

Parallel Algorithms for Atmospheric Modelling

Simon F.B. Tett

**Doctor of Philosophy
University of Edinburgh
1992**



Abstract

In this thesis, the usefulness of massively parallel computers of the MIMD class to satisfy atmospheric modellers demands for increased computing power is examined.

Algorithms to use these computers, for the dynamics, are developed for both grid-point and spectral methods. Scaling formulae for these algorithms are developed and the algorithms are implemented on the Edinburgh Concurrent Supercomputer (ECS).

Another component of atmospheric models is parameterization; in which the effects of unresolved phenomenon on the mean-flow are modelled. Two parameterization schemes are implemented on the ECS and a study of the effects of load-balancing is made. Furthermore it is concluded that implementation of parameterization schemes on data-parallel computers is likely to be difficult, unlike MIMD machines where the implementation is straightforward.

Acknowledgements

First I would like to thank my three supervisors, Bob Harwood, Richard Kenway and Peter White for their various and valued contributions to my education. In particular I would like to mention Bob, for encouraging me to nail this thesis to a church door. I would also like to thank Alan Dickinson at the U.K. Meteorological office who “minded” me during my CASE visits. Thanks go to Tuomo Kauranne at the European Centre for Medium Range Weather Forecasting for many stimulating discussions on the subject of parallel computing and Meteorology. The spectral model code used in Chapter 4 was provided by the Department of Meteorology, Reading University. I am grateful to Mike Blackburn and Andy Heaps for help with understanding the workings of the model.

Financial support was provided by the S.E.R.C. and the U.K. Meteorological Office.

Reference should be made here to my colleagues and friends in Edinburgh who made my Ph.D time there such a pleasure: Steve, Ken and Hon for many Friday nights; Nick who shared an office with me; everyone in 2009 who showed me the ins and outs of the Meiko kit; Mike Brown for keeping the ECS going; Brian Pendleton for relentless good cheer; Charlotte for her friendship; my parents for knowing that I'd make it.

I would also like to mention Richard, Lorna and Andrew and all the rest at E.U. S.F. soc. for many “interesting” nights; Jeremy, Simon and Alan who provided entertainment in Oxford for me whenever I visited Bracknell

and the various people in E.U. Wargames Soc. for the many games that we played.

Thanks to Simon and Robert who put up with me while I wrote this thesis.

Many thanks are due to the Hadley Centre for employing me and giving me time and computer facilities to write this thesis. Particular thanks are to Vicky Pope for the use of her groups' Vax.

Most especially I'd like to thank Claire for her love and devotion to me, and also for latenight typesetting and proofreading above and beyond the call of duty.

Table of Contents

Abstract	1
Declaration	1
Acknowledgements	3
Table of Contents	5
List of Figures	9
List of Tables	12
1. Introduction	14
1.1 Thesis Overview	19
1.2 Parallel Architectures	22
1.2.1 Modestly Parallel Architectures	23
1.2.2 Data-Parallel Architectures	25
1.2.3 MIMD Architectures	27
1.3 Physical Meteorological Background	32
1.3.1 Dry Primitive Equations	32
1.3.2 Hydrodynamic Cycle	35

<i>Table of Contents</i>	6
1.3.3 Co-ordinates	36
1.4 Numerical Methods For Meteorology	37
1.4.1 The Spectral Method	39
1.4.2 Grid-point Methods	45
1.4.3 Fast Fourier Transform	51
1.5 Summary	54
2. Parallel Algorithm Design	55
2.1 Overview	55
2.2 Decomposition Strategies	57
2.3 Topology	58
2.4 Deadlock and Livelock	61
2.5 Associative Operators	64
2.6 Parallel Efficiency	67
2.6.1 Throughput versus Efficiency	70
2.6.2 Measuring Parallel Efficiency	72
2.7 Input/Output	72
3. Subgrid Processes	75
3.1 Introduction	75
3.2 Properties of Parameterization Schemes	80
3.3 Load-Balancing	91
3.4 Conclusion and Discussion	100

