. For life-long Fortran programmers, a processor numbering scheme that starts at 0 is a potential and ongoing source of possible logical confusion. So add logical consistency checks to be safe.

Note also that any sections of the code not made dependent on processor number (by IF...THEN statements) are run simultaneously by all the processors. So *SIZE* is actually computed simultaneously on all *M* processors. This may seem wasteful, but it is actually faster than computing it on processor 0 and subsequently sending out the results as messages. It is important in MPI (and in all other parallel systems) to minimise the amount of interconnection communications traffic being generated. The processors can access their local memories much more quickly than they can send or receive messages from other processors. However, they also do arithmetic much, much faster than even local memory accessing and many times faster than message processing. ***So good parallel algorithms not only need to exploit parallelism but also should seek to localise memory access and explicitly minimise as far as is practicable the amount of message passing being performed.*** This is not only something that you leave to the code-writing stage but, as far as possible, that also needs to be built into the design of the parallel algorithm being coded so that this feature continues to apply as the number of processors is increased.

The third set of DO loops generates random numbers and sends out shares to each processor. Note that this is run only on processor 0. A random number generator is used here instead of reading data from a file in order to make the code self-contained. Processor 0 also keeps a chunk of the work for itself. Also see how the slave processor numbers are part of the DO loop, using 'I' as the index. Remember the potential for confusion here.

To summarise, *npes* contains the number of processors, but the processor numbering goes from 0 to *npes*−1. The MPI subroutine MPI_SEND then sends a message to each of the processors that contains its share of the data. The array X stored on processor 0 is filled with SIZE (I) values, and this is then sent to processor I. The remaining SIZE (0) values are left for processor 0 to handle. A useful trick here would be to have each slave processor verify that the message just received actually came from processor 0. This would be an example of a defensive programming style that is often useful in more complex situations. While processor 0 is doing all this work the other processors are idle, waiting to receive their share of the data. This is the serial part of the program; *cf.* Amdahl's law.

The next section of code does the summation (see the fourth DO loop). Each processor sums its own chunk of data. 'OK, so how does it know that it should sum what is in X(I)?' Well, this bit of the code is executed simultaneously on all processors, *but* each processor has been sent its part of the data by the previous code in DO loop three.

There are now *M* different processor-specific sums that need to be added up. So processor 0 asks each processor in turn to send its result so that it can then add them all up. The answer is accumulated in *tsum*. This involves processor 0 asking processors 1 to *npes*−1 for a message and when this is received summing the result. Meanwhile, all the remaining processors are sending messages containing their part of the answer. The result is a parallel algorithm expressed

in a very general form that will work with any size of *M*, provided that each processor has sufficient local memory to hold its share of the X values.

At this point, you (the reader) really do need to check that you have understood what has happened. If you do not, then reread this section. This is important, as you will not get much further with message passing until this program is seen as trivial and you understand exactly what is going on in it. If you think you do understand this process, then try the following questions (answers in Appendix 7.3):

Question 1. On how many processors is the fourth DO loop run?
(a) 1
(b) 7
(c) 8.

Question 2. On how many processors is the third DO loop run?
(a) 1
(b) 7
(c) 8.

Question 3. On how many processors is the second DO loop run?
(a) 1
(b) 0
(c) 8.

Question 4. On which processor is *tsum* stored?
(a) 0
(b) all of them
(c) 1, 2, 3, 4, 5, 6, 7.

Question 5. What is the value of *tsum* on processor 4?
(a) the same as on processor 0
(b) different from on processor 0
(c) undefined.

Question 6. How many different versions of *sum* exist?
(a) 1
(b) 7
(c) 8.

Finally, note that even in this simple problem there is considerable potential for logical 'mistakes' to creep into the code. Some of these are as follows:

1 errors in the allocation of work so that not all *M* values of X are summed;
2 the same values of X could be sent to each processor by error; and
3 the final global summation could easily use only processor 0 values of *sum* (if the results from the other processors were not sent or received).

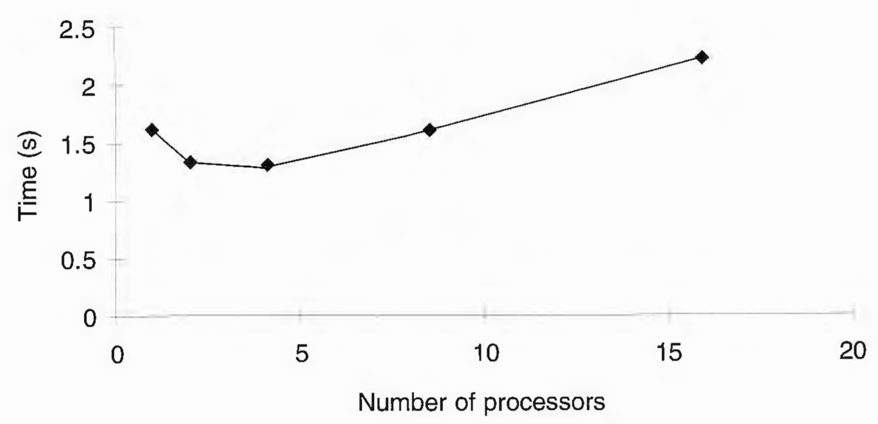

Figure 7.2 Wall clock time for Example 2.

None of these would cause the program to fail during execution or produce any identifiable error. Instead, the results would be 'wrong'. How you verify the results is always a difficult question in all types of programming, but the degrees of difficulty are far greater in a parallel-computing environment, especially when the results may be uncheckable. For instance, a several-hour run may not be verifiable. It worked on a small data set run on few processors, and you have to hope that it worked correctly in the full-size application.

The results of running this program on the Cray T3D are shown in Figures 7.2 and 7.3. Figure 7.2 shows the wall clock time. This is the elapsed time for the run. It decreases initially, reaching a minimum with four processors, and then increases again! A good code for a suitable problem would exhibit a continued decrease in wall clock time as the number of processors increased until it reached that time due to the serial part of the code. Remember the discussions in earlier chapters. Parallel processing does not reduce the total amount of CPU time being expended and usually increases it, hopefully only slightly, due to overheads. The basic equation for the run time on a serial machine is as follows:

total time_1 = time taken by 1 processor

For a parallel machine with *K* processors it is

total time_2 = sum of all processors times expended by each of the *K* processors, and

total time_3 = wall clock time since start of run

Note that in parallel programming the only relevant time is total time_3, and if you are running on *M* processors on a well-behaved problem then time_3 could

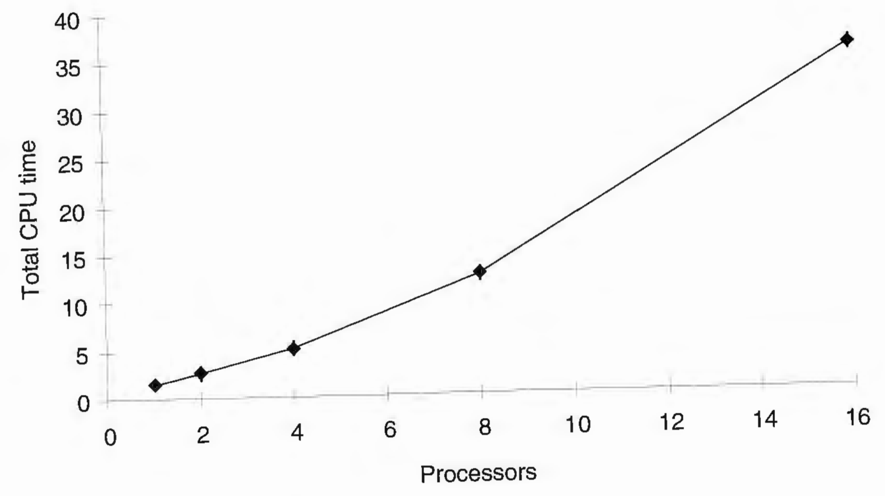

Figure 7.3 Total processor time for Example 2.

well be $1/M$th of total time_1. Note also that total time_2 will always exceed total time_1, even if the algorithm is identical and only one processor is being used. The difference is the parallel-processing overhead implicit in the code, which will usually be small. Total time_3 may (if you are lucky and clever enough to do it) be less than total time_1 divided by *M*, but only if the algorithm has some special superlinear performance features (i.e. it somehow benefits from its parallelism) or benefits from special features of the parallel hardware (*viz.* more or faster caching).

Figure 7.3 shows the total CPU time used. This should be increasing as a straight line, not as an upward curve. The reason here is that although this code is highly parallel it does not scale well. In fact, instead of a speeding-up there is a slowdown! As more processors are used so more messages are passed around, and at the same time there is a reduction in the amount of work being performed by each processor. There soon comes a point where the communications overheads dominate the run times. Quite simply, insufficient work is being done in the DO loops. In other words, the parallelism is too fine-grained for the processor being used. If you increase the amount of data being processed and increase the number of processors it becomes even worse, because communications increase in proportion to the amount of data, while the computational loading on each processor decreases as the number of processors increases. Only if you add some really heavy-duty mathematical functions and computation to the parallel loop will the computation start to exceed the communications overheads. This balancing point is hardware-specific and hence a moving function of HPC technology and time. What worked well on a transputer in 1990 will almost certainly no longer work well on more modern parallel hardware.

A final aspect to consider is load balancing. Here there is no problem, because each processor is given the same share of the data and hence the work to perform. However, if you gave 90 per cent of the numbers to processor 4 on an eight-processor machine, then processors 0–3 and 5–7 would be idle most of the time and you would discover that total time_3 was more or less constant regardless of how many processors were used. Load balancing, or keeping all the processors working flat out for as much of the time as is possible, is discussed further in Chapter 9. It becomes a major design issue where the work is harder to share out in equal lumps or is of unequal computational intensity.

7.7 Example 3: a data parallel spatial interaction in MPI

7.7.1 The theory

Now consider a more useful example. The spatial interaction model can be reprogrammed using MPI. It makes a good but more advanced case study, because this simple-looking model displays considerable complexity when considered from an MPI point of view. When a data parallel approach and HPF was used previously this complexity was completely hidden, but so were various performance issues. With MPI both are fully exposed, but so too are various ways of improving the performance of the program that are not available by any other means.

The first task and the key design decision is once again to decide how to distribute the data. It is useful to remember that the origin-constrained spatial interaction model uses the following arrays:

1 three N by N matrixes for the flows (containing both observed and predicted values) and distance data;
2 two one-dimensional (column) arrays of N elements for the O_i and A_i terms; and
3 a one-dimensional (row) array for the N elements forming the D_j terms.

Note that both arrays (2) and (3) are one-dimensional arrays. The distinction between row and column arrays is only made because it reflects the way in which the spatial interaction model accesses the data and thus gives clues as to how it can be handled in a message passing environment.

Figure 7.4 shows the data stored on a global-memory serial PC or workstation or shared-data parallel machine. All the data are readily accessible to the processor or processors, and no visible message passing is needed to access any of it. In a distributed-memory environment there is no longer a single global memory, so each processor has either a copy of all the data or just some of it. The latter is often the only possible strategy, otherwise the maximum size of problem would be determined by the memory of a single processor rather than by the total sum of the memories of all the processors. We assume, therefore, that each processor is to have only part of the total data held in its local memory.

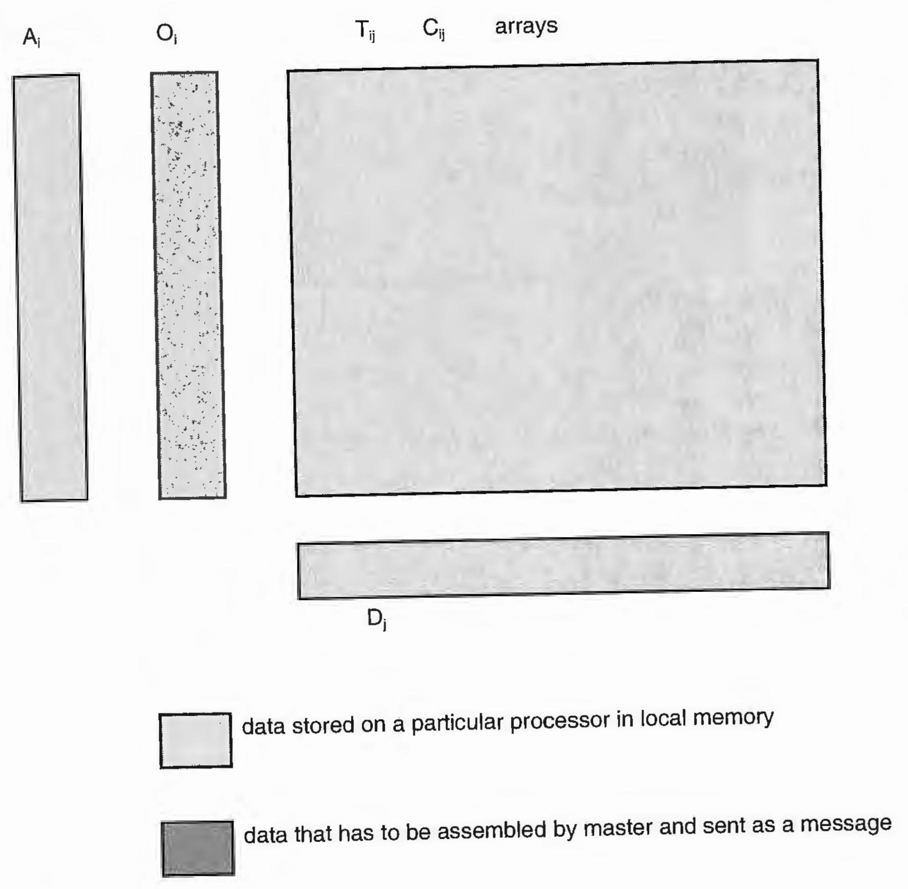
A_i
O_i
T_{ij} C_{ij} arrays
D_j
data stored on a particular processor in local memory
data that has to be assembled by master and sent as a message

Figure 7.4 Layout of spatial interaction model's arrays on a serial processor or on multiple processors with a single global memory.

It is accessible by other processors, but it now requires that all data requests and all data transfers occur as messages. The layout of the data across the processors is quite obviously crucial from a performance point of view. This is not a new problem: it occurs in HPF too. The difference is the extent of its visibility. In MPI it is explicit, in other approaches it is largely implicit and hidden, although you still have to know how to distribute it. There are various ways of distributing the data, some of which are far better than others.

One data decomposition scheme is to assign each processor a square chunk of the data. You now have to imagine what the data stored on any individual processor will look like. As in the previous summation example on a distributed-memory machine, each processor has instant access only to that part of the data held in its local memory, although it can access data held non-locally by asking for it. For example, if there are four processors then the trip and cost arrays

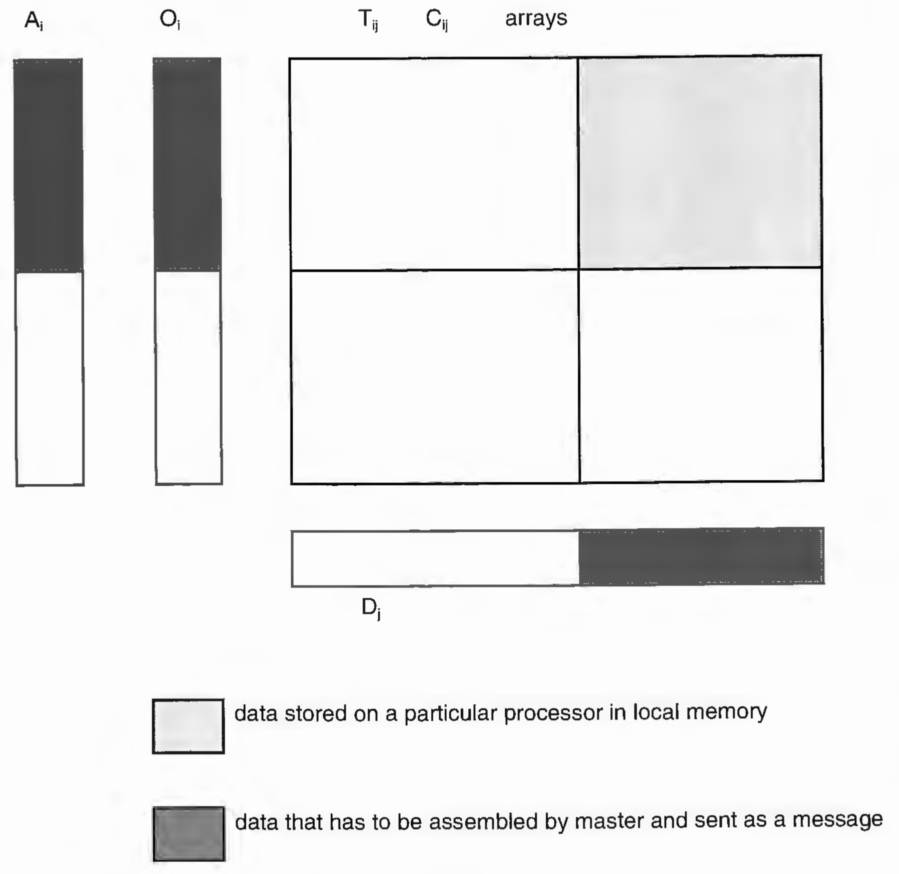
A_i
O_i
T_{ij} C_{ij} arrays
D_j
data stored on a particular processor in local memory
data that has to be assembled by master and sent as a message

Figure 7.5 A block decomposition approach to the layout of the spatial interaction model's arrays for four processors with distributed memory.

could be partitioned as shown in Figure 7.5. The problem is that no complete part of the O_i, D_j and A_i arrays can be stored easily in local memory belonging to all the processors. Instead, these quantities would have to be computed on local processors for that part of the data held in local memory, summed by the master and the results finally sent out to individual processors.

Another decomposition scheme would be by columns; see Figure 7.6. This again shows that none of the row arrays (A_i, O_i) values would be locally available. They would have to be computed by the master and sent out before the model could be computed.

Figure 7.7 shows a row decomposition scheme. Now it is only the array D_j that is not local and thus will need to be sent to and from the master.

A key question now is which layout will be best for the spatial interaction model? If you get it wrong, then you will end up generating a lot of additional

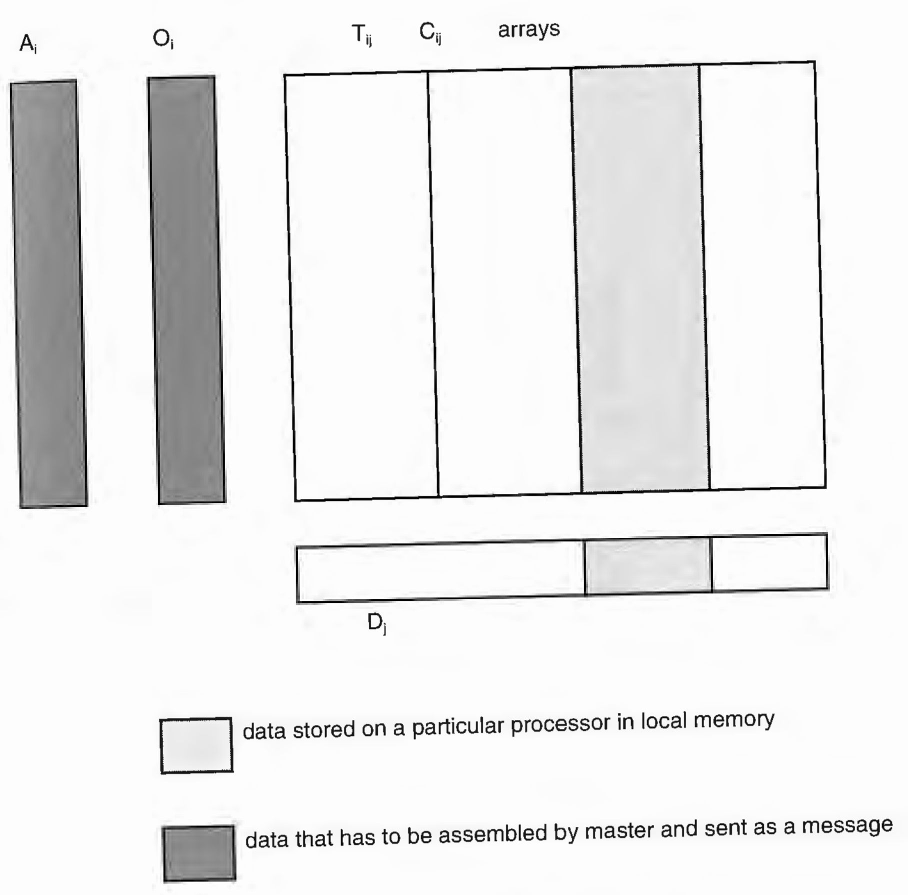
A_i
O_i
T_{ij} C_{ij} arrays
D_j
data stored on a particular processor in local memory
data that has to be assembled by master and sent as a message

Figure 7.6 A column decomposition of the layout of the spatial interaction model's arrays for four processors.

communications traffic which could have been avoided, and this will slow down or even totally wreck the performance of the program. If you compare Figures 7.5, 7.6, and 7.7 then clearly Figure 7.7 will be best, since only the D_j values would have to be computed and sent out. This row decomposition is best simply because it minimises the amount of communications traffic generated by the message passing.

Incidentally, it is very useful to add some check sums to the code in order to verify the logic of the decomposition. As noted before, it is very easy with multiple processors all having their own version of the same variable to get it wrong, to miss out some messages, or to store the results in the correct array and at a valid but logically wrong address. The compiler may not detect it but the results will be wrong! An example of this could be the computation of O_i and D_j values. These should sum to the same value as the total sum of the T_{ij} array. It may not

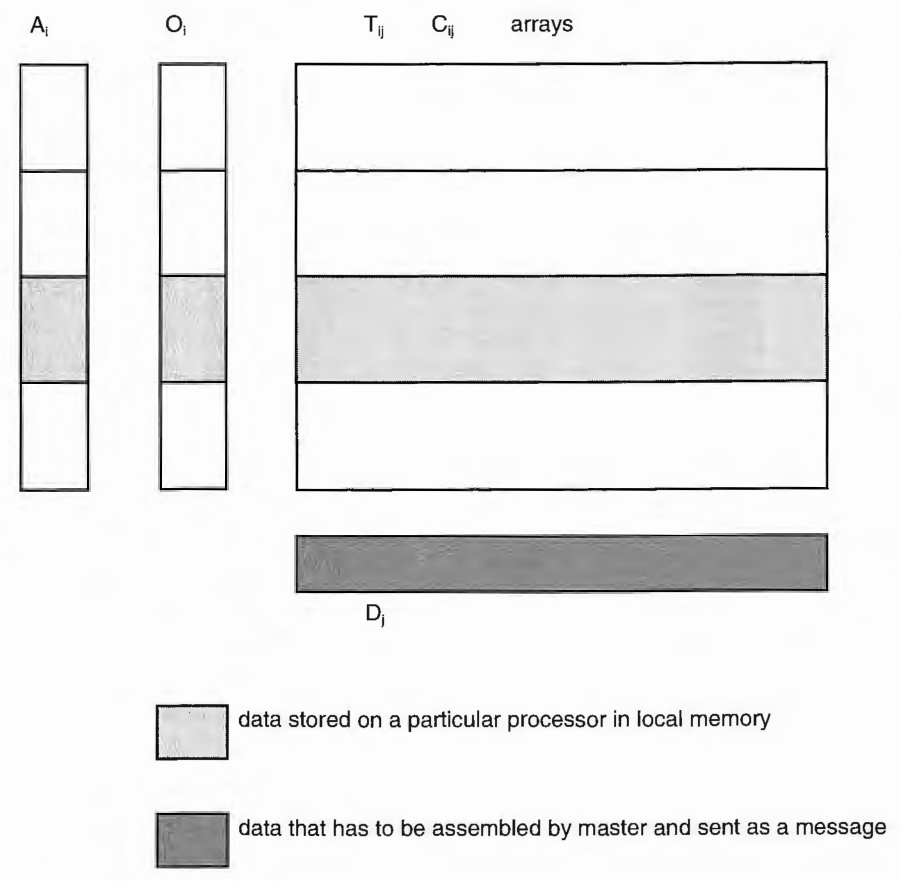
A$_i$
O$_i$
T$_{ij}$
C$_{ij}$
arrays
D$_j$
data stored on a particular processor in local memory
data that has to be assembled by master and sent as a message

Figure 7.7 A row decomposition of the layout of the spatial interaction model's arrays for four processors.

do so if you get the data decomposition logic wrong. On the other hand, consider the sum of squares variable (SS) value. Suppose that you forgot to sum the many local copies. The result would be that the performance of the model (which SS measures) would now improve with increasing numbers of processors! Check sums would spot all these errors. Otherwise, you would have to compare the results with a single processor run or a benchmark obtained elsewhere to detect whether there is a problem, but then you would have no indications of where to look for it. Another useful practice is to verify that the code produces the same result when run on different numbers of processors and with different levels of compiler optimisation – and then maybe if it is possible on different HPC machines.

Hence the spatial interaction model is a good test bed, partly because virtually any other example of data decomposition would be easier and therefore less

useful as an illustrative example. In this model, there are always some data that are not available locally and have to be sent for. The trick is to minimise how much is involved. With MPI, there is a factor of about two increase in code and a factor of ten increase in logic complexity (this perception of complexity decreases with experience), but then you can gain a factor of *M* speeding-up in performance (where *M* is the number of processors and can be quite large) or an ability to process problem sizes that are too large for any other computer due to memory restrictions. Remember that the Cray J90 used in Chapter 6 could not handle more than 2000 rows and columns, whereas the Cray T3D's limit is about fifteen times greater.

Note also the additional flexibility that MPI has provided you with. You can parallelise this model in three ways:

1 by sharing out the critical DO loops as demonstrated here, which is a form of fine-grained parallelism;
2 by running the entire model in parallel, although this requires that the number of parallel model calls is some integer multiple of the number of processors being used and that the nature of the application is best served in this way (if it is not then you will need to change the algorithm that calls the model so that there is some value of 64 or 256 or more concurrent whole model calculations); and
3 by a mixture of (1) and (2); for instance by allocating eight processors to each model and then running thirty-two models at a time in parallel on a 256-processor machine.

Currently, only MPI offers you the flexibility to do this.

7.7.2 The MPI code

The code is given in Appendix 7.4. It can be translated into English as follows. In Section 1 there is the usual MPI wake-up call (MPI_INIT) and a call to determine the number of processors available (MPI_COMM_SIZE), which is variable, and your code needs to be able to handle this variability. Typically, you would start by testing it on one processor (your local workstation) and then increase the number of processors (probably on a local workstation farm) prior to moving on to something far more powerful.

Back to the code. The next important call to MPI determines what processor you are running on. Remember the SPDM model? Then if your processor is the master task (*mype* = 0) you will create some random data using a random number generator (ranf), storing the values in the two arrays T and C. You could have done this on all the processors simultaneously by leaving out the 'if (mype.eq.0) then...section', but let us be awkward and create it on just one processor.

The next step (Section 2) is to distribute the data according to the strategy previously discussed and illustrated in Figure 7.7. This is a little more complex.

First, the master processor calculates how big a chunk each processor will receive. It could be sent out individually, but it is far more efficient to send it out in longer messages (see *buff*), and it is sent to each slave processor using MPI_SEND; T and C values are distributed to processors 1 to *npes*, but again some of the data are kept for the master. Meanwhile (Section 3) each slave processor is expecting to receive some T and C data (MPI_RECV).

Now each processor (Section 4) forms O_i and D_j sums from its share of the T data held in local memory. This is a little tricky, because while all the O_i values can be computed for any individual processor's share of the data, not all the columns will be stored locally: parts are scattered across all the processors being used. So (Section 5) the master has to collect them all, add up all the partial sums and then redistribute them again. Confused? Well, there is no need to be, it is not that difficult. You just need to be careful to separate in your head what the master processor is doing and what the slaves are up to.

The model is now calculated in parallel (Section 6). This is run on each processor concurrently, but remember that each processor is working on a different part of the data. For example, there are now *npes*−1 versions of SS (the sum of the model errors squared), each relating to part of the data. So guess what is next? Yes, you have got it. The master processor asks each slave for its version of SS. It adds them together and computes the global model sum of errors squared.

Note the numerous opportunities here for parallel bugs and programmer confusion to occur. In particular, you could accidentally distribute the same data rather than different data in Section 3 of the program. You could accidentally but wrongly compute the partial D_j sums in Sections 4 or neglect to distribute the final all data D_j values in Section 5. You could easily run the model in Section 6 on the wrong data or in Section 7 forget to gather all the local part SS values from the slave processors. How do we know about these potential bugs? Well, we have probably made them (or similar ones) while learning MPI, or watched others make them. The worst aspect of parallel programming with MPI is that the job would appear to run normally; it is just that the results would be wrong. Even worse is that if you then ran it on a single processor it would probably produce the correct answer! Arggh!

The answer to some of the potential MPI parallel confusion is to start with a clear and explicit definition of the algorithm you plan to program *before* you start programming it. A clear and accurate map of the master–slave relationship is absolutely essential. Once this is right, the rest is relatively easy apart from the odd logic bug. So remember the advice from earlier in this chapter. Add some extra code to do some additional work to test and self-check the logic. An example would be to sum the T values when they were created, get the processors to sum the O_i and D_j values they are working on and test that it equals this original value of sum T. If not, you have a possible logic error in the work distribution section, or in the additional code you have just added to test for logic errors! Of course, logic checking will not tell you where the error(s) are occurring, only that they exist. The nice aspect is that bugs that are processor-number-specific may also be detected.

Spotting and fixing a parallel code bug is really exhilarating! A feeling of immense euphoria is easily engendered, until you discover that the results are still not right. But how do you know what is 'right'? Well that is easy. You run the serial code on a workstation and compare values for test data, checking that changing the numbers of processors still yields the same results. If there is a result difference the questions are now:

1 Is it small enough to be a hardware effect (it is well established that the same sequence of arithmetic on two different machines, or compilers, or languages or operating systems need not be identical, although the differences would normally be very small, i.e. 0.1×10^{-5} or less).
2 Which result is the 'right' one anyway?

Sorry! We cannot help you much further. However, experience suggests that parallelising code may easily create new bugs, but it may also remove old ones that were previously undetected or occasionally activate old bugs that were never previously encountered. A book of devastatingly 'good' parallel programming bugs would make totally fascinating reading.

Finally, another fearful thought. How do you know that this code is actually correct and yields the right result? The answer is that you do not, but you hope that the *ad hoc* testing regime you may have chosen and inflicted on it has been sufficient to detect, and you have corrected, *all* the bugs. Three further rules of thumb here may be of some comfort:

1 Programs seldom work first time, so if they do be extra careful to check them;
2 Expect to find a small number of major bugs, so look until your eyes grow tired or you find some; and
3 If you find one bug then there could well be others.

Then it is a matter of faith and good luck! 'What's that [we hear you ask]? How does a computational scientist know that his massively parallel program that has just run for six weeks on the world's fastest parallel machine has produced the right results?' The simple answer is that you do not know what the right result is! You could run it again on a different machine to see if it provides similar results, or you use what knowledge you have to check that the results make good sense and/or conform to theoretical expectations, etc. Then you just hope! After all, the task of falsification in science is a matter for others to perform on your work. Later.

7.8 Conclusions

This chapter provides a gentle introduction to the art of message passing. It has done this by providing an extended description of three examples with a line-by-line English account of what the message passing is doing. We are convinced that learning MPI by understanding relatively simple case studies is by far the best way for geographers (and others) to develop their message-passing skills. It is then a fairly easy task to read the MPI documentation to discover exactly what the various options do and what other subroutines could have been used as alternatives. Once you have grasped the essence of these three examples and developed an MPI-oriented mind set then you are ready to do some useful parallel programming and be up to the challenge of the more complicated examples discussed in the following chapters.

Appendix 7.1: ping-pong code

Key: All Processors execute
Master only
Slaves only

```
          PROGRAM PING
#include <mpif.h>
          REAL A,B
          INTEGER MYPE,IER
          INTEGER STAT(MPI_STATUS_SIZE)
C  START MPI

          CALL MPI_INIT(IER)
C  FIND OUT WHAT MY ID IS

          CALL MPI_COMM_RANK(MPI_COMM_WORLD,
      X MYPE,IER)
         IF(MYPE.EQ.0) THEN
            A = 27.0
            CALL MPI_SEND(A,1,MPI_REAL,1,1,
      X          MPI_COMM_WORLD,IER)
          ELSE
            CALL MPI_RECV(B,1,MPI_REAL,0,1,
      X          MPI_COMM_WORLD,STAT,IER)
          ENDIF
          IF(MYPE.EQ.0) THEN
            CALL MPI_RECV(B,1,MPI_REAL,0,1,
      X          MPI_COMM_WORLD,STAT,IER)
          ELSE
            A = SQRT (B)
            CALL MPI_SEND(A,1,MPI_REAL,1,1,
      X          MPI_COMM_WORLD,IER)
          ENDIF
C  FINISH UP AND GO TO PUB
          CALL MPI_FINALIZE(IER)
          STOP
          END
```

Appendix 7.2: summing *M* numbers in parallel

Key: All Processors execute
Master only
Slaves only

```
      PROGRAM SUMS
      IMPLICIT NONE
#include <mpif.h>
      INTEGER MYID,NPES,M,MAXPES
      PARAMETER (M=1000000,MAXPES=256)
      INTEGER SIZE(0: MAXPES),OTAG,STAG,IER,COMM,I,J
      REAL X(M),SUM,TSUM
      INTEGER STAT(MPI_STATUS_SIZE)


      OTAG=1
      STAG=2

      CALL MPI_INIT(IER)
      TIMESTART=MPI_WTIME()
      IF(IER.NE.0) THEN
           WRITE(*,*) 'ERROR IN INITIALISATION'
           STOP 1
      ENDIF
      COMM=MPI_COMM_WORLD
      CALL MPI_COMM_SIZE(COMM,NPES,IER)
      CALL MPI_COMM_RANK(COMM,MYID,IER)
C  *FIRST DO LOOP*
      DO I=0,NPES-1
C     DIVIDE THE PROBLEM UP BETWEEN THE PROCESSOR
      SIZE(I)=M/NPES
      ENDDO
C     *SECOND DO LOOP*
C     ADD THE EXTRA BIT TO EACH PROCESSOR IN TURN
C     REMEMBER THIS IS AN INTEGER CALCULATION
      DO I=0,M-NPES*(M/NPES)-1
           SIZE(I)=SIZE(I)+1
      END DO
      IF(MYID.EQ.0) THEN
C     IF MASTER THEN DISTRIBUTE CHUNKS TO EACH PROCESSOR

C     *THIRD DO LOOP*
           DO I=1,NPES-1
```

```
          DO J=1,SIZE(I)
               X(J)=RANF()*10.0
          ENDDO
          CALL MPI_SEND(X,SIZE (I),MPI_REAL,I,OTAG,
          COMM,IER)
          ENDDO
          DO J=1,SIZE(0)
               X(J)=RANF()*10.0
          ENDDO
     ELSE
  C  SLAVES JUST RECEIVE A CHUNK
     CALL MPI_RECV(X,M,MPI_REAL,0,OTAG,COMM,STAT,IER)
     ENDIF
  C  *FOURTH DO LOOP*
     SUM=0.0
     DO I=1,SIZE(MYID)
          SUM=SUM+X(I)
     END DO

     IF(MYID.EQ.0) THEN
  C  MASTER COLLECTS THE RESULTS AND SUMS THEM
          TSUM=SUM
          DO I=1,NPES-1
     CALL MPI_RECV(SUM,1,MPI_REAL,I,STAG,COMM,STAT,IER)
          TSUM=TSUM+SUM
          ENDDO
          WRITE(*,*) 'SUM IS ',TSUM
     ELSE
C    SLAVES JUST SEND THE RESULT TO THE MASTER
          CALL MPI_SEND(SUM,1,MPI_REAL,0,STAG,COMM,IER)
     ENDIF
     TIMEEND=MPI_WTIME()
     IF(MYID.EQ.0) WRITE(*,*) 'ELAPSED TIME ',
    X    (TIMEEND- TIMESTART)
     CALL MPI_FINALIZE(IER)
     END
     STOP
     END
```

Appendix 7.3: answers to self-test questions

Q1	(c)	8
Q2	(a)	1
Q3	(c)	8
Q4	(a)	0

Q5	(c)	undefined
Q6	(c)	8

Appendix 7.4: spatial interaction model example

Key: All Processors execute
Master only
Slaves only

```
C SI_1.F SPATIAL INTERACTION MODEL
        IMPLICIT NONE
#include <mpif.h>
        INTEGER N
        REAL BETA
        PARAMETER (N=1500, BETA=0.25)
        REAL T(N,N),C(N,N),O(N),D(N),P(N,N),SUM,SS,
     X  RANF,A(N),SST,DL(N)
     1  ,BUFF(N)
        INTEGER I,J,SIZE,MSIZE,K
        INTEGER IERR,COMM,MYPE,NPES,
     X  STATUS(MPI_STATUS_SIZE)
        INTEGER TTAG,DTAG,CTAG,STAG
        DOUBLE PRECISION START,END

        TTAG=1
        DTAG=2
        CTAG=3
        STAG=4

C       SECTION 1- START MPI
        CALL MPI_INIT(IERR)
        IF(IERR.NE.0) THEN
          WRITE(*,*) 'ERROR INITIALISING MPI'
          STOP 1
        ENDIF
        START=MPI_WTIME()
        COMM=MPI_COMM_WORLD
C         FIND OUT HOW MANY PROCESSORS THERE ARE
        CALL MPI_COMM_SIZE(COMM,NPES,IERR)
C         FIND OUT WHAT MY ID IS
        CALL MPI_COMM_RANK(COMM,MYPE,IERR)
C         GENERATE SOME RANDOM DATA
C         START OF MASTER REGION
        IF(MYPE.EQ.0) THEN
```

```
           SS=RANF()
          DO I=1,N
            DO J=1,N
                T(I,J)=RANF()*10.0
                C(I,J)=RANF()*100.0
            ENDDO
          ENDDO
C        END OF MASTER REGION
         ENDIF

C        SECTION 2
C        CALCULATE HOW BIG A CHUNK EVERYONE GETS
         SIZE=N/NPES
C          DISTRIBUTE THE DATA
         IF(MYPE.EQ.0) THEN
C          MASTER GETS THE EXTRA BIT
           MSIZE =(N - SIZE*NPES)+ SIZE
C          MASTER DOES THE SEND
           DO J=2,NPES
             DO I=1,SIZE
                   DO K=1,N
                   BUFF(K)=T((J-2)*SIZE +I,K)
             ENDDO
             CALL MPI_SEND(BUFF,N,MPI_REAL,(J-1),TTAG
     1 ,COMM,IERR)
             ENDDO
           ENDDO
           DO J=2,NPES
             DO I=1,SIZE
                   DO K=1,N
                         BUFF(K)=C((J - 2)*SIZE + I,K)
                   ENDDO
                   CALL MPI_SEND(BUFF,N,MPI_REAL,
     1 (J-1),TTAG,COMM,IERR)
             ENDDO
           ENDDO
           DO I=1,MSIZE
             DO J=1,N
                   T(I,J)=T((NPES - 1)*SIZE + I,J)
                   C(I,J)=C((NPES - 1)*SIZE + I,J)
             ENDDO
           ENDDO
           SIZE=MSIZE
          ELSE
```

```
C         SECTION 3 WORKER - RECEIVE DATA
            DO I=1,SIZE
              CALL MPI_RECV(BUFF,N,MPI_REAL,0,
     1      TTAG,COMM,STATUS,IERR)
              DO K=1,N
                   T(I,K)=BUFF(K)
              ENDDO
            ENDDO
            DO I=1,SIZE
              CALL MPI_RECV(BUFF,N,MPI_REAL,0,
     1        TTAG,COMM,STATUS,IERR)
              DO K=1,N
                   C(I,K)=BUFF(K)
              ENDDO
            ENDDO
          ENDIF

C         SECTION 4 CALCULATE O(I) AND D(J)
              DO I=1,N
            O(I)=0.0
            D(I)=0.0
          ENDDO

          DO J=1,N
            DO I=1,SIZE
              O(I)=O(I)+ T(I,J)
              D(J)=D(J)+ T(I,J)
            ENDDO
          ENDDO

          IF(MYPE.EQ.0) THEN

C         SECTION 5
C        MASTER REGION - COLLECT  D(J) AND REDISTRIBUTE
            DO I=2,NPES
              CALL MPI_RECV(DL,N,MPI_REAL,(I-1),
     1             DTAG,COMM,STATUS,IERR)
              DO J=1,N
                   D(J)=D(J)+DL(J)
              ENDDO
            ENDDO
            DO I=2,NPES
              CALL MPI_SEND(D,N,MPI_REAL,(I-1),DTAG
     1             ,COMM,IERR)

       ENDDO
```

```
           ELSE
             CALL MPI_SEND(D,N,MPI_REAL,0,DTAG,COMM,IERR)
             CALL MPI_RECV(D,N,MPI_REAL,0,DTAG,COMM,
         1     STATUS,IERR)
           ENDIF

C          SECTION 6 CALCULATE MODEL
           SS=0.0
C          THIS IS THE PARALLEL LOOP
           DO I=1,SIZE

C        * CALC A(I)
             SUM=0.0
             DO J=1,N
               SUM=SUM+D(J)*EXP(-BETA*C(I,J))
             ENDDO
             A(I)=1.0/SUM
C        * CALC MODEL
             DO J=1,N
               P(I,J)=A(I)*O(I)*D(J)*EXP(-BETA*C(I,J))
               SS=SS+(P(I,J)-T(I,J))**2
             ENDDO
           ENDDO
           IF(MYPE.EQ.0) THEN
             SST=SS

C          SECTION 7 MASTER COLLECTS ALL THE PARTS OF SS
             DO I=2,NPES
               CALL MPI_RECV(SS,1,MPI_REAL,
         1           (I-1),STAG,COMM,STATUS,IERR)
               SST=SST+SS
             ENDDO
             SUM=N
             SST=SST/(SUM*SUM)
             WRITE(6,123)SST
    123      FORMAT(F15.9)
           ELSE
             CALL MPI_SEND(SS,1,MPI_REAL,0,STAG
         1     ,COMM,IERR)
           ENDIF
           END=MPI_WTIME()
IF (MYPE.EQ.0)
         1   WRITE(*,*) 'ELAPSED TIME ',END-START
           CALL MPI_FINALIZE(IERR)
          END
```

8 Parallelising the geographical analysis machine using MPI

This chapter focuses on the task of recoding the GAM for MPI. It examines the design issues and discusses some of the alternative ways of exploiting the parallelism in the GAM. It does not matter that you do not care what the GAM does or is; it merely provides a good case study for learning more about how MPI is used.

8.1 A data parallel GAM using MPI

The geographical analysis machine (GAM) is very simple to program, as it consists of a series of five nested DO loops. In Chapter 5, it was vectorised, even though at the end of the day the tremendous increase in its performance came from reducing the arithmetic rather than from vectorisation. In Chapter 6, it was converted into a multi-tasking program but not into either a shared DO loop or a data parallel form, because it was judged to be unsuitable for these parallel programming models. Indeed, the GAM is inherently far more suitable for a form of parallel processing that can effectively exploit the two-dimensional independent spatial nature of the underlying search. Historically, the early GAM/1 was run on a vector parallel machine only because there was no practical alternative in the mid-1980s, and the algorithm had to be 'shoe-horned' to become vectorised when its natural parallelism was quite different. In principle, the GAM needs an MIMD parallel-programming model, and this makes it a good example on which to practise your message-passing programming using MPI.

The initial plan is to parallelise GAM using the MPI equivalent of a shared DO loop approach. Later on, the flexibility inherent in MPI is used to demonstrate a better, more efficient and far more general approach to parallel programming. The immediate difficulty here is that while the simplest GAM code is massively data parallel (and highly vectorisable) it is also hopelessly inefficient from a performance point of view. This may be a good time to reread Chapter 5 and recollect how the performance went down from 9 days to about 9 minutes. However, these improvements totally destroyed the nice data parallel nature of the code and because of this it was subsequently declared inappropriate for shared DO loops on a shared-memory machine. Here attention is focused for teaching purposes on the parallel elegance of the original code. Of course, if you have access to a 512-processor parallel computer, and if each processor is about the same speed as a Cray T90 vector processor, then 9 days is reduced to about 30 minutes. Not bad, despite the algorithm being so hopelessly inefficient. However, this is also 'not good', because the original algorithmic inefficiency has been ossified. While it now appears to be a triumph of parallel computing, it is in fact a good example of a problem where with a little more effort about 1000 times more science per hour could have been obtained. The pseudo code for this hopeless inefficient data parallel GAM is as follows:

```
LOOP 0:    DO ITER = 1, MAX IT
           if ITER exceeds 1 then create a random data distribution
LOOP 1:    DO RADIUS = RAD_MIN, RAD_MAX, RAD_INC
LOOP 2:    DO CY = MIN_Y, MAX_Y, RADIUS*0.2
LOOP 3:    DO CX = MIN_X, MAX_X, RADIUS*0.2
LOOP 4:    Compute population and cancer counts for all points (X, Y)
           inside a circle centred at CX, CY with radius RADIUS, write it
           out if there are indications of an excessively high cancer rate.
```

If 1991 census enumeration district data for the UK are used, then this involves the following Fortran code:

```
OBSP=0.0
OBSC=0.0
DO I=1,145716
DIS=(X(I)-CX) ** 2 +(Y(I)-CY) ** 2
IF (DIS.GT.0.0) DIS = SQRT (DIS)
IF (DIS.LT.RADIUS) THEN
      OBSP=OBSP+P(I)
      OBSC=OBSC+C (I)
      ENDIF
ENDDO
```

Please note that this version of the GAM is purely for illustrative purposes. If you wrote this code for HPF then it would run very well on a parallel supercomputer. It may well produce the most amazing levels of parallel performance, but it is still hopelessly inefficient compared with what can be done, as has already been demonstrated in Chapter 5, and much better non-data parallel codes for the GAM are described later on. The value of the present code fragment is to illustrate a data parallel approach using MPI and partly as a warning of the dangers of unthinkingly porting a hopelessly inefficient serial code.

Back to the current code story. In a typical GAM cluster hunt involving data for the UK, Loop 0 would be run only once due to computer time restrictions with this code. Ideally, we would like to run it many times as the purpose would be to test for differences between GAM results for the observed 'real' data and results produced for data generated under a null hypothesis, for instance that the

data are random. The quality of the science partially depends on being able to show that any unusual patterns or data abnormalities detected by the GAM are not purely due to chance occurrence. The principal constraint here is the computer time needed to run the GAM 99 or 499 or 999 or more times. Hence one reason for wishing to parallelise it. The other DO loops reflect the spatial resolution of the spatial search for localised clustering. The assumption here is that circle sizes (and hence pattern) are in the range 1–50 km (Loop 1). The other DO loops reflect the dimensions of the study region (Loops 2 and 3) and the amount of data (Loop 4). In the code being examined here, Loop 1 would be performed fifty times, Loop 2 a maximum of about 6000 times but variable depending on radius size, and Loop 3 a maximum of about 3250 times. As a result, Loop 4 would be executed about 30 million times. In Chapter 5, this code was estimated to take 9 days to run. The question now is: how much better will it do under MPI?

8.2 Where is the parallelism in the GAM?

The first task is to 'find' the parallelism. Remember the different degrees of parallelism that exist; see Chapters 3 and 4. This would result in the following classification:

1 Loop 0 is defined as highly coarsely grained parallelism because each iteration is with a different data set (based on either observed or randomised data) and is therefore completely independent of any other. This is also what you would call 'embarrassingly parallel'. That is, the structure of the program is such that you need do nothing to it to make it parallel. Provided that MAXIT is some integer multiple of the number of processors then it would be highly efficient from a parallel computing point of view, albeit totally inefficient from an algorithmic perspective. However, all you would need to do to obtain maximum algorithmic efficiency as well is to replace the previous code for Loops 1 to 4 by the far more efficient version described in Chapter 5. However, this is too easy, and it is best for current purposes not to pursue this option further, because not all problems are as conveniently structured as this GAM. Also, the number of simulations you might wish to use may not be an integer multiple of the number of processors. Hence if you were using some form of Monte Carlo significance test that needed only 100 iterations then you could end up with a large numbers of idle processors. On the Cray T3D, the nearest power of 2 to 99 is 128, resulting in twenty-nine idle processors. So this very coarsely grained level of parallelism is a little too coarse and clumsy. While you could get away with it, it might be difficult to justify that the quality of the resulting science is so heavily dependent on the precise number of processors you are using.

2 Loop 1 is also embarrassingly parallel but at a less coarse level. The problem now is that this particularly DO loop will probably only ever be repeated fifty times or less on practical applications, which suggests that it would run

best on a machine where the number of processors matched this DO loop count or some integer multiple of it. This is probably only useful on machines with small numbers of processors. Again, you cannot really make your GAM search totally dependent for its parallel performance on the number of processors you are using! Well you could, but it does not look like good science. A much greater degree of application independence and flexibility is needed.

3 Now Loops 2 and 3 are medium-grained parallel. If the GAM were to be parallelised at below the Loop 2 level, then Loop 3 (DO loop CX) would have to be spread over multiple processors. The problem now is that all the data would have to be stored on each processor (probably of little consequence today) but, more seriously, the number of times each of these DO loops is executed is variable because it is dependent on radius size. For example, if MIN_X = 1, MAX_X = 1000, RAD_MIN = 1, RAD_MAX = 10, RAD_INC = 1; see Table 8.1. This causes problems, because the number of processors is fixed prior to the start of the run. So if 256 processors are used, then between 25 and 50 per cent of them would be idle for most of the time. For example, for a radius of 50 km there are only 100 DO loops to farm out, so 156 processors would be idle. For a radius of 1 km, the 256 processors would be busy for fifteen-sixteenths of the time and 152 idle for one-sixteenth. This is clearly better, although these problems are avoided if at all possible, but when using a data parallel or shared DO loop approach it is usually outside your control. Indeed, here is one of the most compelling reasons for learning how to use message passing.

4 Loop 4 is obviously far more finely grained, and this is the best bet for a vector processing, shared DO loop and data parallel approach. The 145,716 iterations of this DO loop could be readily spread across large numbers of processors. This 145,716 reflects the data being used; here it is the total number of 1991 census enumeration districts in the UK. This number is study region and application-dependent. It cannot always be relied on to be as big as this, but equally it could be an order of magnitude larger or smaller.

Table 8.1 Loop 3 counts.

Radius	*Number of times DO loop executed*
1	5000
2	2500
3	1666
4	1250
5	1000
10	500
20	250
50	100

Nevertheless, despite this uncertainty as to the precise size of this DO loop, on a shared-memory system with relatively few processors this would also be a good place to start. However, for a message-passing approach it is far from ideal! It is too finely grained as there is probably insufficient computational work to farm out to large numbers of processors. Much will depend on the speed of the processors, their number, the amount of data being processed and the bandwidth of the interconnection along which the messages have to travel. It is not helpful to have algorithms for parallel machines with a level of performance that is highly hardware-specific if it can be avoided. The reasons are discussed in the next section.

There must be a better way that is not based on sharing out whole DO loops, which creates inflexibility and makes performance application-dependent. Fortunately, there is a better way that avoids these problems, but maybe it is useful to first struggle (a little) with some of the less efficient alternatives. Well, you are supposed to be thinking about how to parallelise the GAM, not just reading about how the authors did it (after a long struggle of their own) in an optimal fashion.

8.3 Loop 4 message passing

The obvious place to start is with Loop 4 because this is where nearly all the computation is concentrated in the original version of the GAM. A message-passing data parallel version of this loop could be implemented as follows. It is assumed that there are NP processors and that each processor has a different approximately 1/NPth share of the data stored in its local memory. These exact values for the processor numbered *MYID* is stored in *SIZE.* We shall ignore how that was done as you should know how to do this by now (see Chapter 7 and the spatial interaction model example). The following pseudo code could then be used:

```
              DO I=1, SIZE (MYID)
              DIS=(X(I)-CX)**2+(Y(I)-CY)**2
              IF (DIS. GT. 0.0) DIS=SQRT (DIS)
              IF (DIS. LT. RADIUS) THEN
                     OBSP=OBSP+P (I)
                     OBSC=OBSC+C (I)
                     ENDIF
              ENDDO
C* Master Processor
              IF (MYID.EQ.0) THEN
                            collect OBSP, OBSC from each
                            processor and add together
                            to obtain a global sum
C* Slave Processors
```

```
ELSE
                    send OBSP, OBSC to master
ENDIF
```

Note that if NP = 100 and N = 145,710 then each processor computes an *OBSC* and *OBSP* value based on its local memory share, with an average of 1,457 cases each. Also, each processor has almost an equal amount of work to do, so load balancing is not a problem. From a computational perspective this is a very efficient solution, provided that the work (i.e. DO loop) assigned to each processor is sufficiently large and the number of processors not too numerous or too fast in relation to the number of cases, otherwise Amdahl's law will take its revenge on you as the message-passing overheads may dominate the run time. The current solution certainly looks good, but all is not really that well at all.

It has been repeatedly argued that a very important design aim in message passing is to minimise the number of messages being passed around because this reduces the amount of useful computation being performed. A processor waiting for a message is to all intents and purposes idle! Now let us calculate how many messages are generated by this pseudo code if Loop 4 is executed 30 million times. If there is only one processor (NP = 1), there are no messages as all the data are stored in the local memory of the master processor. If there are two processors (NP = 2), there are two messages (one send, one receive) for each processor every time Loop 4 is executed. This will generate 30 million messages. If there are 100 processors, there are now 3000 million messages being generated! So this pseudo code has some very interesting properties from a message-passing mega-disaster point of view! The good aspect is that the arithmetic being done on each processor decreases linearly with the number of processors. The disastrous aspect is that the number of messages being generated also increases linearly with the number of processors! This is not what you ever wish to happen. With this code, as the number of processors increases so the wall clock time would initially decrease very rapidly but then would start to increase even more rapidly, until the entire machine was swamped by your messages.

By comparison, a Loop 3 level of parallelism would behave far better. Here there is only one message per complete Loop 4 pass (i.e. 15 million) and more importantly, this quantity is now independent of the number of processors being used. Performance would now scale linearly with the number of processors, reaching a limit set by the overhead of handling 15 million messages, the size of the Loop 4 DO loop, and processor speed. It is clearly far more efficient, despite the problems of variable-sized DO loop counts noted earlier, which could reduce the overall efficiency of the program by some small but variable amount that may perhaps be ignored.

Nevertheless, suppose this loss of performance festers away in your mind. You can no longer sleep at night! You have idle processors, uneven load balancing, 15 million messages, etc.; ghost-like figures of famous computer inventors appear in your dreams to start telling you off; you are wasting time. A parallel greed for speed has gained hold of you. This 'illness' is often only diagnosable retrospectively, when many hours have been spent fiddling around with this or that section of code, or doodling whole new algorithms that may avoid the problem, which by now may have assumed an importance to you that bears no relationship to its real significance. Then before you know it, several weeks have flashed by and maybe your code is not much better than before you started. Ouch! This 'greed for speed' can be painful, or perhaps it is just a problem that geographers may have. But let us continue, as it does seem to be quite prevalent.

8.4 Some alternative message-passing schemes

The Loop 3 parallelism was definitely worth having, but is this the best that can be done here? One possibility would be to switch the order of Loops 2 and 3, keeping the bigger range for the innermost one. This can be done because a basic definition of parallel DO loops is whether or not the order can be changed without changing the logic of the code. It can here, because there are no dependencies to worry about and the order in which the results are generated is irrelevant. While this helps a little, it still leaves the idle processor problem largely unresolved due to the fact that the DO loop counts are probably never going to be exact integer multiples of the numbers of processors. Any ideas?

Let us assume that the 'greed-for-speed' syndrome has a firm hold on you and you are unable to ignore it. So what can be done about it? Well it is easy, or it can be when you have your 'parallel head switched on'! Start by rethinking the problem. Remember the basic design criteria for efficient message passing:

1 no dependency between the amount of message passing and the number of processors;
2 if possible no idle processors;
3 the same numerical result regardless of how many processors are used (the latter is slipped in here as a gentle reminder that the parallel code has to run faster, be scaleable, and still get the 'correct' result; otherwise, it is a very backward step!);
4 wall clock times that diminish as the number of processors increase so that you can achieve more science per hour and keep your sponsors happy; and
5 ideally a code that has been so cunningly designed that it will continue to perform well when either the problem size changes or processor speeds increase (or decrease) by a few orders of magnitude.

One way of doing some of this is to make Loop 4 into a subroutine called CIRCLE; e.g. CALL CIRCLE (CX, CY, RADIUS). This is always a good idea, since by reducing code clutter it leaves the rest for you to concentrate your attention on. Now this subroutine will be computed in parallel by each processor able to run it, and each call currently takes an identical length of time since the amount of computing being done in CIRCLE is constant irrespective of CX, CY and RADIUS (albeit due to a gross inefficiency in the algorithm, which we shall ignore at present). So when there are NP processors, the aim is to run

CIRCLE concurrently on NP processors each with different CX, CY, RADIUS values. This can be done as follows:

```
              L=-1
              DO RADIUS=RAD_MIN, RAD_MAX, RAD_INC
              DO CY=MIN_Y, MAX_Y, RADIUS*0.2
              DO CX=MIN_X, MAX_X, RADIUS*0.2
              L=L+1
              XX (L)=CX
              YY (L)=CY
              RR (L)=RADIUS
              IF (L.EQ. NP-1) THEN
                 CALL CIRCLE (XX(MYID), YY (MYID), RR (MYID))
              L=-1
              ENDIF
              ENDDO
              ENDDO
              ENDDO
C* finish off any residue
              DO I=0, L
              IF (MYID.EQ.I) CALL CIRCLE (XX(I), YY(I), RR(I))
              ENDDO
```

This code would be run on all processors. Note that XX, YY and RR are used to store NP sets of circle values. When there is exactly one for each processor then CIRCLE is called. The only region of inefficiency is now due to idle processors at the end, where there is the possibility of some unprocessed values remaining in XX, YY and RR. Simple but neat! Run this code on a 256-processor T3D and the potential 8 days or so on one processor is now reduced to less than one hour of wall clock time. All the processors will be kept busy nearly all the time. However, this code is still not particularly efficient computationally, as a considerable amount of unnecessary computation is going on, and it was the removal of this arithmetic that helped to speed up the version of the GAM in Chapter 5.

Some further comments are in order. Remember how in SPMD the same program is run on each processor. If there are 256 processors, then all the DO loops not *MYID*-dependent are run concurrently 256 times. There are also 256 identical local copies of XX, YY, RR and L, all computed in parallel. The only section of this code that is different on each processor is the call to the CIRCLE subroutine, where the use of the processor number (*MYID*) in the calling statement ensures that each processor computes something different here (unless you create a nice parallel bug, which results in each processor working on identical data!). Everything else is duplicated. What a waste, you think. Well, maybe not, since (1) it will probably take only a few seconds to do all this duplicated work, and (2) the alternative of doing it on only one processor and then sending out

the XX, YY, RR values creates more message traffic and has all but one of the processors idle while this is being done. With message passing (as with early generations of mainframe computer and current PCs) it is usually far quicker (and hence better from an efficiency point of view) to do more computation and less input/output/message handling. The duplication of effort seen here is merely an unavoidable feature of the SPMD model of parallel programming. You benefit because of its simplicity, and maybe it works 'well enough' anyway so that you need not worry about it.

However, such niggling thoughts often gnaw away at your HPC nerves. How 'well' is 'enough'? Is it worth trying for a factor of 2, 10, 100 or even 1000 times more speed? Should the far faster algorithm described in Chapter 5 be ignored? On this occasion, the 'greed-for-speed' syndrome is a real concern because it most critically affects the amount of science you can do per hour of parallel processor time. How can you ignore the dramatic speeding-up in single-processor performance obtained earlier? Well, that latter issue is easily dealt with; just keep quiet and quote lots of megaflop/s speed rates to demonstrate how well the data parallel GAM now runs and there is a very good chance that no one will ever notice! But really, this is not good enough, even if it may be a common practice in other areas of computational science. Geographers need to demonstrate that they know what they are doing even if some other HPC users may not! There is also a very good reason. Remember Loop 0? The quality of the science may need the GAM to be run many hundreds or thousands of times. Whereas now the single run times are acceptable, think how much better it would be if you do 1000 Loop 0 passes in the time that the current code does it once.

8.5 Doing even better by task farming

It is time to reassess the situation. What have we achieved so far? Well, the current data parallel program does quite well. It has the following properties:

1 it keeps all the processors busy nearly all the time;
2 each subroutine call takes an identical length of time, ensuring excellent load balancing;
3 there is a minimum amount of message passing; and
4 performance appears to scale well with the number of processors.

The principal problem is that purely for educational reasons we have successfully parallelised a version of the GAM that earlier on we proved could be speeded up by a factor of about 300. We could simply plug in the revised method, but the speed of the parallel CIRCLE region would now be dominated by the speed of the slowest circle call among the set of NP parallel calls. This may not matter much with the GAM because of the systematic and hence local nature of the spatial search, but it might occasionally become very bad, particularly at the end of a northing row or when sea changes to land and there is a sudden increase in workload. There is also another problem. The computing time taken by the CIRCLE subroutine is no longer constant. Some large circles take more time than small circles, but the nature of the GAM spatial search will even this out since circle sizes change in a gradual and systematic manner. However, the use of faster processors and/or the revised fast spatial search algorithm will dramatically increase the amount of idle time. The computation performed in CIRCLE will be less than the time taken to send the circle coordinates to a processor. This could be fixed by sending not one but many circle coordinates to each processor. However, this is not a good solution, although it would work 'quite well'. It provides a niggling source of ongoing inefficiency fears. It would be nice if this problem could be avoided. It can, and the answer is task farming.

A task farm is one of the simplest and yet most useful ways of decomposing either data or a computational task for a parallel processor. Trewin (1998) gives a good description, which we elaborate further and attempt to simplify the interpretation. In general, a task farm approach works as before by dividing the processors into master and slaves. The master processor controls the work, and the slaves get on with what they are told to do. However, the master processor is a 'cruel' taskmaster, and as soon as a slave processor reports that it has done its work, it is sent more to do. This is a very unfair society. Those slave processors that do their work quickest are punished by being given more work to do, while others that take longer to do their allotted task may end up being pestered by the master processor far less. The good point is that the master seeks to keep all the slaves working for as much of the time as possible while allowing for either unequal workload allocations (because it is hard to predict in advance) or processors that operate at different speeds, or even processors being given different tasks to perform. The spatial interaction model used a far simpler and equal work share for all forms of task farm due to its data parallel nature.

This sounds easy enough in principle, but how easy is it to implement in practice? Well actually, it is not too bad. The basic plan of all task farms is as follows:

```
1  all start and determine processor number
2  if the processor number is 0 you are the master, so read in the data; then
   master
   (a) send a piece of work to each slave
   (b) wait for a slave to finish and then send it some more work
   (c) when there is no more work, send a quit signal
3  if processor number is not 0, you are a slave
   slave
   (a) wait for task
   (b) check if it is a quit signal
   (c) if not
       • do the work for the task
       • send back a result or a message saying you have finished
   (d) else
       • quit
   (e) wait again
```

And that really is all there is to it.

8.6 A task-farming GAM

It should be noted that the simplest form of task farming sets one processor up as the master processor, so only NP − 1 processors are now available to do the real work. It matters little if there are 256 processors, but it matters much more if there are only four! Here we apply the simplest possible approach. Fortunately, task farming is really quite straightforward and possesses what might be termed an appealing neatness.

To recap, the basic approach is to designate processor 0 (which always exists) as the master and all the other processors (regardless of how many they are) as slaves. The idea is that the master gives each slave a task to perform and then waits for any slave to complete its task and send back a result, at which point the master sends out some more work to that slave. This basic procedure is easily applied to the GAM. Consider the following pseudo code:

```
If master processor then
  (a) compute a search location consisting of X, Y and R
  (b) send out an X, Y, R to each slave processor
  (c) repeat (c) and (d) until end of search
  (d) receive a result back from any slave processor
  (e) send that slave another X, Y, R value unless none left
else you are a slave processor, so expect to
  (a) receive X, Y, R or stop if told to do so
  (b) call CIRCLE
  (c) send result to master
  (d) go back to (a)
endif
```

Appendix 8.1 contains the Fortran code for this approach. The code is explained in plain English as follows.

Section 1 of the code is run by all the processors and declares the variables and sets some constants as well as determining each processor's ID. In Section 2, the master processor reads in the data. In the GAM, all the data are distributed to each processor since the data set sizes are now rarely too large to store on a single processor. These data are copied from the master processor to all of the slaves in Section 3 of the code using a broadcast command. This is a far more efficient way of transporting the data than looping over all the processors and doing the copy one processor at a time. In Section 4, the master initially calculates some ranges and carries out some housekeeping. In Section 4.2, the master starts to work through the available work (the circles loop) and initiates each slave in turn. Once all the slaves have started (when IFLAG = the processor number) the master goes into a receive and waits for any slave to send back a reply (the probability). When a message is received, the master sends the next circle to be processed out to the slave that sent the message and goes back to waiting for the next message. When there are no more circles to be sent out, the master exits the loops, see Section 5, and each slave is sent an X value of −1 to tell it to exit. This is important, because you need to stop each processor when there is no more work to be done, or they will wait indefinitely for data that will never be sent and the entire job will stall. The slave's Fortran code is shown in Section 6: all the slaves do is wait for incoming data, check if it is a quit signal (−1) and then process the data they have been sent by calling CIRCLE or by jumping out of the loop if they have been told to quit.

8.7 Improvements?

The only problem is that there are now four messages per circle, making a total of 30 × 4 million, which is still rather a lot. The question, as ever, is how to improve things? There are a number of possibilities:

1 Reduce the number of messages by a factor of three. You can send X, Y, R as an array in a single message rather than as three separate messages. This is useful, because the time taken to send a message is not constant; there is a fixed cost (or latency) for each message and an extra cost related to its length. Figure 8.1 gives a pictorial illustration. The exact curve is machine-specific. The time taken may also depend on the particular processors being used as well as being dependent on the interconnections and machine architecture. However, there is nothing we can do about this, so let us ignore it and let the computer scientists worry about such things.
2 You could send out more than one circle at a time to each slave. However, this may be tricky and error-prone to program. Nevertheless, there are two ways of doing this.
   (a) Send out chunks of CX to each processor – CY, CX_start, CX_ increment, CX Number – although you would have to be careful at the end of the CX loop. The CX_Number can be varied to accommodate this aspect. If CX_Number is 10 then this would reduce messages passed by that factor (to 4.6 million). An appealing feature of this strategy is that the size of the chunks could be dynamically varied within a GAM run (big for small circles, which are quicker to compute, and smaller for larger ones, which require more computation). It could also be set to best match a particular machine architecture (the code could use circle timing information to ensure a sufficiently large chunk size in order to keep itself working at maximum efficiency). The latter is a little too complex and is probably not necessary here, but it does at least illustrate the flexibility provided by MPI in providing a platform for developing

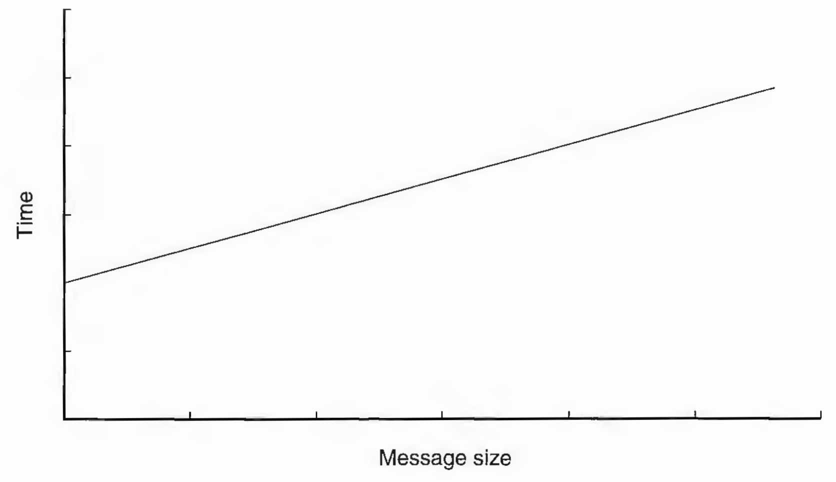

Figure 8.1 Time taken to send a message as a function of message size.

algorithms that dynamically optimise their performance on specific hardware. This is a very nice feature for an algorithm to possess. It is really elegant. It also illustrates the sort of thinking that is important if you are to produce scaleable codes that will work well on future HPC machines.
   (b) An even more radical possibility is to send out a complete CX loop to each processor. This would dramatically reduce the number of messages, probably by a factor of 1000 or so, but it may result in idle processors near the end of the run. Maybe this does not matter.
3 You could consider extending option (1) by having the slaves save up their results before sending them back to the master in a single long message. However, this will not work, because the master only knows when a slave is idle when a result message is received from it. Buffering messages is usually a good idea, but not here.

The problem with many of these suggestions is that the slaves may be able to do their processing faster than new work can be handed out to them. This is unlikely to be a problem with strategy 2(b). However, 2(a) might be better because the parallelism is in smaller chunks.

The optimal answer may depend on how much computational work is performed within each GAM circle. Currently, this involves a statistical significance test which can be either very fast (i.e. a Poisson probability) or very slow (i.e. a bootstrap or Monte Carlo testing procedure). The amount of computational work being performed here would dramatically increase if the GAM were to be extended to handle space–time effects or if GIS coverage permutations were used to partition the data within each circle into homogeneous subsets; see

Openshaw (1998), and Openshaw *et al.* (1999). If the computational workload increases by orders of magnitude then the original method or option (1) would be more than adequate to yield excellent results.

8.8 Loop 0 complications

If Loop 0 enters the picture, then many of these potential work distribution problems will also be resolved. As has been noted, the GAM is embarrassingly parallel at this Loop 0 level. Each iteration is independent of each other. However, if this is executed 1000 times, then whatever task-farming strategy is used, the number of messages increases 1000-fold, and so does total computing time. Yet it may be important to be able to perform Loop 0 runs on some data deemed to be especially critical. Indeed, an early criticism was that the GAM results were a product of random data being subjected to multiple testing. For instance, if you test 23 million circles then at a significance level of 0.05 you would expect (on average) that 23 million × 0.05 (1.15 million) would be significant even in purely random data, although this assumes that the hypotheses are independent of each other (which they are not, because they overlap). Nevertheless, it is sometimes important to be able to dispel such fears by computer simulation, particularly if the application is considered to be hypercritical or so sensitive that you have to be 'sure' that the results are 'right' (e.g. analysis of data for children with a rare disease). You could even argue that the quality of the science in the GAM depends on knowing that the observed patterns are unlikely to be a chance occurrence of the sort likely to be readily found in purely random data. Strategy 2(b) is now extremely viable, because the amount of work to be distributed has greatly increased (by 1000 times), whereas the idle processor period at the end is the same as previously. However, there is an even better possible strategy. If sufficient memory is available, some or all of these Loop 0 simulations could be moved into the work being distributed. This increases the amount of work being done, but by a factor which is less than the number of simulations. For example, the cost of running GAM is essentially the time taken by the CIRCLE routine. If instead of each CIRCLE call producing a single case count it produces ten different values at a time (one for each of ten simulated or real data sets), then the time taken is probably only a factor of two higher, whereas if Loop 0 is left as an outer loop it is a factor of ten greater. Clearly, this strategy is a double winner: more work for each processor and a reduction in total run time due to exploiting the hidden parallelism implicit in the GAM. If this is combined with strategy 2(a) or (b) then the best possible performances would be obtained. However, there are possible problems due to memory restrictions. Each processor would now need to hold a complete set of the observed data, plus 1000 or 10,000 random simulations in local memory. This may not always be possible without some ingenuity being exhibited, *viz.* reducing the storage needs by using data compression and sparse matrix methods, or recompute rather than store the random simulations each time.

8.9 Multiple task farms

There are always other possibilities. Indeed, when do you stop? Most applications can be programmed in several different ways, some more efficient than others. Experience tends to be a useful guide as to the possible 'best', otherwise you end up trying more than one, particularly if there are concerns over performance. So let us examine another approach to Loop 0 using MPI. One such method would be to have multiple sets of masters and slaves, each working on a different set of simulations. On a parallel machine with 512 processors this is often partitioned in various ways, for example 64, 64, 128, 256. Each can have different users running on it. However, you can also do this yourself within a system processor partition. For example, suppose that your GAM program does not scale linearly with the number of processors but peaks at 64. Suppose also that the administrators of the machine encourage you to make use of all 512 processors (since it looks better to outsiders and helps to justify a bigger and more powerful new machine every three years or so). What do you do? The options are (1) run on 512 and ignore the problem and maybe no one will ever know or care; or (2) run on 512 processors but with your GAM split into eight separate sets of 64 processors. You do not gain anything you would not get by running it eight times on 64 processors, but 'by heck' it looks so much better to outsiders. This multiple masters approach is principally of value when performance peaks at a small number of processors.

8.10 More advanced MPI routines

The six basic MPI subroutines have so far served us well, so what about the other 227? Well, you will probably never use most of them. They largely reflect the historical origins of MPI, in which seemingly each member of the MPI forum attempted to retain their favourite features from whatever message-passing system they had previously used! Some are extremely useful and will be used in virtually every message-passing program you will write. This was the justification for the choice of the original six. However, another five basic routines now need to be added, mainly because (1) they reduce the amount of work you have to do; (2) they are probably implemented in a maximally machine-efficient way; and (3) they may reduce the number of errors you could potentially make. Table 8.2 lists them. If you wish to know what they do and their parameters then read MPI manuals. You will need to anyway.

8.11 Conclusions

This chapter has shown you how to develop your knowledge of message passing and apply it to the GAM. The result is a most useful parallel code for a useful spatial analysis tool. The text is verbose because we wanted to explain the thinking and some of the thought processes that lie behind the programming. Most books fail to cover these aspects, but they are very important, especially when

Table 8.2 Five more basic MPI subroutines.

MPI_BCAST
MPI_GATHER
MPI_SCATTER
MPI_REDUCE
MPI_ALL REDUCE

you are trying to teach yourself message passing from scratch. You will probably rediscover them and could add many additions, but by then you will have outgrown any need for this textbook.

Appendix 8.1: GAM example

Key:	all processors execute
	master only
	slaves only

```
C      * SECTION 1
      IMPLICIT NONE
#include <mpif.h>
      INTEGER NCASE
      PARAMETER (NCASE=150 000)
      DOUBLE PRECISION OVERAT,XMINE,XMAXN,XMINN,XMAXE,
     X    OBSP,OBSC,RADIUS,CX,CY,
     X    PROB,RADSQ,
     X    X(NCASE),Y(NCASE),P(NCASE),C(NCASE),RADSQL,
     X    RADINC,RADMAX,RADMIN,POPMIN,CANMIN,THRESH,
     X    OBSPL,OBSCL
      CHARACTER*100 XYDATF,PCDATF,OUTFIL
      INTEGER I,LOOP,ICOL,IROW,IFLAG,TARGET,MYCOMM
      INTEGER IER,NP,MYID,STAT(MPI_STATUS_SIZE),ITAG,
     X    MYSIZE,PSIZE,RTAG
      DOUBLE PRECISION START,END
      INTEGER TOTCAL,TOTDAT,TOTNCS,TOTNHY,STEP,ID,N,
     X    MINN,MAXN,MINE,MAXE,NTIMES,NCALC,NDAT,
     X    NCALS,NHY,
     X    N2
      COMMON IAN/X,Y,P,C,THRESH,POPMIN,CANMIN,NCALC,
     X NHY,NDAT,NCALS
C * START UP MPI
      CALL MPI_INIT(IER)
      MYCOMM=MPI_COMM_WORLD
      CALL MPI_COMM_SIZE(MYCOMM,NP,IER)
      CALL MPI_COMM_RANK(MYCOMM,MYID,IER)
```

```
      START=MPI_WTIME()

      ITAG=1
      RTAG=2
      IF(MYID.EQ.0) THEN
C  SECTION 2
C ONLY THE MASTER NEED DO THIS BIT
        WRITE(6,78001)
 78001  FORMAT('*GEOGRAPHICAL ANALYSIS MACHINE '
     X  ,'GAM/1 (FEB 1997)'//)

C      * STEP 1. READ DATA===============

C      * SET CONSTANTS

C      * READ INI.FILE
      OPEN(UNIT=1,FILE='gamfiles.dat',
     X     FORM='FORMATTED',
     X     STATUS='OLD')

C      * READ USER DATA FILE NAMES
C..    THIS FILE CONTAINS X,Y DATA
        READ(1,10001) XYDATF
C..    THIS FILE CONTAINS POP AT RISK AND COUNT OF REAL
C      CASES
        READ(1,10001) PCDATF
 10001  FORMAT(A)
C      * GET OUTPUT RESULTS FILE NAME
        READ(1,10001) OUTFIL
        CLOSE(UNIT=1,STATUS='KEEP')


C      * READ X-Y DATA
        WRITE(6,6707) XYDATF
 6707   FORMAT('*USER INPUT X,Y FILE IS;',A)
        OPEN(UNIT=1,FILE=XYDATF,
     X     STATUS='OLD',
     X     FORM='FORMATTED')

        DO I=1,NCASE
          X(I)=0.0
          Y(I)=0.0

        ENDDO
        N=0
```

```
          DO I=1,NCASE
            READ(1,*,END=999) ID,X(ID),Y(ID)
            N=I
          ENDDO
 999      WRITE(6,123) N
 123      FORMAT(5X,'*EOF AT CASE NUMBER',I10)
          CLOSE(UNIT=1,STATUS='KEEP')
C       * NO DATA READ?
          IF(N.EQ.0) STOP 1

C       * READ POPULATION AND OBSERVED CANCER DATA
          WRITE(6,6708) PCDATF
 6708     FORMAT('*USER INPUT DATA FILE IS;',A)
          OPEN(UNIT=1,FILE=PCDATF,
     X        STATUS='OLD',
     X        FORM='FORMATTED')

          DO I=1,NCASE
            P(I)=0.0
            C(I)=0.0
          ENDDO
          N2=0
          DO I=1,NCASE
            READ(1,*,END=199) ID,C(ID),P(ID)
            N2=I
          ENDDO
 199      WRITE(6,123) N2
          CLOSE(UNIT=1,STATUS='KEEP')
        ENDIF
C       END OF MASTER BLOCK

C       SECTION 3
C       BROADCAST DATA TO SLAVES
        CALL MPI_BCAST(N,1,MPI_INTEGER,0,
     X MPI_COMM_WORLD,IER)
        CALL MPI_BCAST(N2,1,MPI_INTEGER,0,
     X    MPI_COMM_WORLD,IER)
        CALL MPI_BCAST(X,NCASE,MPI_REAL,0,
     X MPI_COMM_WORLD,IER)
        CALL MPI_BCAST(Y,NCASE,MPI_REAL,0,
     X MPI_COMM_WORLD,IER)
        CALL MPI_BCAST(C,NCASE,MPI_REAL,0,
     X MPI_COMM_WORLD,IER)
        CALL MPI_BCAST(P,NCASE,MPI_REAL,0,
```

```
     X MPI_COMM_WORLD,IER)
C       *NO DATA READ?
        IF(N2.EQ.0) STOP 2
C       * FILES DO NOT MATCH
        IF(N.NE.N2) STOP 3

        MYSIZE=N/NP
        PSIZE=N/NP
        IF(MYID.EQ.NP-1) MYSIZE=MYSIZE+(N-((N/NP)*NP))
C       * GO THRU DATA AND PRODUCE COUNTS
        OBSPL=0.0
        OBSCL=0.0
        DO I=1,N
          OBSCL=OBSCL+ABS(C(I))
          OBSPL=OBSPL+P(I)
        ENDDO
        IF(MYID.EQ.0) THEN
          WRITE(6,8) N,OBSPL,OBSCL
 8        FORMAT(
     X      ' *NUMBER OF INPUT DATA RECORDS: ',I8/
     X      ' *TOTAL POPULATION AT RISK: ',F10.0/
     X      ' *TOTAL CASES   ',F10.0)
        ENDIF
        IF(OBSPL.EQ.0.0.OR.OBSCL.EQ.0.0)STOP 3
        OVERAT=OBSCL/OBSPL

C       * FIND MIN AND MAX X,Y VALUES TO DEFINE SEARCH
C         REGION
        XMINE=999999999.0
        XMINN=999999999.0
        XMAXE=0.0
        XMAXN=0.0

C       * DATA ARE IN 1 KM UNITS
        DO I=1,N
          XMINE=DMIN1(XMINE,X(I))
          XMINN=DMIN1(XMINN,Y(I))
          XMAXE=DMAX1(XMAXE,X(I))
          XMAXN=DMAX1(XMAXN,Y(I))
        ENDDO

        IF(MYID.EQ.0) THEN
C  SECTION 4.1
          WRITE(6,7123) OVERAT,XMINE,XMAXE,XMINN,XMAXN
 7123     FORMAT(
     X      ' *GLOBAL INCIDENCE RATE PER POPULATION '
     X      ,'AT RISK IS ',F12.8/
     X      ' *MINIMUM EASTING IS',F12.1,
     X      ' MAXIMUM IS',F12.1/
     X      ' *MINIMUM NORTHING IS',F12.1,
     X      ' MAXIMUM IS',F12.1)
      ENDIF
      MINN=XMINN-1.0
      MINE=XMINE-1.0
      MAXN=XMAXN+1.0
      MAXE=XMAXE+1.0

C     * STEP 2. SET SEARCH PARAMETERS================

C     * CIRCLE RADII ARE IN KM
      RADMIN=10.0
      RADMAX=10.0
      RADINC=1.0

C     * SELECT PROBABILITY THRESHOLD

      THRESH=0.005

C     * SET MINIMUM CIRCLE SIZE
      POPMIN=100.0
C     * SET MINIMUM CANCER COUNT SIZE
      CANMIN=2.0
      IF(MYID.EQ.0) THEN
C     * WRITE SEARCH PARAMETERS OUT
        WRITE(6,76541) RADMIN,RADMAX,RADINC,POPMIN
76541   FORMAT('*MINIMUM CIRCLE RADIUS IS',
     X    F10.3,' 1 KM'/
     X      '*MAXIMUM CIRCLE RADIUS IS',F10.3,' 1 KM'/
     X      '*CIRCLE INCREMENT SET TO',F10.3,' 1 KM'/
     X      '*MINIMUM POPULATION SIZE IS',F10.0)

        WRITE(6,78234) THRESH
78234   FORMAT(' *SIGNIFICANCE THRESHOLD SET AT',
     X   F12.6)
      ENDIF
C     * OTHER GLOBAL INITS
      TOTCAL=0
      TOTDAT=0
      TOTNCS=0
      TOTNHY=0
```

```
C     * CONVERT ALL POPULATION COUNTS INTO EXPECTED
C      VALUES
      DO I=1,N
        P(I)=P(I)*OVERAT
      ENDDO

C     * RESET MINIMUM VALUE
      POPMIN=POPMIN*OVERAT

C     * SET INITIAL RADIUS FOR CIRCLES
      RADIUS=RADMIN-RADINC

C     * COMPUTE NUMBER OF CIRCLE SIZES TO BE EXAMINED
      NTIMES=(RADMAX-RADMIN)/RADINC+1.0

      IF(MYID.EQ.0) THEN
C SECTION 4.2
C      MASTER BLOCK
        IFLAG=1

C     * OPEN OUTPUT FILE
        OPEN(UNIT=9,FILE=OUTFIL,STATUS='UNKNOWN',
     X      FORM='FORMATTED')

C     * STEP 3. CIRCLE SIZE LOOP==================

C     * *********CIRCLE SIZE LOOP STARTS HERE

      DO LOOP=1,NTIMES
C     * SET CIRCLE RADIUS
        RADIUS=RADIUS+RADINC
        RADSQL=RADIUS*RADIUS
        STEP=RADIUS
        IF(STEP. EQ.0) STEP=1

        NCALC=0
        NDAT=0
        NCALS=0
        NHY=0

C     * STEP 4. GRID SEARCH: NORTHING LOOP ==========

        DO 100 IROW = MINN, MAXN, STEP
          CY=IROW
```

```
C      * STEP 5. GRID SEARCH: EASTING LOOP ============

           DO 200 ICOL = MINE, MAXE, STEP
             CX=ICOL
C      * SKIP IF POPULATION COUNT IS TOO SMALL
             IF(OBSP. LT.POPMIN)GOTO 200
C      * SKIP IF TOO SMALL TO BE OF INTEREST
             IF(OBSC.LT.CANMIN) GOTO 200
             NDAT=NDAT+1
             IF(IFLAG.LT.NP) THEN
C      IF WE ARE STARTING WE SEND TO THE NEXT PROCESSOR
C      IN LINE
               IFLAG=IFLAG+1
               TARGET=IFLAG
               CALL MPI_SEND(CX,1,MPI_REAL,TARGET,ITAG,
     X             MYCOMM,IER)
               CALL MPI_SEND(CY,1,MPI_REAL,TARGET,ITAG,
     X             MYCOMM,IER)
               CALL MPI_SEND(RADSQ,1,MPI_REAL,TARGET,
     X              ITAG,MYCOMM,IER)
               CALL MPI_SEND(OBSP,1,MPI_REAL,TARGET,
     X              ITAG,MYCOMM,IER)
               CALL MPI_SEND(OBSC,1,MPI_REAL,TARGET,
     X              ITAG,MYCOMM,IER)
             ELSE
C      ONCE WE'VE STARTED ALL THE SLAVES WE WAIT FOR AN
C      ANWSER FROM ANY SLAVE
               CALL MPI_RECV(PROB,1,MPI_REAL,
     X                MPI_ANY_SOURCE,RTAG,MYCOMM
     X                ,STAT,IER)
C      NOW WE FIND OUT WHO THE MESSAGE WAS FROM
               TARGET=STAT(MPI_SOURCE)

C      * CALCULATE SIGNIFICANCE LEVEL

               NHY=NHY+1
               IF(PROB.LT.THRESH) NCALS=NCALS+1
C AND THEN SEND OUT THE NEXT CIRCLE TO THAT SLAVE
               CALL MPI_SEND(CX,1,MPI_REAL,TARGET,
     X             ITAG,MYCOMM,IER)
               CALL MPI_SEND(CY,1,MPI_REAL,TARGET,
     X             ITAG,MYCOMM,IER)
               CALL MPI_SEND(RADSQ,1,MPI_REAL,TARGET,
     X             ITAG,MYCOMM,IER)
               CALL MPI_SEND(OBSP,1,MPI_REAL,TARGET,
```

```
     X             ITAG,MYCOMM,IER)
               CALL MPI_SEND(OBSC,1,MPI_REAL,TARGET,
     X             ITAG,MYCOMM,IER)
             ENDIF
C      * END OF EASTING
 200     CONTINUE
C      * END OF NORTHING
 100     CONTINUE


C      ************
C          END OF SEARCH LOOP FOR GIVEN CIRCLE RADIUS
C      *************



           WRITE(6,78221)RADIUS,STEP,NCALC,NDAT,
     X        NHY,NCALS
78221      FORMAT(40(1H-)/' *RADIUS=',F12.2,
     X        'KM WITH STEP OF',I6 ' KM'/
     X        1H ,5X,
     X        '*NUMBER OF SITES GENERATED ',I10/
     X        1H ,5X,
     X        '*NUMBER OF SITES EXAMINED ',I10/
     X        1H ,5X,
     X        '*NUMBER OF HYPOTHESES TESTED ',I10/
     X        1H ,5X,
     X        '*NUMBER OF SIGNIFICANT CIRCLES',I10)

C      * FORM GLOBAL STATS
           TOTCAL=TOTCAL+NCALC
           TOTDAT=TOTDAT+NDAT
           TOTNHY=TOTNHY+NHY
           TOTNCS=TOTNCS+NCALS
C      * GO BACK AND DO ANOTHER CIRCLE SIZE


         ENDDO
C SECTION 5
         DO I=1,NP
C        ALL CIRCLES HAVE BEEN ALLOCATED SO START
C        TO SEND OUT QUIT SIGNALS
           CALL MPI_RECV(PROB,1,MPI_REAL,
     X        MPI_ANY_SOURCE,RTAG,MYCOMM,STAT
     X        ,IER)
         TARGET=STAT(MPI_SOURCE)
```

```
          CALL MPI_SEND(-1.0,1,MPI_REAL,TARGET,
     X        ITAG,MYCOMM,IER)
        ENDDO
C**************************************************
C      * END OF ALL RUNS
C**************************************************
        END=MPI_WTIME()
          WRITE(*,*) 'TIME = ',(END-START)
          WRITE(6,887) TOTCAL,TOTDAT,TOTNHY,TOTNCS
 887      FORMAT(
     X '0********** END OF GAM RUN ************'/
     X      1H ,'*TOTAL SITES GENERATED IS',I10/
     X      1H ,'*TOTAL SITES EXAMINED ',I10/
     X      1H ,'*TOTAL HYPOTHESES TESTED ',I10/
     X      1H ,'*TOTAL SIGNIFICANT CIRCLES ',I10)
C       END OF MASTER BLOCK
        ELSE
C       SLAVE BLOCK
C SECTION 6
 1223      CONTINUE
C       SLAVES JUST WAIT FOR CIRCLES TO BE SENT TO THEM
          CALL MPI_RECV(CX,1,MPI_REAL,0,ITAG
     X       ,MYCOMM,STAT,IER)
          IF(CX.LT.0) GOTO 234
C IF CX IS NEGATIVE THEN IT'S A STOP SIGNAL SO WE EXIT
C THE LOOP
          CALL MPI_RECV(CY,1,MPI_REAL,0,ITAG,
     X      MYCOMM,STAT,IER)
          CALL MPI_RECV(RADSQ,1,MPI_REAL,0,
     X      ITAG,MYCOMM,STAT,IER)
          CALL MPI_RECV(OBSP,1,MPI_REAL,0,
     X      ITAG,MYCOMM,STAT,IER)
          CALL MPI_RECV(OBSC,1,MPI_REAL,0,
     X      ITAG,MYCOMM,STAT,IER)
          CALL CIRCLE(CX,CY,RADSQ,OBSP,OBSC,PROB)
          CALL MPI_SEND(PROB,1,MPI_REAL,0,
     X      RTAG,MYCOMM,IER)
          GOTO 1223
C       END OF SLAVE BLOCK
        ENDIF
 234    CONTINUE
        CALL MPI_FINALIZE(IER)
        STOP
        END
```

9 Optimising performance and debugging hints

This chapter offers practical advice and suggestions as to how you can optimise performance and squash nasty parallel bugs. Both are important and will occupy most of the software development time associated with many HPC applications. Probably the more worrisome aspect is the difficulty of knowing or proving that the results obtained from HPC are reasonably correct.

9.1 Introduction

Converting your code for a parallel processor is just the start of a potentially long drawn-out 'adventure' (some would say 'struggle' or 'war') between man (or woman) and machine (or compiler) in what at times appears to be a ceaseless fight to either get code to work or extract the last factor of two of improvement in wall clock times. It may not be worth the extra effort, but if you really do need parallel hardware in order to obtain maximum performance for your HPC application; if you have a research project that will 'fail' without it, or is being slowed down by extended run times that are either infeasible or so large as to appear embarrassingly ridiculous; if you want to make sure that your code now runs at speeds you would get only on a multi-teraflop machine in five years time; or if you are just fascinated by the thought of making it run *much* faster: this chapter reports some of the authors' experiences and struggles in this area of parallel processing. This quest for HPC speed may involve not just the world's fastest and biggest and most expensive HPC hardware, it could also involve the world's cheapest! One of the practical attractions of message-passing codes (such as MPI) is that you can download (free!) versions that will allow you to create your own virtual parallel supercomputer from networked Unix workstations or even PCs. Indeed, workstation farms are probably the principal way whereby industrial organisations exploit HPC at present. If they can, so can you!

Demel notes that 'to get full speed out of the architecture, one must exploit parallelism, pipelining, and locality. These are ubiquitous issues at all levels of parallel computing . . . It is a challenge to juggle all three simultaneously and get good performance' (p.1 lecture 3, CS 267, 1996). The challenge also involves the reorganisation of algorithms to enhance all three. The danger is that too much attention is paid to optimisations that are unique to particular hardware and hence are not generic. These are issues even with serial computers, as it is good advice for any RISC machine as well as parallel ones. So here is a golden rule of thumb: maybe you should always assume that the parallel HPC hardware that you are using today will be gone or different (faster and bigger) tomorrow; but your code and algorithms need not be so short-lived. Significant codes have long lives, and their underlying algorithms may last even longer! The most enduring performance optimisations are therefore those 'written' into the design of parallel algorithms and are not a particular time- stamped historical bit of completely machine-specific code optimisation. On the other hand, it is important not to ignore any local changes that help, particularly if the 'equivalent' exists on other machines.

9.2 First optimise your algorithm rather than fiddling with code

As Chapter 5 demonstrated so well, the best way of optimising performance so that code runs faster is to optimise the performance of the algorithm by rethinking it to minimise the amount of arithmetic performed and to localise memory reads and writes. Each time you assign a value to a variable there is a memory read and a write (or at least a register-to-memory read/write operation). Imagine that your memory is equivalent to reading and writing a randomly organised disk file and you will readily appreciate the potential performance issues. Over the last decade and for the next decade (or two), arithmetic speeds are increasing (and have increased) many times faster than memory reads and writes. Getting or putting values from or to memory is now almost as bad (relatively speaking) as reading or writing disk files was 10–20 years ago. Memory, even local memory, is now a slow storage device compared with the speed of computation and most other operations. So redesigning algorithms to handle fast modern microprocessors is a really good idea. By comparison, rearranging DO loops and cache fiddling offer fairly trivial gains.

There is a danger of what can be termed 'algorithmic complacency'. This line of reasoning goes as follows: 'Well it worked on supercomputer X. It was after all designed for that purpose ten years ago. So let's port it 'as is' on to latest machine Z'. However, this can be disastrous! The GAM case study offers a useful lesson here. The original problem was put into a vector form. Maximum use was made of local subroutines optimised for the hardware, and historically acceptably good levels of performance were achieved. However, this was in the late 1980s! The same code was much later ported on to the Cray J90 and worked at about twice the previous performance level. The code was also ported to the Cray T3D, and it also worked fairly well on up to 64 processors, which basically gave it a slight speed-up compared with the Cray J90. But all was not well! The problems were as follows:

1 the algorithm did not scale beyond a modest number of processors;
2 large parts of the original code are no longer needed (this related to the use of data compression necessary to fit the data on to 1 megaword of memory on now extinct hardware);
3 the original algorithm was highly complex and efficient, although some of the efficiency was lost due to the need for additional code to enhance vectorisable operations; and
4 the algorithm needed rethinking in the HPC environment of now rather than then (when it was probably the best that could be managed).

The results shown in Chapter 5 illustrate what an algorithmic rethink can achieve. The lesson here is that porting old code largely unchanged on to parallel machines is a recipe for potential disaster, albeit a largely invisible one. For instance, would anyone have known that the original GAM was now so hopelessly inefficient? No, sadly they would not, except that the quality of the science that could have been performed using it would have been much reduced. Despite this latent inefficiency, the scientific case made to support this code now would be as strong (or even stronger) than it was then! The case would be stronger because of the wider proliferation of important spatial data that needs to be analysed! The code could be altered to improve its scaling performance but leaving its algorithmic deficiencies largely untouched. We wonder how many other codes successfully ported on to parallel hardware are similar to the original GAM.

If you are naive, it all appears ever so easy! You move your code to a multiprocessor machine. You emit a big sigh of relief when it works after two minor changes. You plug in locally optimised subroutines for linear algebra, etc. It goes faster. Wow! A good time to go to the pub and celebrate. But what have you really achieved? If the machine is 100 times faster than previously, did you get a 100 times speed-up? Of course not; you never do. But did you attain 50 per cent? No! What about 30 per cent? No? Well, what about 10 per cent? Yes! Well done. Your port was certainly not worth the effort! If you do not believe this then try the following simple test. Take your serial code and run it on a multi-processor box, selecting the appropriate parallel compiler options. Now run it. 'Oh! It took longer on four processors than on one! Well, let's not bother! This parallel stuff is not worth while.' However, you merely need to think about redesigning your algorithm to enhance its parallel content. It has been emphasised that it is you and not the compiler who will succeed best at this task, because it requires an algorithmic rethink rather than code optimisation.

Some other rules of thumb apply here: the larger the code, the harder it is to change; the older the code the harder it is to change, because the original authors are either dead or retired; the degree of commenting in the code is probably not too critical beyond a certain minimum point. Perhaps all code not explicitly written for parallel hardware should have a 'best before' date stamped on it. Maybe the new millennium should have been a good code self-destructs threshold!

In case you are wondering, how an algorithm is optimised is seldom straightforward. Basically, it is a process of trial and error, relentless experimentation, lateral thinking, and even good luck. Oh yes, and common sense also helps.

9.3 Now start to fiddle

If you cannot improve your algorithm then start to fiddle with the code. Quite often, new algorithms emerge from code fiddling; your brain just sort of comes up with it! The following steps are sometimes useful:

Step 1: Benchmark the original code on a suitably representative but conveniently sized problem.

Step 2: Profile the code to detect hot spots: these are the parts of your program where most time is spent. There is no need to optimise a subroutine which takes, for example, 1 per cent of the total run time; concentrate on the largest parts. If a subroutine is called millions of times, even a small saving can be very important. Use your knowledge of what the algorithm does to improve its performance, but without destroying its generality. For example, if you know that a function will be computed *K* million times, then can you update it rather than recompute it? This strategy destroys megaflop rates but can speed up an application by thousands of times. It is easier not to do this, but maybe the performance gains can more than justify it.

Step 3: Try to pull the 'hot spot' out of the program so that you can change it and test it quickly. Then experiment with different codes to see what goes fastest.

Step 4: Put the new code back into the main program and check that you still get the same answer for your test problem.

Step 5: Repeat from Step 2 until no further improvement is possible or necessary, or you have run out of obvious ideas. There are still many tricks to investigate, including merging DO loops, localising memory access to remove random addressing for data used more than once, and rereading Chapter 5 about pipeline stalls and what to do about it. Many compilers offer DO loop unrolling, so investigate it. It does not always help.

9.4 Scaleable performance

Scaleability is the great hope of parallel programming. Scaleability means that as you add more processors to your machine so its performance (i.e. the wall clock or elapsed time taken) on your code decreases by a corresponding degree. Doubling the number of processors may halve the total run time, or that is the hope. In practice, speeding-up will be less than perfect, for reasons discussed earlier when Amdahl's law was considered.

Linear scaleability is the ideal case. However, performance is often best expressed in terms of some application-specific measure of the amount of useful work being performed, not merely a count of the amount of arithmetic being done each second. Indeed, be careful not to seek to mislead others or be misled yourself. The best measure of performance is not a count of Mflop/s but the amount of useful work (or 'science') being performed. The reason is simply that

the two need not be related; in fact, there could be an inverse relationship. For example, consider a computer model that is going to be evaluated 10,000 times. Do you:

1. change the model to maximise the Mflop/s attained by the processors; or
2. change the model to maximise the number of model evaluations performed per hour of wall clock time.

If (1), then you can boost Mflop/s by altering the code to do more arithmetic, *viz.* make use of BLAS routines, remove sparsity and unroll large loops. This will often optimise Mflop/s, produce scaleable performances and look exceedingly good when viewed from outside. If (2), you seek to speed up the code by reducing the amount of arithmetic being performed, exploiting sparsity, using storage to avoid recomputing functions, and perhaps switching to 32-bit mathematical functions. The more successful you are, the lower the number of Mflop/s per second because times are now dominated by the serial part; performance may no longer scale at the same level of parallel granularity as previously, but you may achieve two or three orders of magnitude more model evaluations per hour! More science for less cost but with seemingly far poorer Mflop/s characteristics. Option (2) is what you should aim for, but in many cases option (1) is all you get!

Some applications when expressed in a parallel form achieve superlinear speed-up. Superlinear scaleability is defined as an improvement in 'performance' as the number of processors increase that is better than linear. This may seem to be impossible, but it can happen for various reasons: for example, a sudden reduction in remote memory reads due to more processors, resulting in a better local distribution of the data; threshold effects due to caching, pipelining, and other architecture specific peculiarities (*viz.* a reduction in paging); or even an algorithm that becomes more efficient as the numbers of processors increase. The latter may appear miraculous and it is to some extent but it is not impossible! Certainly, some methods, such as genetic algorithms, can perform better when more processors allow larger population sizes to be used, and this may improve the results. So although more work is being done during each iteration, the total work being performed overall may actually be reduced because the best solution is found more quickly! Another benefit occurs when more processors permit more work to be performed and thus provide improved results or higher-quality solutions. Hence the superlinearity relates to the quality of the results (another measure of performance) and reflects the availability of more processor time. Faster turn-around can itself yield significant 'extra' benefits in a real-time modelling or spatial analysis context.

Far more common is the opposite effect. Performance scales for a while and then deteriorates dramatically. This can be caused by several factors, including an increase in communication overhead as the amount of work assigned to each processor is reduced or processor speed increases. Maybe the answer is to try to create algorithms that have an adaptive component to them so that they can sense the time taken to perform a subtask and then distribute the load accordingly. MPI gives you the flexibility to do this if you wish, but you have to be clever enough to build auto-adaptation into your algorithms. It cannot do it for you. There is no magic MPI subroutine we have not told you about!

9.5 Exploit Amdahl's law

Morse (1994: p. 239) suggests two very useful rules of thumb.

> **Rule 1:** Efficiency decreases with machine size if the problem size is held constant and more processors are used, which is a re-expression of Amdahl's law. So avoid the problem by increasing the problem size either by using more data or by increasing the computing load (i.e. by switching to more robust and less assumption-dependent computationally more intensive alternatives that yield better science). For example, instead of conventional non-linear optimisation methods, which can get stuck in local non-convexities, use genetic algorithm-based alternatives, which are better (in the quality sense) but need 100 times more computing power.

The purpose of HPC is not just a faster results delivery service but also to yield better science. Better science may be obtained by a variety of routes but mainly via a more timely result and by a better-quality result. In a geocomputational context, quality may be partly a function of data resolution and hence quantity, the pedigree of whatever is being computed, robustness or optimality of results, better knowledge about the result (*viz.* confidence intervals, sensitivity analysis and error propagation properties), and more data due to ease or convenience of HPC applications in a GIS data-rich age.

> **Rule 2:** Efficiency increases with problem size if the number of processors is held constant. This feature has been observed in many real-world problems and is seemingly related to the differences in growth rates of the parallel and sequential parts of the problem. This is also something that algorithm designers can influence.

In essence, there is no escaping the basic proposition that large parallel machines need large problems, where large in the first instance relates to the number of processors and in the second instance to the computational content of the application. You can 'exploit' parallel HPC by 'growing' the computational content and the quality of the science at a rate that reflects increases in machine speeds. This is not a new idea but a useful fact of life that geographers could usefully exploit.

9.6 Some MPI optimisation secrets

9.6.1 Limit the number of messages

Messages are essential to parallel programs so that processors have the correct data and can report results for their calculations. However, messages are 'expensive', that is they take a long time relative to arithmetic calculations, which are what you actually want a parallel machine to be doing. So remember that when a processor is communicating or waiting to receive or waiting to send a message, it is in fact idle. The megaflop rate is 0. Nothing useful is happening from a computational point of view on that processor. So all the time that M processors spend speaking to each other is idle or wasted time. You clearly need to reduce this chattering as much as possible.

A key part of the optimisation process is limiting the number and size of messages. A big message obviously takes longer than a small message, but there is a fixed cost to all messages and it can often be better to send a single large message than lots of small messages, even after the extra work of building a buffer and packing the data into it. Another unnecessary cause of messages is data distribution and reduction. As we saw in both the GAM and the spatial interaction model case studies much of the work involves sending out data and collecting results. There are some obvious and simple methods of doing this, but they may not be the best way of doing it, for example if we need to sum a series of numbers and distribute the result to all processors. The 'obvious' way to do this is to have all processors send their results to the master to do the summation and then send out the result. This needs $2(N - 1)$ messages. Another method is to imagine the processors being in a ring and passing their local results clockwise and summing their result with the one received and then passing on that result. This leads to $(N - 1)^2$ messages but less local contention. 'Contention' is the term used to describe the 'traffic jam' that occurs when a processor tries to send or receive a large number of messages at the same time. Usually, sending a message requires a large number of operations such as copying the data to a buffer, looking up the address of the receiver and negotiating with the receiver. This all takes time, and even with asynchronous messages nothing else can be done while this happening. In the first case, processor 0 receives $N - 1$ messages and sends $N - 1$ messages. Since it can only send a single message at a time, this obviously takes quite some time. In the second case there are more messages, but they are spread more evenly between the processors, which may be faster.

As a programmer, you may need to know what sort of processor interconnection network you have, what the time costs of messages are and how your system handles message contention before you can decide which is the best method to use, and this is for the simple case of summing a set of numbers. But do not despair! The MPI standard provides a number of high-level data distribution and reduction methods. So whoever developed MPI for your system will know (or have discovered) the key parameters for your system and will (hopefully) have implemented the best method for you. So all you need to do is call

MPI_REDUCE and it will be the optimal solution for your current system, but when you move to a new machine your code will still be optimal with no changes on your part, because the version of MPI_REDUCE for that new machine will also have been optimised.

A final thought. In case you are wondering how to count messages, there are two alternatives. Some software tools will do this for you and map out the flows visually. Alternatively, you can add counts to the processor-specific code and then report the totals at the end of the run.

9.6.2 Data locality

Messages are expensive, so if you can avoid passing bits of data from processor to processor you should. If you have enough memory space, you can store the whole data set on each processor and just pass an index to a processor that tells it which part of the data to work on. If that is not possible, then try to determine what is the best way to lay out the data between the processors. As we saw in the spatial interaction model case study, if you choose the wrong method the number of messages required can be much higher. Try to draw a picture of the data layouts at each step of your algorithms: watch for reduction and distribution operations especially carefully.

Profiling can help here. You may notice that a single message call is inside the main loop but could be moved outside it, or replaced with some replicated calculation on each processor. For instance, it may be necessary to calculate how much data each processor has. This can be done on the master processor and distributed to each processor, or each processor can be sent the size of the total data and then calculate for itself what its share is. Look for opportunities to use local storage rather than message passing, do more computation instead of message passing, and avoid creating code where the number of messages being passed is an exponential function of the number of processors being used.

You need to look for opportunities to improve data locality. Consider the following code fragment:

```
DO   I=1, 99999
. . .
. . .
DO   K=1, N
X (K)=Y (K)/Z (INDEX(K))+X (K)
ENDDO
. . .
. . .
ENDDO
```

Far better is

```
DO K=1, N
ZZ (K)=Z (INDEX (K))
```

```
ENDDO
DO   I=1, 99999
. . .
. . .
DO K=1, N
X(K)=Y(K)/ZZ (K)+X(K)
ENDDO
. . .
. . .
ENDDO
```

because ZZ is a serial version of randomly accessed array Z. Do it once, so that subsequent accesses are sequential in memory. Remember that on modern hardware memory access is far slower than computation, and if you can access memory sequentially then you will gain major performance benefits from caching. That is, the processor will often read ahead and store the data in very fast memory called a cache. If the next memory access is to this cache, then it will be retrieved much more quickly than if the access is random, when the memory cache will have no effect.

9.6.3 Synchronous versus asynchronous messages

As we have already discussed, MPI provides two different types of message: synchronous and asynchronous. A synchronous message pair is like a telephone call; both parties must be involved at the same time, and they must both finish communicating at the same time. An asynchronous message is like an e-mail message: the sender can carry on working after the message has been sent, and the recipient can collect the message whenever it is convenient to do so. An important speed improvement can be obtained by 'hiding' the time taken for a message to complete by using an asynchronous message and continuing with a calculation while it completes. However, there are a few problems that must be considered. If the slaves are producing large numbers of messages, it is possible for them to overwhelm the master. So you should always take care to check that the previous message has been received before sending the next one. While this will slow down your program in some cases, it will also prevent it crashing!

However, in some circumstances asynchronous messaging can be a very useful technique. For example, in genetic programming it is common to have individual processors or groups of processors working on a separate population. When a processor finds a new best solution it is useful to store this result, so it must be passed to the master processor. But there is no need for the evaluation of the population to halt while this is carried out, so an asynchronous send is the best message type to use.

9.7 Debugging parallel code is harder than serial code

Morse (1994: p. 200) modestly notes that: 'There are a number of coding errors unique to parallel machines, whose detection and resolution may require special debugging services'. He also adds that the answer to the question 'is parallel code more difficult to debug than serial code?' is yes in nearly all cases. The relative simplicity of handling a single instruction stream by either a debugger or print statements becomes at least a factor of *M* times harder when *M* different instructions streams are being executed concurrently, any or all of which you can print or be symbolically debugged. Some of the problems are:

1 Compiler bugs are far more common than are now typical in serial environments, where the software is very mature and almost completely bug-free.
2 Parallel synchronisation faults are hard to spot, because they may occur only sporadically or be dependent on other activities going on in the hardware that are unrelated to your code.
3 Wrong results can easily occur but without any indication of any errors being detected, due to errors in the parallel logic (see Chapters 3 and 4 for suggestions).
4 The order in which operations are executed is often non-deterministic and can differ from one run to the next, so errors may be difficult to reproduce or may be different on different machines or the same machine with different numbers of processors.
5 The problems caused by logic errors may become visible only when a particular number of processors is used and are thus difficult to recreate.
6 There is a factor of *M* times increase in potential debugging output with *M* processors running in parallel, so start by being careful.
7 A high frequency of transient errors that are problem-data-and problem-size-specific.
8 Errors are often highly non-local in that an error on processor 1 is sent to processor 2 and then onwards to processor 17, etc. before it is detected, which makes debugging very difficult.
9 Often the programmer's intuition, combined with prolific use of selective print statements and good luck, is the only hope of spotting complex bugs; but this is hit or miss and can take much effort and time to be successful.
10 Interactions can occur between the debugger and compiler optimisations, as the optimising compiler often restructures code, which destroys the connection with the code that the debugger was run on, so there may be little point in stepping through parallel code to find a bug because in the final optimised version the sequence of operations may be different.

Despite these problems, debugging is also often a source of considerable intellectual excitement and satisfaction when an elusive bug is finally obliterated. On a rainy, dull day, there is sometimes little more thrilling than spotting and then killing a particularly subtle parallel HPC bug.

9.8 Debugging message passing

The art of debugging parallel code is to keep the debug output to a minimum and then seek to monitor the local behaviour of a specific processor in order to

develop an understanding of the problem. You need to be able to spot patterns, look for clues and in the manner of a crime detective identify the suspects or villainous bits of code before dispatching them to oblivion.

A possible strategy for bug detection is as follows:

1 Identify that there is a problem, even if the program finishes normally. It is much easier to know you have a problem if the program terminates abnormally. Clues to normal completion but totally 'wrong' results are:
   - they do not make sense (but this may mean that your expectations are at fault);
   - they fail logical consistency checks that you have put into the code hoping they are never triggered (but you may often have bugs in these checks);
   - the run is completed far too rapidly (this is always a cause for suspicion);
   - you obtain very different results each time you run it or if the number of processors is changed (this is extremely worrying);
   - you obtain a different result on your workstation with a single processor; and
   - you obtain different results if compiler optimisation is turned off or changed.
2 Save and check intermediate results later.
3 Run on fake data for which the 'correct result' is known either accurately or approximately. (If you use random data what would you expect to get? If you manufacture synthetic data with known properties do the results resemble prior expectations?)
4 Check that you obtain the same result with one processor as you do with many (although be aware that data set size restrictions may result in the test data being too small to be realistic or to trigger subtle bugs).
5 Try modifying all array sizes to minimum data set specific values to see if this triggers something that is detectable.
6 Trap or detect (if possible) all operation exceptions (e.g. arithmetic underflows, overflows, NaNs etc.).
7 Turn all debugging, subscript checking and cross-referencing options on, and beware of any compiler (and language) that cannot detect invalid subscripts.

This is the major problem with C and maybe also in some versions of Fortran 90, as attempting to access invalid array addresses is probably the most common error you can make. The statement

```
REAL X(100)
X(101)=57.6
```

is a bug not a feature, no matter how hard C programmers may try to convince you that it does not matter. It does! Subscript bugs are the most common programming error you are likely to make. Compilers that possess no means of subscript checking are doing no one a service and could even one day be responsible for the next world war! Also, look at the compiler warning messages. Make all declarations explicit (in Fortran use IMPLICIT NONE). If possible, flag uninitialised variables: just because X is set on processor 1 does not mean it will be set on processors 2 to *N*. Avoid COMMON storage in Fortran.

8 Run on a single node to determine whether the error is parallel in nature.
9 Look at the code and try to identify the cause of the problem; it will often be self-evident. You do not need a symbolic debugger if you really need to think about the code. Sketch out the DO loops, check the logic of the algorithm by rewriting it from the code, and add more comments. Try explaining it to a colleague. If all that fails, then use a debugger as the last resort rather than a first resort.
10 Look at the logical structure of the code and try to imagine what the algorithm is doing, initially around where you think the problem lurks locally and then more globally.
11 Only if you still have a problem resort to using a debugger. One of the authors claims that he has never had much need to do this; the other would argue the opposite in that he never tests a program that fails without it. The alternative is to add select IF statements in order to test the functioning of the program.

A large number of trivial problems that can cause serious bugs are probably due to some of the following:

- use of a misspelled variable name that is valid
- wandering off the end of an array
- misplaced ENDIF statements that altered the logic of the code
- incorrectly programming the algorithm in the first place
- errors in arguments to subroutine calls, although many compilers will now detect this for you.

Some basic rules of thumb are as follows:

1 bugs are seldom due to the compiler not working properly;
2 bugs are even less likely to be due to a hardware fault that results in erroneous results;
3 nearly all bugs are due to logic errors in the coding of an algorithm, or probably more likely due to errors in the intended functioning or design of the algorithm;
4 because you failed to anticipate 'bad' data or unexpected data-dependent conditions arising during execution of the program. Never rely on the end-user of your program obeying the rules. You simply cannot trust users, so do not write code that does!

If you have access to good parallel debuggers and performance visualisers then use them, but their use will by themselves not necessarily solve your problem. You need to think about the problem using whatever evidence exists to localise and then trace the source. Quite often, the debugger is only useful in helping you to understand a bug that you have had to find by other means. The PRINT or WRITE statement is still a most useful device. Indeed, we suspect that it is still used to debug many parallel codes.

9.9 Defensive coding

The most difficult bug to find is probably the one that depends on the order in which events occur. Most of the time there is no problem as messages are generated in the expected sequence, but suddenly the sequence changes (perhaps due to external factors such as a change in load on the HPC) and an error appears. Alternatively, there is a logic error that does not matter most of the time but which is triggered when the size of application changes, a new version of a compiler is installed or unanticipated data values occur. It is potentially very worrying when you hear people talking about codes that work on 64 or 256 processors but not 512 or 128, or require use of the previous version of the compiler, or even require that all optimisation is turned off so that it will run! Such situations, it seems, are quite common but are almost certainly indicative of bugs. Bug-ridden codes cannot be trusted to yield good science, or even safe science. So beware! Quirky codes are really bug-ridden codes, except that the bugs are alive and well and you are the intended victims. You can have no great confidence in such codes or the results they provide. Sorry! Computers are not to be trusted. Codes that contain bugs cannot be trusted; nor should they be. Codes that contain no visible bugs may still be untrustworthy! The onus is on the user to demonstrate unequivocally that the results are 'correct'. Formal proofs of software correctness would be nice but do not exist for most applications. HPC applications in geography and GIS are not mission critical, but this does not mean that the results do not matter. They do!

One solution is to reprogram the same algorithm by five independent researchers, but what if the error is in the algorithm? Another is to test the code on many different applications by many different users, asking them to individually and independently validate the results (the beta testing approach). Maybe all you can do is to perform all obvious and reasonable checking (necessary to avoid potential litigation, if relevant) and then create a mechanism for users to report bugs or potential bugs.

A good way of avoiding some of the problems is to build logical checking into the code. Examples are given of this in Chapter 7. For example, it is always a good guide to try to validate the address of the sender or the message number and build this logic checking into the code if you can. If you can establish consistency checks that can be applied to the results (i.e. they should sum to a known value), then do so. Ideally, you want codes to self-test themselves if at all possible as often as possible.

9.10 Shared-memory debugging

The global memory of this hardware makes debugging easier than with distributed-memory systems. However, the hardest bugs occur in shared or global memory being written into by multiple processors or multiple subroutines. The Fortran COMMON statement can cause much mayhem; for example, these statements are acceptable Fortran:

```
      COMMON FRED, N, M, X(2)
then in SUB1
      COMMON FRED, X(2), N, M
      X (1)=27.0

then in SUB2
      COMMON FRED, N, M, X(2)
      DO N=1, M
```

but they can create a very difficult to-detect-bug. Here the layout of the data in the COMMON block was different and because of this quite different, values come out of it. Also, the identifier N has been used as the index in a DO loop, with possibly devastating consequences the next time access is made to the value stored in N. The debugger will tell you that N has the wrong value, but the error was caused somewhere else that may be difficult to trace. Much of this advice also applies to debugging non-HPC codes.

Perhaps more common are synchronisation bugs where a critical variable is updated simultaneously by multiple processors due to a missing statement. The results produced are wrong, but the program does not fail. Somehow, *you* have to be clever enough to spot that the results are incorrect. So build in as many self-checking logic tests as you can manage and leave them in your code.

9.11 Message-passing debugging

There are some very common errors, which can be listed as follows.

1 **Message pairs do not complete.** Processor 0 expects *N* messages when in fact with *N* processors it must allow for itself and only needs to receive $N-1$ messages. In this case processor, 0 will wait until the program is killed since it will never receive the last message, or it will grab the first message of the next batch but do the wrong thing with it.

2 **Messages are the wrong size.** If processor 0 sends processor 1 a message of twenty numbers when processor 1 is expecting seven numbers then the remainder of the program may fail horribly (if you are lucky), or it may just carry on but be wrong and maybe you will never be any the wiser!

3 **Not passing key parameters out to other processors.** If processor 0 reads in the size of the data set (N) from a file and then uses this to calculate how much data to send out to the other processors, you must send N to all the other processors so that they can try to receive their N/M share elements of data, but on many machines N will be 0 or undefined.

4 **Not telling the slaves to stop.** In many MPI programs, the processors are split into a master and the remainder are slaves. In general, slaves consist of a simple loop that receives an instruction from the master, executes some calculations and sends back a result. It is vital to have a mechanism for the master to send a slave a 'quit' signal so that it knows there is no more work and that it can exit or proceed to the next task. In this situation, it is also important for the master to keep track of how many slaves it has and how many have been informed that it is time to stop. The master cannot stop until the last slave has also been told to stop work.

5 **Deadlocking.** A deadlock can occur if two processors attempt to send synchronous messages to each other at the same time. Unfortunately, neither processor can now proceed until the other has received the message but neither can receive it until the send has been completed. This can be avoided by using asynchronous sends or by restructuring the code on one of the processors.

Deadlocking is easy to spot in the two-processor case, but it can be harder to see with more processors. For instance, if it is necessary to sum a series of numbers held on different processors, one possible algorithm is to pass a number to the next processor and add that to your number and then pass it on. If all the processors use a synchronous send, then all four will deadlock. One way to spot this is to draw a series of time lines with messages as arrows connecting the lines. It would then be immediately clear that the processors cannot reach the receives. A possible solution is to have even-numbered processors send and odd-numbered processors receive first, then *vice versa*.

6 **Data division.** Many MPI codes will divide the data between the processors. If care is not taken, a number of errors can creep in at this point. A common error is to miscalculate the start point of the blocks of data and either miss out some data values or send the same data values to two different processors. Another difficulty is handling the master processor. Since it holds all the data it does not need to send this portion, but it does need to copy it to the same place as the other processors. It is also vital to calculate the sizes of the blocks correctly. In very few cases will the amount of data divide evenly by the number of processors, so take extra care as to what happens to the remainder. If you do not, then you may never spot the loss of data that may occur as the number of processors is changed. The solution is to try to program the data division code correctly and then, in case you get it wrong (well it was Friday afternoon!), check that the number of cases tallies. Likewise, some global data check sums can be helpful; e.g. add up all the values of a variable when read in and check it later against the accumulated sums returned from each of the separate processors. They should match (apart from rounding error).

9.12 Conclusions

This chapter has concentrated on the boring part of parallel programming an HPC. Debugging is obvious but is often hard work. Error checking and results verification are also most important but are often taken for granted. Scientific papers are subject to extensive and rigorous peer review, but their computer code is never examined or independently evaluated or reviewed.

The absence of error indications may mean any of the following:

1 the answers are correct
2 the bugs were never found
3 the algorithm was correctly coded but was wrong
4 the code contained undetected bugs but was able to cope with them
5 massive undetected numerical rounding inaccuracies occurred, but no one noticed
6 the code and algorithm were correct but the data were wrong
7 all run-time error flags were turned off to improve performance, so no one spotted the six billion divides by zero that occurred due to correct but unanticipated data
8 the values created by the parallel random number generator let you down and generated the same sequence on each processor which you failed to notice.

Outcome (4) is interesting. Genetic algorithms are extremely robust. Quite often they will deliver 'correct' results even if the code contains logical errors. Performance suffers but not the quality of the results. It would be nice to be able to build more self-healing codes.

The chapter also offered advice regarding performance optimisation. This consolidates the practical experiences of the preceding four chapters. Maybe it is all self-evident, but few parallel programmers have documented the tricks of their trade or the lessons they have learned, or talk about their experiences. Well, we have 'bared all', apart from the *really* embarrassing mistakes, which we do not care to talk about! The hope is that by reading about our experiences you will be able to improve on them.

10 Putting it all together

It is now 'do-it-yourself' time. This chapter introduces you to the idea of benchmarking HPC hardware. It then briefly describes the so-termed social science benchmark and then invites you to download the source code from a web site. This code will run on virtually any hardware ranging from a PC to a Unix workstation to the world's fastest parallel supercomputer. Having read this book, you should be able to understand most of the source code. You may even be possessed of sufficient curiosity to try it out. The next step after that is to move on to writing your own parallel programming software all by yourself.

10.1 Background

The purpose of this chapter is to put together some of the skills learned in previous chapters and use them to examine code that has been created to measure the performance of HPC hardware. Openshaw and Schmidt (1997) describe what they call the social science benchmark (SSB/1), which uses a spatial interaction model to measure the performance of HPC systems. Three versions of this code have been developed, and two of these are examined briefly here to illustrate different parallel programming styles. The three different codes are Highly Parallel Fortran (HPF), MPI and bulk synchronous parallel (BSP) model, although the latter is left for the reader to study. Our advice is that MPI offers the most flexible, the most portable and the most future-proof approach, but it is, nevertheless, still interesting to see how other languages perform the same task.

10.2 Introduction to benchmarking

Computer benchmarks serve no really useful research or scientific purpose other than to satisfy an innate curiosity about how fast one lump of HPC metal goes compared with another. It is interesting because different HPC hardware based on different components employing different architectures will run the same application quite differently. You must have wondered how badly your classic 33 MHz PC from 1993 runs in comparison with a Pentium II chip rated at

450 MHz? Does the Pentium II chip really run your code 450/33 times faster? Additionally, how do you measure speed at all? The clock frequency of the microprocessor is not much help in judging how fast it will run a model or handle a database. So the usual answer is to identify a representative application (often several) that the hardware is being purchased to run and then time how well it can handle it. Typically, you would also vary the amount of data being processed to check on the degree to which the code will scale to larger problems. Of course, no research council is (yet) likely to consider the purchase of a leading-edge HPC to meet the specialist needs of geographers or other social scientists. Nevertheless, it is still interesting to see how different hardware and different parallel-programming models and languages can handle a spatial interaction model.

A spatial interaction model is useful in this context because the explosion in the availability of flow data of all types has created a latent demand to run these models on data too large to be easily handled by more conventional computers. For example, to model the largest public domain flow table available in the UK (*viz.* the journey to work flows from the 1991 census for all wards in Britain) there is a requirement to process a table comprising 10,764 columns by 10,764 rows. Yet the maximum flow data table (i.e. telephone flows between houses) that may exist could well have between 1.7 and 30 million rows and columns. Data mining and modelling of these, and other even larger, data sets can now be undertaken and the results geographically analysed and modelled using HPC technologies. So HPC has the potential to open up new areas for research, but only if those researchers with the potential applications can grasp the new opportunities, if the applications are sufficiently important to attract HPC resources, and if the HPC hardware can cope with the computing loads likely to be placed upon it. Many other geographical problems are naturally data-intensive and put heavy demands on storage resources. They also require large amounts of memory and disk space. So given the processing capabilities of contemporary parallel systems, the modelling and analysis of these large data sets is now within reach.

There are many different standard scientific benchmarks that are often used to assess the performance of HPCs. Many vendors and NASA researchers reported the results for the NAS parallel benchmark, but it is representative only of certain types of aero-physics applications. The GENESIS benchmark (see Hey, 1991) for distributed-memory machines relates mostly to problems from physics and theoretical chemistry. The SPEChpc96 suite incorporates two application areas, the seismic industry computational work and computational chemistry, but has not yet been widely adopted. Perhaps the most commonly used is Dongarra's Linpack suite for solving dense systems of linear equations (see Dongarra, 1995), but none of these has much relevance to GIS, geocomputation or geography as most focus on raw-number crunching power, whereas most geographical applications are simultaneously memory-bound as well as computing-intensive. Hence the benchmark codes described here add to a pool of tools for evaluating parallel hardware from the perspective of this type of geography application. It is argued that what is good for geography is also useful for the other social sciences.

10.3 The spatial interaction model as a benchmark code

10.3.1 Brief description

The basic structure of a spatial interaction model has already been described in earlier chapters. The version used here is slightly different in that a sparse array data structure is used to hold the non-zero flows instead of storing them all. This allows much larger problem sizes to be considered, because it makes much more efficient use of memory, but it also destroys the simple data parallel structure of the models examined in earlier chapters. As a result, it presents a different and slightly greater parallel-programming challenge. Also, two different types of spatial interaction model are used here as benchmark applications.

10.3.2 An origin-constrained model

The equations for an origin (singly) constrained model (SC) are given below:

$$T_{ij} = O_i\, D_j\, A_i \exp(-\beta C_{ij})C_{ij}^{\alpha} \quad i = 1,\ldots,N \qquad j = 1,\ldots,M \tag{10.1}$$

with:

$$A_i = 1/\sum_{j=1}^{M} D_j f(C_{ij}) \qquad i = 1,\ldots, N \tag{10.2}$$

to ensure that:

$$\sum_{j=1}^{M} T_{ij} = O_i \qquad i = 1,..,N \tag{10.3}$$

where:
T_{ij} is the number of trips (flows) between zone i and zone j

O_i is the number of trips starting in zone (origin) i
D_j is the number of trips ending in zone (destination) j
C_{ij} is the distance between origin i and destination j
N is the number of origins
M is the number of destinations
$\exp(-\beta C_{ij})C_{ij}^{\alpha}$ is the deterrence function.

The modelling process requires the computation of all elements of $\{T_{ij}^{\text{predicted}}\}$, which is a highly parallelisable task. Typically, an error measure such as the sum of errors squared

$$F = \sum_{i=1}^{N}\sum_{j=1}^{M}\left(T_{ij}^{\text{predicted}} - T_{ij}^{\text{observed}}\right)^2 \tag{10.4}$$

would be used to assess the model's goodness of fit to an observed data set.

Another version of the singly constrained model is the destination-constrained model, where the constraints relate to the destination ends. This model is a mirror image of the origin-constrained model, and since it does not bring any new computations, it is not considered for benchmarking.

10.3.3 A doubly constrained model

The equations for a doubly constrained (DC) model are more complex than those for the singly constrained model and are as follows:

$$T_{ij} = O_i D_j A_i B_j \exp(-\beta C_{ij}) C_{ij}^{\alpha} \qquad i = 1,\ldots,N \qquad j = 1,\ldots,M \qquad (10.5)$$

where:

$$A_i = 1/\sum_{j=1}^{M} D_j B_j f(C_{ij}) \qquad i = 1,\ldots,N \qquad (10.6)$$

$$B_j = 1/\sum_{i=1}^{N} O_i A_i f(C_{ij}) \qquad j = 1,\ldots,M \qquad (10.7)$$

This model is now constrained at both ends:

$$\sum_{j=1}^{M} T_{ij} = O_i \qquad i = 1,\ldots,N \qquad (10.8)$$

and

$$\sum_{i=1}^{N} T_{ij} = D_j \qquad j = 1,\ldots,M \qquad (10.9)$$

where T_{ij}, O_i, D_j, C_{ij}, N, M and the $\exp(-\beta C_{ij})C_{ij}^{\alpha}$ function have the same meaning as described previously.

10.3.4 Interesting properties of the spatial interaction model as a benchmark

These models are useful for benchmarking because they represent a class of geographical application that is far more data-intensive than is found in many other areas of science. Typically, in a large number of geographical and social science applications, little computation is being executed in comparison with the number of memory references performed. This model is therefore a good example of a class of problems that make heavy demands on memory resources. Their performance reflects a machine's ability in executing a small ratio of arithmetic

operations to memory references. Additionally, this benchmark has a reasonably representative level of the non-floating-point integer arithmetic as it uses sparse matrix methods to store the observed flow data for T_{ij}. The need to perform large amounts of integer arithmetic is often overlooked in HPC benchmarks and in hardware designed solely to optimise pure number-crunching throughput on data stored in a highly optimised cache with a large amount of arithmetic for each memory access. In other words, benchmark codes based on physics, chemistry and most other hard science applications are probably not particularly appropriate for the computationally far less intense applications characteristic of many areas of geography, GIS and social science, e.g. neural networks, genetic algorithms and geographical models. HPC hardware that can do billions of dot products per second may not do nearly so well on a spatial interaction model code.

Finally, it is noted that in both models the quality (i.e. spatial resolution) of the results depends on both the number of origins (N) and the number of destination ends (M). Increasing values of N and M for a given geographical area lead to more realistic modelling at a finer spatial grain, resulting in arguably better science. However, this increases demands on memory as there is a need to store larger arrays. Models with small N and M values (i.e. 1000 or less) can be run on a PC; large values (i.e. 25,000) need a parallel system with many processing nodes (e.g. Cray T3D); and the bottom end of the maximum possible values (i.e. 1.6 million) may need the next generation or two of highly parallel teraflop computers before they can be processed at all.

The key features of the spatial interaction model that make it a good benchmark code are as follows:

1 The benchmark is easily portable to a broad range of hardware. It contains only a few hundred lines of code; therefore it is easy to understand and port to a new platform. A useful benchmark must require the minimum of vendors' time and effort to apply it. Porting a large code is a non-trivial task, and the simplicity of the model and code are important. Its small length also eases *Your* task of understanding the code.
2 The code is available in the public domain and can be easily downloaded from the World Wide Web; see http://www.leeds.ac.uk/ucs/projects/benchmarks/index.html It would be useful if you were to download this code and consult it in conjunction with the rest of this chapter.
3 The benchmark is based upon a model used in real-life applications, but it also encompasses a sparse data structure that is common to some other real-world applications. A good benchmark should be representative of the area of science it represents.
4 The benchmark has a built-in data generator, so there is no need to store any data on disk or perform large amounts of input and output during the benchmarking. This also avoids the problem of shipping large volumes of data around and yet permits benchmark runs to be performed on realistic and variable sizes of data set. The self-contained nature of the benchmark

code is another important ease-of-use feature. If you want a code to practice your parallel processing skills on then this is a good one because it is fairly simple, it comes with its own data and it is freely available.

5 The executable time and memory requirements for the benchmark are easily adjustable (you merely have to alter the problem size, i.e. *N*, *M* values). A standard set of ten (*N*, *M*) values is suggested that reflects different sizes of real-world application and offer a basis for scaleability and performance measurement experiments on comparable problems on different hardware.

6 The benchmark permits a wide range of data set sizes to be investigated, providing a platform for various numerical experiments and also allowing for scaling experiments with data sets of virtually any size that may be considered important in the future.

7 The ratio of computation to memory references in the benchmark is typical of many geographical and GIS problems requiring the use of high-performance computers. Much social science computing involves a high ratio of memory access to computation. In many statistical models, there is also a memory access for each floating-point operation performed.

8 The performance indicators can be readily interpreted because it is easy to establish a model of the computational load being generated by the benchmark. A model in this context is a count of the number of flops or integer adds or messages passed. How do you count these quantities? Well that is easy, you add counters to your code!

9 The use of exp and log functions in the deterrence function is deliberate. It is designed to be representative of mathematical library functions commonly used in geography and GIS. Note that HPCs designed for number crunching often fare poorly in computing billions of exponential or log functions.

10 The benchmark has a built-in results verifier that checks against the reference values whether or not the numerical results are acceptable for a standard set of benchmarks problem sizes. This is useful, because it ensures that any changes of compiler or hardware or new versions of code still produce the same, hopefully correct, answer.

10.4 The high-performance Fortran version

From a parallel-programming point of view both spatial interaction models are inherently data parallel. This feature was noted in Chapters 5 and 6. The available parallelism can be exploited by partitioning the data between several processors and assigning the work using the *owner computes rule*. The processor owning the data performs the computations on them and then communicates only the data elements needed by other processors to them. This form of data parallel computing can be readily operationalised via High-Performance Fortran (HPF).

The data generator in the benchmark creates the $\{C_{ij}\}$, $\{T_{ij}^{\text{observed}}\}$, $\{D_j\}$ and $\{O_i\}$ matrices; however, the trip matrix is stored in a sparse format. Sparsity is

measured as the percentage of zero elements to the total number of elements in the $\{T_{ij}\}$ matrix. This is fixed throughout the benchmark at 90%. The array $\{T_{ij}^{\text{observed}}\}$ is compressed by eliminating all zero elements and by storing only the non-zero values in a one-dimensional array. Additional arrays are created that contain pointers to the real position of the data in the uncompressed array. The arrays are communicated to all processors. The $\{O_i\}$ matrix (or vector) is also sent to each processor. However, this is where it becomes more difficult. The $\{D_j\}$ matrix (or vector) is generated by each processor separately and contains only the partial values that have been computed for the block of rows allocated to that processor. Subsequently, a global reduction across all the processors needs to be performed on them. This neatly avoids the problem of generating large flow tables using only one of multiple processors while the remainder are idle. If this is confusing, then maybe you need to reread the account of alternative data distribution strategies for the spatial interaction model given in Chapters 6 and 7.

Current HPF compilers target an SPMD (single-program multiple-data) programming concept and implement the owner computes rule; each processor executes the same program but operates on a local portion of the distributed data. Although HPF does not provide as much control over the data distribution as MPI, it does offer a dramatic simplicity of approach and implementation at a higher level of abstraction. Most importantly, HPF implementation does not require a serial code to be explicitly restructured for parallel processing, provided that the algorithm is already in a data parallel or vectorisable form.

For HPF, the cost array are distributed blockwise by rows. Another copy of the array is made and distributed blockwise by columns across parallel processors. The following statements are added to the serial code explicitly to map the data on to parallel processors:

```
!HPF$ DISTRIBUTE cij(BLOCK,*)
!HPF$ DISTRIBUTE cijc(*,BLOCK)
!HPF$ DISTRIBUTE rn(BLOCK,*)
!HPF$ DISTRIBUTE mcol(BLOCK,*)
```

This mapping has to reflect the way the data are to be accessed or performance will be greatly affected. The serial data generator was transformed to a data parallel version. The subroutine generating the data was modified not to cause side effects and converted into what is called a 'pure' subroutine. In the serial code, the $\{T_{ij}^{\text{predicted}}\}$ matrix is stored in sparse format in a one-dimensional array with an additional integer array for indexing. This approach was beneficial in terms of memory requirements. However, it could not be directly implemented in HPF, which lacks a convenient way for the indirect addressing of data. Instead, the sparse matrix, was substituted by a regular matrix which loses the benefits of sparse storage. HPF implementation of the benchmark contains a number of !HPF$ INDEPENDENT statements in front of the DO loops in the source code. Examples of statements converted into a parallel form by inserting the

!HPF$ INDEPENDENT directive are shown in Appendix 10.1. Finally, HPF allows a programmer to map and tune an algorithm on to a target architecture. Preliminary experiments with tuning did not deliver any significant performance improvement for the code running on the Cray T3D.

10.5 The message-passing code using MPI

The main reason for choosing the message-passing programming model is the portability of the benchmark which could be achieved by using a global standard for message passing–MPI and MPI2, which are now available on a wide variety of parallel architectures.

The efficiency of data distribution is very critical because both forms of spatial interaction models, like many other GIS problems, are data-intensive and memory-bounded. In this case, the data distribution strategy affects the size of the problem that can be computed on a parallel system. For the MPI implementation, the simplest data distribution strategy would be to broadcast the entire $\{C_{ij}\}$ matrix to each processor. This is an efficient method in terms of the amount of inter-process communications required, as the matrix is accessed in read-only mode by each processor. However, this would severely limit the size of the data sets that can be used for benchmarking. Instead, the $\{C_{ij}\}$ matrix must be distributed (or generated) blockwise to each processor. Different strategies are required for each of the two models.

For *the singly constrained model*, the partial sums of trips and partial sums of origins are computed on separate processors. The values are communicated to the master processor, where the reduction followed by broadcasting takes place. While computing the model, each processor works (in parallel) on its own separate block of data. It computes a block of rows of the $\{T_{ij}^{\text{predicted}}\}$ using the block of rows of the $\{C_{ij}\}$ matrix needed for this task. Finally, each processor, computes partial error measures that are collected by the master processor, where the global reduction takes place and the final global errors are computed. Computation of the model requires very little inter-process communication, which takes place only in the last step where the partial error sums are reduced (*viz.* added together to form a global sum).

For *the doubly constrained model* the same data distribution strategy is used, with the exception that the $\{C_{ij}\}$ values now also need to be distributed by columns. Each block of rows generated by an individual processor is divided into smaller chunks containing blocks of columns and then scattered among other processors. This task, which is also performed in parallel, ensures that each processor has in its local memory the block of rows and the block of columns of the cost matrix that it will be working on.

Equations (10.6) and (10.7) imply that the values of A_i and B_j terms are functions of each other and have to be estimated iteratively until the required convergence criterion is satisfied. The use of a convergence criterion would result in variable amounts of computational work being performed as the amount of data changed over and above that due to problem size changes. To avoid this, the

number of iterations performed to compute the A_i and B_j is fixed at twenty. The parallel algorithm first sweeps forward across the rows computing the A_i values and communicating them to other processors and then sweeps across the columns computing the B_j values and communicating them to all other processors. This process is repeated until a fixed number of iterations has been performed. The basic operations for the doubly constrained model and the corresponding communication pattern are shown in Appendix 10.2.

Finally, after A_i and B_j have been calculated and communicated to all processors, each processor computes a block of rows of the $\{T_{ij}^{\text{predicted}}\}$ matrix and the partial error measures that are collected by the master processor, where the global reduction on them takes place and the global errors are obtained. Note that for the DC model computations are interspersed with the communication between processing nodes. It is also a good problem on which to practise parallel decomption skills.

10.6 The bulk synchronous parallel model

The bulk synchronous parallel (BSP) model has not previously been described in this book. It is one of a few alternative promising parallel-programming models. The original idea was proposed by Valiant (1990). Since then, it has been developed into a general-purpose approach to parallel computing at Oxford University by McColl and others. They believe that it offers a robust model for parallel computation, with the prospect of both scaleable performance and architecture-independent software. The code for this version of the spatial interaction model is also available via David Henty of the EPCC, Edinburgh University;

see http: //www.ccg.leeds.ac.uk/

for details of where it can be downloaded. The BSP model is attracting much attention from computer scientists because it has some nice theoretical properties. It is uncertain as yet whether it constitutes a practical tool.

10.7 Measuring performance using MPI and serial code

The aim of benchmarking is to compare the performance of different computer systems in relation to the representative application being used for the benchmark. The performance indicator used here is the elapsed execution time for one evaluation of the model. This is an absolute metric. It indicates which hardware performs best running this type of application. In other words, it allows the user to identify the platform (with the smallest value of the performance indicator) that is seemingly most suitable for running code that has a similar mix of computations to that of the benchmark. Note that as in all benchmarks problems of a dissimilar computational structure may perform very differently. HPC hardware is notoriously quirky, and the selection of hardware is nearly always going to favour some applications more than others.

A number of computer systems have been evaluated using the benchmark, ranging from PCs through workstations to vector processors and massively parallel systems. Table 10.1 presents the results (the elapsed execution time for one

Table 10.1 Results for the parallel singly constrained model with 1000 origins and 1000 destinations.

System name	*Number of processors*	*Time (ms)*	*Relative performance*
SGI Origin	14	46.2	180.8
SGI Origin	12	52.1	160.3
Cray T3D	256	61	136.9
SGI Origin	10	62	134.7
SGI Origin	8	78.5	106.4
Cray T3D	128	129	64.7
SGI Origin	4	154	54.2
SGI Onyx	4	173	48.3
Cray T3D	64	202	41.3
SGI Origin	2	323	25.9
Cray T3D	32	439	19.0
SGI Power Challenge	4	530	15.8
IBM SP2	8	560	14.9
Cray T3D	16	796	10.5
Cray J90	8	847	9.9
IBM SP2	4	1060	7.9
SGI Challenge	4	1479	5.6
Cray T3D	8	1580	5.3
Cray T3D	4	3186	2.6

Source: Openshaw and Schmidt (1997).

evaluation of the model) for the origin-constrained model with 1000 origins and 1000 destinations for different parallel platforms, and Table 10.2 for some serial HPC hardware. Note that the third column contains the execution time of the benchmark and the fourth column the relative performance of the system, which has been computed as the execution time of the benchmark on the system under evaluation divided by the execution time of the benchmark on a 133 MHz Pentium PC. Tables 10.3 and 10.4 present equivalent results for the doubly constrained model.

The relative performance measure suggests that the singly constrained model runs 137 times faster on the 256-processor Cray T3D than on a PC, but the doubly constrained model runs only 86 times faster in comparison with a PC run. This may be due to a large amount of inter-processor message passing in the doubly constrained model. It is also a reflection of the small (by HPC standards) problem size. Note also that the 1000-origin by 1000-destination sizes were the largest that could be run on all available platforms. Finally, the problem size with 25000 × 25000 origin–destination pairs computed very quickly (13 seconds for the singly constrained model and 570 seconds for the doubly constrained model) on the 512-processor Cray T3D. However, this size of model could not be computed on any of the other systems because of memory restrictions. This illustrates another benefit of HPC: large real memory sizes. Benchmark results for other problem sizes and systems are available on the

Table 10.2 Results for the serial singly constrained model with 1000 origins and 1000 destinations.

System name	*Number of processors*	*Time(s)*	*Relative performance*
Fujitsu VPX	S	170	49.1
DEC Alpha 600	S	729	11.5
SGI Origin	S	730	11.4
SGI Onyx	S	900	9.3
DEC 8400	S	988	8.5
SGI Power Challenge	S	1744	4.8
Sun Ultra-2	S	2144	3.9
Sun Ultra-1	S	2491	3.4
HP9000/K460	S	3110	2.7
HP9000/C160	S	3180	2.6
SGI Indy	S	3645	2.3
IBM SP2	S	4743	1.8
Cray CS6400	S	5122	1.6
Cray J90	S	6085	1.4
SGI Challenge	S	6360	1.3
Pentium PC	S	8351	1.0
Sun 10/41	S	8459	1.0
Cray T3D	S	12832	0.7
KSRI	S	22379	0.4
486 PC	S	33242	0.3

Source: Openshaw and Schmidt (1997).

World Wide Web and are discussed more fully in Openshaw and Schmidt (1997).

10.8 A comparison of HPF and MPI codes

The benchmark can also be used to measure the performance of HPF relative to MPI. The Portland Group's High-Performance Fortran (pghpf) compiler was used for the HPF runs. Table 10.5 contains the results for the singly constrained model for the problem size 1000 by 1000, and the results for the 5000 by 5000 problem size are given in Table 10.6.

The *speed-up factor* is computed as:

$$S(M) = T(1)/T(M)$$

where T(1) is the execution time of the benchmark on 1 processor and T(M) is the execution time on M processors. Calculating this factor, we used as T(1) the elapsed execution time of one model evaluation for each implementation on the Cray T3D. The speed-up statistics show how well both implementations performed with an increase in the number of processors.

Table 10.3 Results for the parallel doubly constrained model with 1000 origins and 1000 destinations.

System name	*Number of processors*	*Time (s)*	*Relative performance*
SGI Origin	14	1.57	155.5
SGI Origin	12	1.76	138.8
SGI Origin	10	2.08	117.4
SGI Origin	8	2.59	94.3
Cray T3D	256	2.83	86.3
Cray T3D	128	4.41	55.4
SGI Origin	4	5.12	47.7
SGI Onyx	4	5.39	45.3
Cray T3D	64	6.67	36.6
SGI Origin	2	10.5	23.3
Cray J90	8	14.5	16.8
Cray T3D	32	14.5	16.8
IBM SP2	8	20.8	11.7
SGI Power Challenge	4	22.4	10.9
Cray T3D	16	25.7	9.5
IBM SP2	4	41.2	5.9
Cray T3D	8	51.4	4.8
SGI Challenge	4	67.9	3.6
Cray T3D	4	102.7	2.4

Source: Openshaw and Schmidt (1997).

Table 10.4 Results for the serial doubly constrained model with 1000 origins and 1000 destinations.

System name	*Number of processors*	*Time (s)*	*Relative performance*
Fujitsu VPX	S	6.56	37.2
SGI Origin	S	19.2	12.7
SGI Onyx	S	19.5	12.5
DEC Alph 600	S	24.0	10.2
DEC 8400	S	34.3	7.1
SGI Power Challenge	S	72.0	3.4
Sun Ultra-2	S	73.5	3.3
Sun Ultra-1	S	87.6	2.8
Cray J90	S	88.0	2.8
HP9000/K460	S	102.5	2.4
HP9000/C160	S	103.3	2.4
SGI Indy	S	129.3	1.9
IBM SP2	S	166.9	1.5
Cray CS6400	S	167.5	1.5
SGI Challenge	S	196.0	1.2
Pentium PC	S	244.2	1.0
Sun10/41	S	277.7	0.9
Cray T3D	S	457.0	0.5
486 PC	S	1045.8	0.2

Source: Openshaw and Schmidt (1997).

Table 10.5 Comparison of MPI and HPF using the singly constrained model.

Number of processors	*SC 1000 × 1000 problem size*				
	HPF execution time (ms)	*HPE speed-up*	*MPI execution time (ms)*	*MPI speed-up*	*Relative difference for HPF & MPI (%)*
1	13697	100	11351	1.00	20.67
4	3422	4.00	2943	3.86	16.28
8	1709	8.01	1294	8.77	32.07
16	864	15.85	644	17.63	34.16
32	440	3113	327	34.71	34.56
64	221	61.98	165	68.79	33.94
128	112	122.29	83.3	136.27	34.94
256	57	240.30	43	263.98	32.56
512	30	456.57	22	515.95	36.36

Table 10.6 Comparison of MPI and HPF using the singly constrained model.

Number of processors	*SC 5000 × 5000 problem size*				
	HPF execution time (ms)	*HPF speed-up*	*MPI execution time (ms)*	*MPI speed-up*	*Relative difference for HPF & MPI (%)*
16	*		15997	16.07	–
32	10937	32.00	8032	32.00	36
64	5506	63.56	4036	63.68	36
128	2786	125.62	2038	126.12	37
256	1395	250.88	1022	251.49	36
512	699	500.69	515	499.08	36

Note: *there was insufficient memory to run this size of problem with 16 processors.
Source: Openshaw and Schmidt (1979).

The *relative difference* is calculated as follows:

$$D(M) = (T_{MPI}(M) - T_{HPF}(M)/T_{MPI}(M)$$

where $T_{HPF}(M)$ is the execution time for HPF implementation and $T_{MPI}(M)$ is the execution time for the MPI implementation. The values show the degree to which the MPI implementation outperforms the HPF implementation.

The MPI implementation of the singly constrained model performed better in terms of the execution time than the HPF implementation for both problem sizes on the Cray T3D. For the 1000 by 1000 problem size, the relative difference for the four processor job is approximately 16 per cent. With an increase of the number of processors to eight, the relative difference doubles to 32 per cent and then increases slightly to 36 per cent for different numbers of processors up to 512. The 5000 by 5000 HPF implementation produces results on average 36 per cent

Table 10.7 Comparison of MPI and HPF using the doubly constrained model.

Number of processors	*DC 1000 × 1000 problem size*				
	HPF execution time (s)	*HPF speed-up*	*MPI execution time (s)*	*MPI speed-up*	*Relative difference for HPF & MPI (%)*
1	433	1.00	441	1.00	1.81
4	109	3.97	124.4	3.55	12.38
8	54.4	7.96	54.29	8.12	0.20
16	27.7	15.63	27.17	16.23	1.95
32	14.5	29.86	13.9	31.73	4.32
64	8.44	51.30	7.13	61.85	18.37
128	8.18	52.93	3.94	111.93	107.61
256	19	22.79	2.86	154.20	564.34
512	65	6.66	3.02	146.03	2052.3

Source: Openshaw and Schmidt (1997).

Table 10.8 Comparison of MPI and HPF using the doubly constrained model.

Number of processors	*DC 5000 × 5000 problem size*				
	HPF execution time (s)	*HPF speed-up*	*MPI execution time (s)*	*MPI speed-up*	*Relative difference for HPF & MPI (%)*
16	*	–	672.09	21.65	–
32	353.7	32	454.8	32.0	22
64	179.8	62.95	208.9	69.67	14
128	96.6	117.17	87.9	165.57	10
256	64	176.85	44.9	324.13	43
512	90.7	124.79	29.9	486.74	203

Note: *there was insufficient memory to run this size of problem with 16 processors.
Source: Openshaw and Schmidt (1997).

worse than the MPI implementation. This high relative difference between implementations is due to the type of the model under consideration in which the ratio of memory references to computations is very high. The computation of this model involved very little inter-processor communications, so the main overheads were probably due to the creation of parallel threads and building the communications schedule into the HPF implementation. Nevertheless, given the simple and highly data parallel nature of the singly constrained model, this is a very disappointing result for HPF and must cast doubts over its usefulness in those applications where maximum execution time performance is required.

Tables 10.7 and 10.8 contain equivalent results for the doubly constrained models. The 1000 by 1000 model shows approximate linear scaleability up to 64 processors. However, with more processors performance no longer scales. This is most evident for the HPF implementation, which achieves a scaleability

of 51 for 64 processors and 53 for 128 processors but to only 7 for 512 processors! The MPI implementation with 64 processors displays a scaleability of 62, which slowly increases to 146 for 512 processors. With a large number of processors, the workload per processor is very low and thus the overheads of parallelisation and communications prevent the performance of the model scaling. For example, for the 128-processor run each processor had only eight rows of data to work on before communicating the values and then worked on eight columns, and again this was followed by communication. This process was repeated for twenty iterations to compute the A_i and B_j terms before the model was computed. The doubly constrained model results demonstrate that the user must be aware of the type of operations in an application when transforming it to a parallel version. The ratio of communication to computation in an application must be known to choose the best number of processors for the job. The HPF implementation brings additional overheads, but these are only easily recovered for small numbers of processors, where the amount of work per processor is larger.

The HPF implementation for the 5000 by 5000 problem size obtains a scaleability of 117 for 128 processors and 124 for 512 processors. However, the MPI implementation scales linearly up to 512 processors although it is possible that with a further increase in the number of processors this implementation will not scale much further for reasons given earlier. This is also a reflection of Amdahl's law. The solution is to scale up the problem size as the number of processors increases. The problems seen here would then disappear. Finally, the very poor performance of the HPF implementation is rather disturbing. It is questionable whether the gains in ease of programming using HPF are worth the loss of performance in HPC. After all, HPC should be most useful for those applications on the edge of what is computable, and it is here where squeezing the last drops of speed out of a code, a compiler and hardware is likely to be most worth while.

10.9 Conclusions

This chapter has briefly described a social science benchmark (SSB/1) code suitable for measuring the performance of the widest possible set of HPC hardware on a representative social science and geographical computational problem. The benchmark is thought to be a useful means for geographers and other HPC users to quantify the performance of the diversity of computational systems they have or could have access to. A large number of performance results have been collected, and only a few have been reported here; these are available on the World Wide Web; see

http://www.leeds.ac.uk/ucs/projects/benchmarks

The performance tables express HPC developments in terms of the hardware that was available for geographers to use in the period 1994–1998. However, such is the current rate of improvement in hardware speeds that it is important for the performance results that are presented to be updated regularly. Knowing how fast a computer runs is fundamental to geographical users of HPC as well as being interesting in its own right. Meanwhile, the reader has access to different versions of code for the same model. They are cordially invited to see whether their versions of MPI or HPF or HPC hardware can do much better.

Appendix 10.1: HPF code fragment for a singly constrained spatial interaction model

```
!HPF$INDEPENDENT,NEW(k1,k2,t,j,icol,aa,pred,sum,b,
add on),and !HPF$&        REDUCTION (f1,f2,c1)

      DO 33000 i=1, n
       k1=ip (i)
       k2=num (i)
       DO 11 j=1, m
        t (j)=0.0
11      END DO
       IF (k2.ge.k1) then
        DO 12 j=k1, k2
         icol=mcol (i,j)
          t (icol)=rn (i,j)
12     END DO
      ENDIF
      sum=0.0
       DO 14 j=1, m
       aa=beta * cij (i, j)
       IF (aa.gt.prec) aa=prec
       IF (aa.lt. - prec) aa=-prec
       pred=exp (aa) * d (j)
       addon=cij (i, j) **alpha
       pred=pred * addon
       sum=sum + pred
       b (j)=pred
  14  END DO
      IF (sum.lt.small) sum=small
      sum=o (i) / sum
      DO 384 j=1, m
      pred=b (j) * sum
      f1=f1 + (pred-t (j) ) **2
      f2=f2 + abs (pred-t (j))
      c1=c1 + pred * cij (i,j)
384  END DO
33000 END DO
```

Appendix 10.2: Parallelisation of a doubly constrained spatial interaction model

master node communicates to each processor:	istart,iend; jstart,jend; n, m, nproc, alpha, beta
data generation on each (r-th)processor:	C_{kj} , k = istart,. . .,iend; j = 1,. . .,m
	T_{kj}^{observed} , k = istart,..,iend; j=1,. . .,m
	O_i, i = 1,. . .,n
	partial values of D^r_j , j = 1,. . .,m
master node reduces and broadcasts:	$Dj = \sum_{r=1}^{p},m$
each processor communicates to others:	C_{kl} , k = istart,. . .,iend; l = jstart,. . .,jend
each processor computes:	$T^r = \sum_{k=istart}^{icend} \sum_{j=1}^{M} T_{kj}^{\text{observed}}; O^r = \sum_{k=istart}^{icend} O_k;$
	$Z^r = \sum_{k=istart}^{icend} \sum_{j=1}^{M} T_{kj}^{\text{observed}} f(C_{kj})$
master node reduces:	$T = \sum_{r=1}^{p} T^r ; O = \sum_{r=1}^{p} O^r ; Z = \sum_{r=1}^{p} Z^r$
master node computes and broacasts:	$K = Z/O$
loop for 20 iterations:	
each processor (r-th)computes:	$A_k = \sum_{j=1}^{im} B_j f(C_{kj})$, K = istart,. . .,iend; $SA^r = \sum_{k=istart}^{iend} A_k$
each processor communicates to master:	A_k, k = istart,. . .,iend
master broadcasts:	A_i, i = 1,. . .,n
master node reduces received partial *SA*^{r}:	$SA = \sum_{r=1}^{p} SA^r$
each processor computes:	$B_1 = \sum_{i=1}^{n} A_i f(C_{il})$, l = start,. . .,jend; $SB^r = \sum_{l=jstart}^{jend} B_l$
master node reduces received partial *SB*^{r}:	$SB = \sum_{r=1}^{p} SB^r$
master node computes and broadcasts:	$f = SB/SA$
each processor computes:	$B_l = D_l\ B_l\ /\ f$, l=jstart,. . .,jend;
each processor communicates to master:	B_l, l=jstart,...,jend
each processor computes:	$F^r = \sum_{k=istart}^{icend} \sum_{j=1}^{m} (T_{kj}^{\text{predicted}} - T_{kj}^{\text{observed}})\ 2;$
	$G^r = \sum_{k=istart}^{icend} \sum_{j=1}^{m} (T_{kj}^{\text{predicted}} - T_{kj}^{\text{observed}})\ ;$
	$H^r = \sum_{k=istart}^{icend} \sum_{j=1}^{m} T_{kj}^{\text{predicted}} f(C_{kj})$
master node reduces partial errors:	$F = \sum_{r=1}^{p} F^r : G = \sum_{r=1}^{p} G^r ; H = \sum_{r=1}^{p} H^r$
master node computes:	$E = (K - H/O))^2$
master node broadcasts the error norms:	E, F; G

11 Epilogue for geographers and social scientists

This chapter stands back from the intricacies of HPC and parallel processing and looks briefly at the context within which these developments are located. It attempts to identify some of the new opportunities for the greater use of HPC in a socially responsible way. In some ways, it builds on the arguments of Chapters 1 and 2 and attempts to project them into the future world of the twenty-first century.

11.1 The global challenge

A professor of computer science at the University of Newcastle upon Tyne once said 'that faster computers merely allowed users to make more mistakes faster'. He was probably right. HPC certainly creates an environment within which users can make the most colossal of errors! Parallel programming is a major mistake-facilitation tool of historically unprecedented power! However, this power and speed is also why HPC is such a key technology. In common with many other scientists, geographers need HPC (as indeed do many of the social sciences) to cope with the rapidly increasing complexities associated with studying most aspects of the modern world. The complaint at present is that not only are we failing to cope but also that many of the contemporary methodologies and tools are actually moving backwards into qualitative descriptions of the unique rather than even standing still. Instead of trying to compute our way out of an immense data inundation that continues to grow in both intensity and depth of coverage, too many of our colleagues have given up using advanced informatics, preferring instead to observe the effects on others. The world needs both, but neither paradigm is by itself sufficient.

People's behaviour is now known to be far more complex and difficult to understand than we previously dared to fear. There are three further complications. All over the world, human behaviour is changing, and the intensity and frequency of social–economic–physical interactions are both increasing and shifting with new patterns and forms of spatial human, economic and perhaps political organisational structures appearing at many different scales and levels of generalisation. If that is not sufficient, static cross-sectional models of the recent (often the not so recent) past now desperately need to be replaced by dynamic

models of micro-and macro-behaviour and patterns capable of functioning in real time in a proper spatial setting.

Major new global challenges are becoming evident that will require a substantial research response involving HPC, probably very soon. Here are just a few of a growing list.

1 Global climatic change could conceivably affect billions of people, but we have no good idea of what the human or economic effects may be or how to model or forecast them or their geographical distribution and intensity. Seemingly, we are close to a global environmental catastrophe of historically unparalleled dimensions, but not a single flop of HPC is currently being devoted to exploring and understanding any of it.

2 Most of the world's population now lives in urban areas, but the best models we have of the dynamics and growth of urban areas are about thirty years 'old' and reflect quite different interests and historical periods, as well as being constrained by what can be termed those palaeo-methodologies that pre-date HPC. Is this sensible? Why are there not HPC-based planning models able to simulate and hence better inform the land-use planning process? Is it acceptable to use planning methodologies that have scarcely changed since the Second World War?

3 Increasingly, people live in areas polluted by chemicals released into their living environments. How can the controllable human and institutional systems be organised, changed, even manipulated to reduce the potential health risks? What are these risks? What geographical monitoring systems are needed to provide assurance that major epidemics are not occurring? Why do the guardians of public health and safety have no access to HPC to help to answer these questions? Why are they so self-confident in their legacy technologies that, maybe, they see no need even to try? Why are they so unaware?

4 Why do we continue to gather data and store it if the systems for its analysis and modelling cannot cope with more than a very conservative 0.0001 per cent of it? How should we cope with the data flood? What new machine-based inductive knowledge-creating tools are now required, and what will it take to build them? To what extent is this a uniquely geographical problem, and how can HPC be used to resolve it?

5 Why do the GIS systems we have today have so few new functionalities compared with ten years ago? The geo-data functions have not extended much beyond geo-data collection, but this is no longer sufficient. Once it mattered much less than now. What can be done about it? Is it a reflection of a lack of computing power? Or is it a lack of imagination? Is it a lack of awareness?

6 Why do so many human geographers shut themselves off from the challenges of coping with the modern digital world, ignoring all the data, failing to learn any IT-related skills and becoming trapped in an abstract self-critical conceptual black hole of their own creation? Instead of watching

and describing people and organisations failing to cope, maybe we should start offering geo-help. The underlying assumption seems to be that you cannot have a scientific approach to studying people and society, because the task is too difficult for statistical methods to cope with it. Maybe, historically speaking, this observation was more correct that it is today. HPC changes everything and provides the powerhouse for developing entirely new scientific human geographical methodologies if we want them and, more significantly, if sufficient human geographers have the skills to enable them to develop new tools, new models and new methodologies. Modern human geographers have (in theory) so much to offer but, in common with many other social sciences, they have seemingly and apparently deliberately chosen to do nothing but serve their own overly self-constructed scholarly concerns, largely oblivious to the greater and more urgent needs of the outside world they claim to study. Even worse, they reuse, reinterpret and reclaim the vocabulary of the earlier scientific geography but assign a new but distorted meaning that has nothing to do with science. They abuse the words, using them as a veneer to hide their own intellectual shortcomings and, while totally denying all scientific affiliations, they 'bask' in the intangible scholastic benefits that their scientific fraud gives them. Their greatest challenge now is to start to become computer-oriented by 'importing' their ideas in a computer (not statistical or mathematical form or literary form). Maybe they have much to contribute to understanding the world of tomorrow; maybe they have nothing. However, without some means of testing or validating or formalising or standardising or reconstructing their ideas in a computer-compatible form, no one will ever know. If the 'theories' have something important to say about the processes and operation of the modern world, if they are indeed significant additions to collective knowledge and wisdom, then some means needs to be found of 'using them'. If they can account for what is missing, then why not create computer systems and models that include in them the essence of their theories? This may entail envisaging entirely new and completely different computer-based methodologies. What of it? Just because it has not been done before does not mean that it cannot be done now. Just because these methodologies do not exist does not mean that they cannot be invented soon. Engagement and geocomputational translation would be both an interesting experiment and a fundamentally important activity for the twenty-first century. It will inevitably become increasingly difficult for human geographers to sustain their ignorance of the realities of having IT in a 100 per cent digital world for much longer. No matter how far you retreat into the depths of obscure sociology or history or cultural anthropology, there is no escape from the digital realities. HPC provides a basis for bringing this world back into science.

7 There is a growing ethical imperative for geographers to return to using and developing their traditional map skills. The 'where' component is even more important than previously in a data-drenched world. But so too is the 'why'.

New ways of using the location dimension of geoinformation need to be devised. What exists today is a legacy technology that has been computerised and given a modern 'feel', but the tools themselves need updating. HPC has a pivotal role to play here.

8 Survival at all levels will increasingly depend on companies, agencies, countries and disciplines discovering how to make 'good use' of their databases. They need to understand individual behaviour while investing nothing in basic research. Research councils throughout the world are seemingly charged with this responsibility, but they are massively under-resourced and are still firmly wedded to research agendas that largely pre-date the 1990s. They are still focusing far too much on the themes and priorities of yesteryear while lacking the vision and pro-activity needed for the world of next year.

9 Governments also need to understand more about the concerns and interests of their citizens. An Internet digital democracy has the potential to change many established procedures, especially where the issues are hard, contentious and a cause of great public concern. However, electronic referenda are no substitute for a micro-understanding of human behaviour in key areas of state importance so that attitudes and behaviours may be better understood and, where important to the good functioning of society and government, predicted. Governments all over the world are increasingly being forced by circumstances beyond their control to adopt policies that require changes in behaviour patterns or restrict public freedoms, e.g. to combat traffic congestion and transport-related atmospheric pollution. To be successful, you need to be able to model the likely response and then correctly forecast the long-term outcome. This is a serious practical problem that currently cannot be handled except in a most facile way. Even the economic modelling of national economies is laughable in its lack of sophistication, starved of resources, and almost a complete joke in terms of the underlying science, yet the consequences of error are massive. It needs to be done properly, and this would appear to require computational models as complex as any found in other areas of science, not a set of 1960s-style linear simultaneous equations run on a PC. Why does the Bank of England not use a Cray T3E (or faster machines) to assess the effects of interest rate rises on the British economy? The subject is so important as to justify virtually anything! Cost is not a relevant factor when errors cost billions! The problem is that bankers are not innovative; they are just pathetic in their use of old methods that were more appropriate to the 1960s, but they get away with it! Why? How can it be that the economic planning of major developed countries is so poor? Do they even realise what is now possible?

10 A major crime occurs. Society expects that all available resources be used to apprehend the criminal(s) responsible. Yet in the long history of HPC, what fraction of all Mflops has ever been devoted to either crime pattern analysis or criminal detection? Is it 0.0 or 0.00000001 per cent? Why?

11 We live in a world facing global climate change, but essentially we lack the science necessary to (1) prove beyond reasonable doubt that it is

happening; (2) identify with a reasonable degree of certainty what the impacts will be; (3) indicate the speed of possible changes; and (4) measure the possible effects on people and society. If this issue is really of earth-shattering importance and could affect billions of people then why are we not devoting billions of research pounds to developing a better understanding of the physical and human processes? It really does matter, but the world is just not treating it with the seriousness it should. Why?

12 'Big Brother' exists but is currently ineffective, clumsy and laughable. However, many areas of a future world would benefit from benevolent computer systems designed to improve the 'public good'. Openshaw (1992) outlines a few possibilities. At present, we do not have a good idea of how to do much of it. It is not about the infringement of civil liberties or privacy but merely an attempt to ensure that some more of the evils and dangers of the modern world can be reduced, managed or (ideally) avoided.

11.2 What has HPC got to do with any of this?

That should be obvious. It is a key enabling technology on which many of the necessary tools will one day have to be built. The modern and future world is a 100 per cent digital world. It is important to come to terms with what this means. Currently, our technological abilities to build HPCs, and gather and store data far outstrip the abilities of geography and the social sciences to do virtually anything at all useful with it. The same applies to countries and most companies. Yet in a digital world you cannot ignore it for long. The question now is whether or not HPC offers a way of computing our way out of many of the problems of today's world. Does computation provide the basis for new tools that are better able to cope? Does it offer a paradigm that can thrive on massive data, incorporate soft knowledge and tackle at least some of the world's pressing problems? In short, does computation provide a viable basis for a new science of geography and even for putting some of the science back into social science?

The short answer is *yes*. You could add 'but' as a qualification. Indeed, we are convinced that it does, but it will only do so if more geographers and social scientists can start to grasp the opportunities, learn the new skills and evolve their computational thinking from where it is now to where it could be if they thought more along the lines of a computational physicist or chemist or aeronautical engineer applied to a human context. It cannot be done overnight, but the time to start is most definitely now.

11.3 Revising the definition of geocomputation

To geographers, the emergence of geocomputation as a late-1990s paradigm is interesting; see Longley *et al.* (1998), Openshaw and Abrahart (1999) The original idea of geocomputation is a good one, but it is also a little deficient. Geocomputation was invented in 1996 to reflect a fusion of three key technologies of interest to the geosciences:

1 HPC (high-performance computing);
2 AI (artificial intelligence, computational intelligence); and
3 GIS (geographical information systems).

It is deficient only that there needs to be an explicit fourth component:

4 HSM (human systems modelling)

It makes less and less sense to model processes in the physical environment without also including people and society as processes, even in the physical systems being investigated. People live on river terraces and have done so for almost as long as the fluvial processes acting on landscapes that appeared from under the last Ice Age. People are a geomorphological process as much as water is! Also, lest we forget, global climatic change is created by human systems interacting with global environmental processes. It is time that efforts were made to build computer models of the principal human systems with the eventual goal of modelling individual people and their basic space–time patterns of behaviour. It is no longer sufficient merely to understand their culture but to improve the scientific understanding sufficiently to allow us to begin to build predictive models of people and their behaviour patterns that matter. HSM is not an easy option. The computer methodologies needed to make it happen in general do not yet exist, except in a most rudimentary form. Most of the conceptual framework is missing. All we have is a mass of descriptive understanding and even more masses of data. It is time for research councils to be brave, to grasp the nettle of HPC in a people science context, and start pro-actively to initiate its usage.

Computer-based geographical data mining might be useful in data-logged situations where we have far more data than commonsense or knowledge. In other areas, there is soft knowledge that we need to try to incorporate into our computer models. It is not an easy task. People are complex entities and need to be modelled as such. If you wished to build a dynamic model of 57 million people seeking to cover at least some of their daily space–time experiences and behaviour, you would probably not use any of the mathematical and statistical models of classical quantitative geography or social science. Equally, existing HPC would not be fast enough or big enough: and even if the computer power existed, the modelling methodologies do not. However, this does not mean that they could not be created, or that because of this or that outburst by pompous, self-righteous, computer Luddites it would be a mistake even to try. At a time when biologists are creating new life forms, never seen before in the cause of science, it would be very silly to declare that we should not even try to formulate and solve equivalent grand challenges in the people-modelling arena! If social scientists fail to grasp the possibilities, then no doubt others will. Maybe social scientist's do not have long to get their research act together before other scientists in the form of engineers realise that people matter too. However, maybe there are also other more obviously pressing issues that need to be handled.

11.4 A GIS–HPC research agenda

If attention is switched to GIS then the situation is also problematic, but far less so. GIS has a data-computer-quantitative and mapping science culture that is alive and flourishing on a global scale. No dramatic change of paradigm is needed here. Yet this apart, the same legacy technology problems remain to be overturned, and HPC has an important role to play. GIS urgently needs to improve its exploratory geographical analysis and spatial modelling toolkits. As their users complete their data-capture projects and after a decade or so complete their institutionalisation of GIS in an enterprise setting, so they will want to collect on the secondary promises that GIS holds out of adding value to data via analysis and modelling and not just from data integration. The problems are once again the lack of a suitable technology able to meet these generic needs in an end-user context. If suitable technology existed, the GIS developers would probably have already packaged it, but much of what is needed does not exist, although the bare outlines of what is needed are fairly well known; see, for example, Openshaw (1991, 1995b, 1996). Maybe it is also a matter of waiting for faster hardware to permit these methods to be readily used on end-user platforms. However, this particular excuse is evaporating rapidly.

Openshaw (1998b) and Openshaw and Perree (1996) outline a view that the end-user ability problem can be resolved by developing a visual spatial analysis technology that is able to communicate its findings with its users because the results are self-evident and intuitively obvious. The map (and map animation) are wonderful result-communication tools, but they are extremely deficient if they are used solely as data display and spatial analytical devices (their traditional uses). If animation (and multi-media) communication potentials are to be properly utilised then it is important that the results being displayed are meaningful and not just pretty and colourful representations of randomness. HPC has a key role to play here. You also need intelligent agents (in the broadest sense) able to explore universes of hyper-dimensional data complexity and then report their discoveries via maps.

A subsequent issue is how then would you integrate HPC-dependent tools within the non-HPC-driven world of today's GIS? One solution is to create stand-alone Internet-accessible tools that communicate with GIS via data files and that are special-purpose virtual machines. They perform only one function each. Their Internet location means that they could be powered by HPC or downloaded by registered users on to a local HPC, or run on whatever hardware they need. You no longer need to have everything available in the same software package located on a single lump of hardware and all accessed by a single interface. In the Internet age, functionality can be distributed in cyberspace. This provides an obvious and easy way of using HPC in a seamless, transparent and almost totally invisible manner so that virtually anyone could use it without tears. You can read about a WWW GAM in Openshaw *et al.* (1999b), or try it out on

http: //www.ccg.leeds.ac.uk/smart/intro.html

The required functions for this new world of geographical analysis include the following:

1 Exploratory geographical analysis of the GAM type, which searches for localised clustering in space in any geographically referenced database.
2 Geographical explanation machines (GEMs), which seek to find recurrent geographical associations that provide a 'geographical explanation' for the clustering as a means of generating explanatory clues; see Openshaw *et al.* (1999a).
3 Exploratory space-time pattern hunting. If a GAM were to be used then computing times could well increase by two or three orders of magnitude and GAM-T would become a wholly HPC-dependent tool (once again).
4 There are prototype space–time–attribute smart explorers that would broaden the pattern hunt to all the data domains that characterise GIS (space, time, attributes of cases) or any permutations of them. The basic principles have been established (see Openshaw 1994d, 1995b), but full-scale geographical data mining (GDM) applications await experimentation using synthetic data on HPC platforms. Indeed, these raise an important observation. HPC may be more useful in providing a basis for the experimental testing of new analysis tools that once perfected may be able to perform the vast majority of applications on far less powerful hardware. This sort of HPC testing is something that could be more widely applied.
5 Spatial modelling is another major need. Once again some of the basic tools exist, but maybe they need to be made easier to use by users who have little statistical training. Impossible, you say. Well why not try thinking about it from a computational perspective! One computational strategy would be as follows: select a broad variety of alternative models, apply them all to your data and use cross-validation methods to identify the best one. Such a modelling machine is adaptive and slightly intelligent. It finds the 'best' from a set of alternative models, and it could base the search on hundreds or thousands of likely candidates. 'Ah!' cry the conventionalists. 'What a waste when an expert could find a good model for you.' However, this is totally ignores the fact that (1) such experts are rare compared with the number of potential users and data sets; (2) there is no assurance that the expert would find a good, let alone the best, model for arbitrary data that he has no prior knowledge about; and (3) there could be much real confidence in an applied spatial data-modelling setting that the expert in one technology could compete with a modelling machine with access to several different ones. Maybe there is a basis here for some kind of challenge?
6 Real-time analysis and modelling present another set of problems of generic importance.
7 Finally, prototype new model-breeding machines have been demonstrated; see Openshaw (1988), Turton *et al.* (1997), Diplock (1996, 1998). Why not simply scale them up from prototype demonstrators to real applications?

11.5 HPC futures in geography, etc.

The purpose in raising all these issues in this chapter is to re-emphasise the point that if much progress is going to be made, at least in the short term, it is essential that HPC in geography, GIS and the social sciences achieves a much higher profile. If we want new HPC-powered tools, at least initially, we will have to develop them ourselves as no one else is likely to do it for us. This has implications for recommended research training programmes, as postgraduate students are currently not being trained in or exposed to any HPC skills.

We have attempted to show that HPC

- is applicable to geography and GIS
- offers considerable potential here and in other social sciences
- and the new skills needed to use it are not difficult to learn
- is available now.

We hope that by producing a 'chatty', informal text this will help to foster all four of these objectives. At the very least it will help to demystify the myth of complexity. If we can do it, so should you!

References and further reading

Alexander, F.E. and Boyle, P., 1996, *Methods for Investigating Localised Clustering of Disease.* IARC Scientific Publications No. 135, Lyon, France.

Almasi, G.S. and Gottlieb, A., 1989, *Highly Parallel Computing.* Benjamin-Cummings Publishing Co., Redwood, Calif.

Amdahl, G.M., 1967, 'Validity of the single processor approach to achieving large scale computing capabilities', in *AFPIS Conference Proceedings* 30, 483–485.

Appel, A., 1985, 'An efficient program for many-body simulations'. *SIAM Journal on Scientific and Statistical Computing* 6, 85–103.

Baker, L. and Smith, B.J., 1996, *Parallel Programming.* McGraw-Hill, New York.

Bell, G., 1994, 'Scalable, parallel computers: alternatives, issues and challenges'. *International Journal of Parallel Programming* 22, 3–46.

Bentley, J., 1986, *Programming Pearls.* Addison-Wesley, Reading, Mass.

Besag, J. and Newell, J., 1991, 'The detection of clusters in rare diseases'. *Journal of the Royal Statistical Society,* Series A 154, 143–155.

Bezdek, J.C., 1994, 'What is computational intelligence?' in J.M. Zurada, R.J. Marks and C.J. Robinson (eds) *Computational Intelligence: Imitating Life*, IEEE. New York. 1–12.

Birkin, M., Clarke, G., Clarke M. and Wilson, A.G., 1996, *Intelligent GIS.* GeoInformation International, Cambridge.

Birkin, M., Clarke, M. and George, F.. 1995, 'The use of parallel computers to solve non-linear spatial optimisation problems'. *Environment and Planning A* 27, 1049–1068.

Bomans, L., Roose, D. and Hempel, R, 1990, 'The Argonne/GMD macros in FORTRAN for portable parallel programming and their implementation on the Intel iPSC/2'. *Parallel Computing*, 15, 119–132.

Boyle, J., Butler, R., Disz, T., Glickfeld, B., Lusk, E., Overbeek, R., Patterson, J. and Stevens, R., 1987, *Portable Programs for Parallel Processors.* Holt, Rinehart, & Winston.

Braun, T., 1993, *Parallel Programming.* Prentice-Hall, New York.

Brocklehurst, E.R., 1991, *Survey of Benchmarks*, NPL Report, DITC 192/91.

Brunsdon, C., Fotheringham, A.S. and Charlton, M.E., 1996, 'Geographically weighted regression: a method for exploring spatial nonstationarity'. *Geographical Analysis* 28, 281–298.

Catlow, C.R.A., 1992, *Research Requirements for High Performance Computing.* Report of the Scientific Working Party. SERC, Swindon.

Chalmers, A. and Tidmus, J., 1996, *Practical Parallel Processing: An Itroduction to Problem Solving in Parallel.* International Thomson Computer Press, London.

Chandy, E.M. and Taylor, S., 1992, *An Introduction to Parallel Programming.* Jones and Bartlett Publishers, Boston.

Coucelis, H., 1998, 'GeoComputation in context.' in Longley *et al.* (eds) *Geocomputation: A Primer.* Wiley, Chichester.

Demmel, J., 1996, 'CS267: Lecture 1, Jan. 1996: Introduction to Parallel Computing', from http: /www.CS.Berkeley.EDU/~ demmel/cs267/lecture01.html

Densham, P. J. and Armstrong, M.P. , 1998, 'Spatial analysis', in R. Healey, S. Dowers and B. Gittings (eds) *Parallel Processing Algorithms for GIS.* Taylor & Francis, London. 387–413.

Dibble, C., 1996, 'Theory in a complex world: agent based simulation of geographical systems'. *Proceedings of GeoComputation 1* (Leeds University, September), Vol. 1, 210–213.

Diplock, G.J., 1996, *The application of evolutionary computing techniques to spatial interaction modelling* (unpublished PhD thesis, University of Leeds).

Diplock, G.J., 1998, 'Building new spatial interaction models by using genetic programming and a supercomputer'. *Environment and Planning A* 30, 1893–1904.

Diplock, G.J. and Openshaw, S., 1996, 'Using simple genetic algorithms to calibrate spatial interaction models'. *Geographical Analysis* 28, 262–279.

Dongarra, J.J., 1995, Performance of various computers using standard linear equations software, CS-89-85, Computer Science Department, University of Tennessee, Knoxville, TN 37996-1301.

Dowd, K., 1993, *High Performance Computing.* O'Reilly and Associates, Sebastopol, Calif.

Efron, B. and Gong, G., 1983, 'A leisurely look at the bootstrap, the jack-knife, and cross-validation'. *American Statistician* 37, 36–48.

Ein-Dor, P., 1985. 'Grosch's Law revisited'. *Communications of the ACM* 28, 142–151.

EPSRC 1995, *A Review of Supercomputing 1994.* EPSRC, Swindon.

Flynn, M.J., 1972, 'Some computer organisations and their effectiveness'. *IEEE Trans Computers* 21, 948–960.

Foster, I., 1995, *Designing and Building Parallel Programs.* Addison-Wesley, New York.

Fotheringham, A.S., Charlton, M.E. and Brunsdon, C., 1997, 'Two techniques for exploring nonstationarity in geographical data'. *Geographical Systems* 4, 59–82.

Fox, G., 1988, *Solving Problems on Concurrent Processors: General Techniques and Regular Problems.* Prentice-Hall, Englewood Cliffs, N.J.

George, F., 1993, 'Spatial interaction modelling on parallel computers'. EPCC-PAR-GMAP Report, EPCC, Edinburgh University.

Getis, A. and Ord, J.K., 1992, 'The analysis of spatial association by use of distance statistics'. *Geographical Analysis* 24, 189–206.

Gilbert, G.N. and Doran, J., 1994, *Simulating Societies: the Computer Simulation of Social Phenomena.* UCL Press, London.

Gilbert, N. and Conte, R. (eds) 1995 *Artificial Societies.* UCL Press, London.

Gropp, W., Lusk, A., and Skjellum, A., 1994, *USING MPI: Portable Parallel Programming with the Message-Passing Interface.* MIT Press, Cambridge, Mass.

Gustafson, D., Rover, D., Elbert, S., and Carter, M., 1991, 'The design of a scaleable, fixed-time computer benchmark'. *Journal of Parallel and Distributed Computing* 12, 388–401.

Harris, B., 1985, 'Some notes on parallel computing with special reference to transportation and land use modelling'. *Environment and Planning A* 17, 1275–1278.

Healey, R., Dowers, S., Gittings, B. and Mineter, M. (eds) 1998, *Parallel Processing Algorithms for GIS.* Taylor & Francis, London.

Hey, A.J.G., 1991, 'The GENESIS Distributed Memory Benchmarks'. *Parallel Computing* 17, 1275–1283.

Hillis, W.D., 1992, 'What is massively parallel computing and why is it important?' in N. Metropolis and G.C. Rota (eds) *A New Era in Computation.* MIT Press, Cambridge, Mass. 1–15.

Hockney, R.W. and Jesshope, C.R., 1981, *Parallel Computers.* Adam Hilger, New York.

Hockney, R.W. and Jesshope, C.R., 1988, *Parallel Computers 2: Architecture, Programming and Algorithms.* Institute of Physics Publishing, Bristol and Philadelphia.

Hwang, K., 1993, *Advanced Computer Architecture: Parallelism, Scalability, Programmability.* McGraw-Hill.

Hwang, K. and Briggs, F.A., 1984, *Computer Architecture and Parallel Processing.* McGraw-Hill, New York.

Kernighan, B. and Plauger, P.J., 1974, *The Elements of Programming Style.* McGraw-Hill, New York.

Kober, R., 1988, *Parallelrechner-Architekturen.* Springer-Verlag, Berlin.

Kogge, K.M., 1981, *The Architecture of Pipelined Computers.* McGraw-Hill, New York.

Koza, J.R., 1992, *Genetic Programming.* MIT Press, Cambridge, Mass.

Koza, J.R., 1994, *Genetic Programming II: Automatic Discovery of Re-usable Programs.* MIT Press, Cambridge, Mass.

Landau, R.H. and Fink, P.J., 1993, *A Scientist's and Engineer's Guide to Workstations and Supercomputers.* Wiley, New York.

Lewis, T.G. and El-Rewini, H. 1992, *Introduction to Parallel Computing.* Prentice-Hall, New Jersey.

Longley, P.A., Brooks, S.M., Mcdonnell, R. and Macmillan, B. (eds) 1998, *Geocomputation: A Primer.* Wiley, Chichester.

Martin, D., 1998, 'Optimizing census geography: the separation of collection and output geographies. *International Journal of Geographical Information Science* 12 (in press).

Morse, H.S., 1995, *Practical Parallel Computing.* AP Professional, Boston.

MPI specification, version 1.1, 1995, ftp://ftp.mcs.anl.gov/pub/mpi/mpi-1.jun95/mpi-report.ps

O'Hare, G.M.P. and Jennings, N.R., 1996, *Foundations of Distributed Artificial Intelligence.* Wiley, New York.

Openshaw, S., 1976, 'A geographical solution to scale and aggregation problems in region building, partitioning and spatial modelling'. *Transactions of the Institute of British Geographers,* New Series 2, 459–472.

Openshaw, S., 1978, 'An empirical study of some zone design criteria'. *Environment and Planning A* 10, 781–794.

Openshaw, S., 1984, 'Ecological fallacies and the analysis of areal census data'. *Environment and Planning A* 16, 17–31.

Openshaw, S., 1987, 'Some applications of supercomputers in urban and regional analysis and modelling'. *Environment and Planning A* 19, 853–860.

Openshaw, S., 1988, 'Building an automated modelling system to explore a universe of spatial interaction models'. *Geographical Analysis* 20, 31–46.

Openshaw, S., 1990, 'Automating the search for cancer clusters: a review of problems, progress and opportunities', in R.W. Thomas (ed) *Spatial Epidemiology.* Pion, London. 48–78.

Openshaw, S., 1991, 'A new approach to the detection and validation of cancer clusters: a review of opportunities, progress and problems', in F. Dunstan and J. Pickles (eds) *Statistics in Medicine.* Clarendon Press, Oxford. 49–64.

Openshaw, S., 1994a, 'Computational human geography: towards a research agenda'. *Environment and Planning A* 26, 499–505.

Openshaw, S., 1994b, 'Computational human geography: exploring the Geocyberspace'. *Leeds Review* 37, 201–220.

Openshaw, S., 1994c, 'Neuroclassification of spatial data', in B. Hewitson and R. Crane (eds) *Neural Nets: Applications in Geography*. Kluwer Academic, Dordrecht. 53–70.

Openshaw, S., 1994d, 'Two exploratory space–time–attribute pattern analysers relevant to GIS', in S. Fotheringham and P. Rogerson (eds), *Spatial Analysis and GIS*, Taylor & Francis, London. 83–104.

Openshaw, S., 1994e, 'A concepts rich approach to spatial analysis, theory generation and scientific discovery in GIS using massively parallel computing', in M. Worboys (ed.) *Innovations in GIS*. Taylor & Francis, London. 123–138.

Openshaw, S., 1995a, 'Human systems modelling as a new grand challenge area in science'. *Environment and Planning A* 27, 159–164.

Openshaw, S., 1995b, 'Developing automated and smart spatial pattern exploration tools for GIS applications'. *The Statistician* 44, 3–16.

Openshaw, S., 1996, 'Developing GIS relevant zone based spatial analysis methods', in *M. Batty* and *P. Longley* (eds) *Spatial Analysis: Modelling in a GIS Environment*. GeoInformation International, Cambridge. p 55–73.

Openshaw, S., 1998a, 'Neural network, genetic and fuzzy logic models of spatial interaction'. *Environment and Planning A* 30, 1857–1872.

Openshaw, S., 1998b, 'Building automated geographical, analysis and explanation machines', in Longley *et al.* (eds) *Geocomputation: A Primer*. Wiley, Chichester. 95–116.

Openshaw, S., 1998c, 'Towards a more computationally minded scientific human geography'. *Environment and Planning A* 30, 317–332.

Openshaw, S. and Abrahart, R.J. (eds) 1999, *GeoCompution*. Taylor & Francis, London.

Openshaw, S. and Alvanides, S., 1999, Applying GeoComputation to the analysis of spatial distributions, in P. Longley, M.F. Goodchild, D.J. Maguire, D.W. Rhind (eds) *GIS: Principles, Techniques, Management and Applications*. Wiley, New York.

Openshaw, S., Blake, M. and Wymer, C., 1995, 'Using neurocomputing methods to classify Britain's residential areas', in P Fisher (ed.) *Innovations in GIS2*. Taylor and Francis, London. p 97–112.

Openshaw, S., Charlton, M., Craft A. and Birch, J.M., 1988, 'An investigation of leukaemia clusters by use of a geographical analysis machine'. *The Lancet* Feb 6, 272–273.

Openshaw, S., Charlton, M., Wymer, C. and Craft, A. 1987, 'A mark I geographical analysis machine for the automated analysis of point data sets'. *International Journal of GIS* 1, 335–358.

Openshaw, S. and Craft, A., 1991, 'Using the geographical analysis machine to search for evidence of clusters and clustering in childhood leukaemia and non-Hodgkin lymphomas in Britain', in G. Draper (ed.) *The Geographical Epidemiology of Childhood Leukemia and Non-Hodgkin Lymphomas in Great Britain, 1966–83*. HMSO, London. 109–122.

Openshaw, S., Cross, A. and Charlton, M., 1990, 'Building a prototype geographical correlates exploration machine'. *International Journal of GIS* 3, 297–312.

Openshaw, S. and Openshaw, C.A., 1997, *Artificial Intelligence in Geography*. Wiley, London.

Openshaw, S. and Perree, T., 1996, 'User centered intelligent spatial analysis of point data', in D. Parker (ed.) *Innovations in GIS 3*. Taylor & Francis, London. 119–134.

Openshaw, S. and Rao, L., 1995, 'Algorithms for re-engineering 1991 census geography', *Environment and Planning A* 27, 425–446.

Openshaw, S. and Schmidt, J., 1996, 'Parallel simulated annealing and genetic algorithms for re-engineering zoning systems', *Geographical Systems* 3, 201–220.

Openshaw, S. and Schmidt, J., 1997, 'A social science benchmark (SSB/1) Code for serial, vector, and parallel supercomputers'. *International Journal of Geographical and Environmental Modelling* 1, 65–82.

Openshaw, S. and Sumner, R., 1995, 'Parallel spatial interaction modelling on the KSR1-64 supercomputer'. Working Paper 95/15, School of Geography, University of Leeds.

Openshaw, S. and Turton, I., 1996, 'A parallel Kohonen algorithm for the classification of large spatial datasets'. *Computers and Geosciences* 22, 1019–1026.

Openshaw, S., Turton, I. and Macgill, J., 1999a, 'Using the geographical analysis machine to analyse census long term limiting illness data', *Geographical and Environmental Modelling* 3., 83–99.

Openshaw, S., Turton, I., Macgill, J. and Davy, J., 1999b, 'Putting the geographical analysis machine on the Internet'. *Innovations in GIS 7*. Taylor & Francis, London.

Openshaw, S., Wilkie, D., Binks, K., Wakefield, R., Gerrard, H.H. and Crosdale, M.R., 1989, 'A method of detecting spatial clustering of disease', in W.A. Crosbie and J.H. Gittus (eds) *Medical Response to the Effects of Ionising Radiation*. Elsevier, Amsterdam.

Pachero, P. S., 1997*, Parallel Programming with MPI*. Morgan Kaufmann Publishers, San Francisco.

Pitas, I., 1993, *Parallel Algorithms: For Digital Image Processing, Computer Vision and Neural Networks*. Wiley, Chichester.

Sawyer, M., 1998, 'Software environments and standardisation initiatives', in R. Healey, S. Dowers and B. Gittings (eds) *Parallel Processing Algorithms for GIS*. Taylor & Francis, London. 33–57.

Small, D.R. and Edelstein, H.A., 1997, 'Scaleable Data Mining'. Available from Two Crows Corp. on the WWW.

SPEChpc96 Results http: //www.specbench.org/hpg/results.html.

Treleaven, P., Brownbridge, D.R. and Hopkins, R.P., 1982, 'Data driven and demand driven computer architecture'. *ACM Computing Surveys* 14, 95–143.

Trewin, S.M., 1998, 'High-level support for parallel programming', in R. Healey, S. Dowers and B. Gittings (eds) *Parallel Processing Algorithms for GIS*. Taylor & Francis, London. 59–86.

Turton, I., 1997, 'Application of Pattern Recognition to Concept Discovery in Geography, Unpublished MSc dissertation, Human Geography, School of Geography, University of Leeds, Leeds, LS2 9JT.

Turton, I. and Openshaw, S., 1996, 'Modelling and optimising flows using parallel spatial interaction models', in L. Bouge, P. Fraigniaud, A. Mignotte., Y. Roberts (eds.) *Euro-Par' 96 Parallel Processing* Vol. 2, Lecture Notes in Computer Science 1124. Springer, Berlin. p 270–275.

Turton, I. and Openshaw, S. 1997, 'Parallel spatial interaction models'. *Geographical and Environmental Models* 1, 179–197.

Turton, I. and Openshaw, S., 1998, 'High performance computing and geography: developments, issues, and case studies'. *Environment and Planning A* 30, 1839–1856.

Turton, I., Openshaw, S. and Diplock, G.J. 1996, 'Some geographical applications of genetic programming on the Cray T3D supercomputer', in C. Jesshope and A. Shafarenko (eds) *UK Parallel '96*. Springer, Berlin. 135–150.

Turton, I., Openshaw, S. and Diplock, G.J., 1997, 'A genetic programming approach to building new spatial models relevant to GIS', in Z. Kemp (ed.) *Innovations in GIS* 4. Taylor & Francis, London. 89–102.

Valiant, L.G., 1990, A bridging model for parallel computation. *Commun. ACM*, 33, 103–111

Wilson, A.G., 1970, *Entropy in Urban and Regional Modelling*. Pion, London.

Wilson, A.G., 1974, *Urban and Regional Models in Geography and Planning*. Wiley, London.

Wilson, A.G., 1981, *Geography and the Environment, System Analytical Methods*. Wiley, London.

Wilson, A.G., Coelho, J.D., MacGill, S.M. and Williams, H.C.W.L., 1981, *Optimisation in Location and Transport Analysis*. Wiley, Chichester.

Wilson, G.W., 1995, *Practical Parallel Programming*. MIT Press, Cambridge, Mass.

Xiong, D. and Marble, D.F., 1996, 'Strategies for real-time spatial analysis using massively parallel SIMD computers: an application to urban traffic flow analysis'. *International Journal of GIS* 10, 769–789.

Index

Hi Pf Com & Art of Para Pr
ISBN 0-415-15692-0
9 780415 156929