4. Spherical Harmonic Methods	102
4.1 Introduction	102
4.2 Pipelines	104
4.2.1 General Pipelines	105
4.2.2 Pipelining the Spectral Method	108
4.3 The Fast Fourier Transform	110
4.3.1 Balancing the Fourier Pipeline	115
4.3.2 Inverse FFT	117
4.3.3 Real Fourier Transforms	118
4.3.4 FFT Summary	124
4.4 Legendre Transform	126
4.5 The Full Spectral Transform	135
4.5.1 The Pipeline	136
4.5.2 Time-Step Increments	142
4.6 Implementation Details	149
4.7 Results	153
4.8 Conclusion and Possible Extensions	161
5. Grid-Point Methods	166
5.1 Introduction	166
5.2 Limited Area Models	168
5.2.1 Geometrical Decomposition	168
5.2.2 Vertical Decompositions	181

5.3	Global Models	187
5.3.1	Computing the Maximum Velocity	187
5.3.2	Fourier Transforms in Grid Point Models	190
5.3.3	Load-balancing the Fourier Transforms	194
5.4	Conclusion	203
6.	Conclusion	206
A.	Terminology	214
B.	Grid-point Model Description	217
B.1	Primitive Equations	217
B.2	Finite Difference Scheme Used	218
B.2.1	Adjustment Step	218
B.2.2	Advection Scheme	221
C.	Technical Proofs	222
C.1	Derivations for the FFT	222
C.1.1	Mapping Proof	222
C.1.2	Proofs for the Real packing of the FFT	223
C.2	Derivations for the Legendre Tree	224
C.3	Vertical Iterations Grouping factor	226
	Bibliography	228

List of Figures

1-1	The T800 Transputer	29
1-2	T800 communicating speeds	31
1-3	One dimensional grids	49
1-4	Arakawa grids	50
2-1	Deadlock in a ring	61
2-2	Process design that leads to livelock	63
2-3	Topology for cascade sum	65
2-4	Schematic illustration of parallel iterative operations	68
3-1	Surface data used for the large-scale rainfall parameteriza- tion scheme	82
3-2	Surface data used for the convection scheme	83
3-3	Time distribution functions for convection and rainfall schemes	84
3-4	Distribution of work for the large-scale rainfall scheme	86
3-5	Distribution of work for the convection scheme	87
3-6	Scaled time for parameterization schemes	89
3-7	Schematic diagram of load-balancer	93
3-8	Scaled speedup with load-balancing	98

4-1	A simple pipeline	105
4-2	Task work patterns	107
4-3	Flowchart of spectral transform	109
4-4	Pipeline and computational complexity for spectral method	109
4-5	The processor topology for the fast Fourier transform	115
4-6	Communications pattern for computation of real fields . . .	121
4-7	Communications pattern for computation of real fields from a transformed complex field.	123
4-8	Processor topology for entire FFT	127
4-9	Processor topology for Legendre transform	129
4-10	Processor topology for the inverse spectral transformation .	138
4-11	Communications within trees	143
4-12	Plot of relative speedup vs β and ΔS_{\max}	147
4-13	T21 model partitioned into 9 T7 modules	152
4-14	Times taken with increasing Gaussian latitudes	159
4-15	Estimate of efficiency, processors and speedup against trun- cation number	162
5-1	Processors with grid over-laid.	169
5-2	Idealized separation of sub-domain into inner and outer re- gions	171
5-3	A more typical example of how the grid-points split up into two regions	171
5-4	Calculated efficiencies for different communications.	177

5-5	Times and efficiencies for the grid-point model with changing sub-domain size	179
5-6	Times taken with varying processor number	180
5-7	Movement of data so that every atmospheric column is on a single processor	182
5-8	Processor topology used to map a sphere	188
5-9	A grid point processor row with a FFT butterfly added	192
5-10	Vertical Butterfly	195
5-11	Simulated effects of FFT load-balancing	198
5-12	Times per site for multiplexer	200
5-13	Speedups for the FFT with different amounts of load-balancing	202

List of Tables

1-1	Parallel computing taxonomy	22
1-2	T800 compute speeds	32
3-1	Minimum and maximum times for two schemes	76
3-2	Parameterization schemes on the ECS	90
3-3	Balancer results	99
4-1	Required parallel complexity	110
4-2	Maximum speedup	132
4-3	Maximum speedup for the non-concurrent communications	132
4-4	Properties of the spectral transform components (concurrent case)	140
4-5	Properties of the spectral transform components (non-concurrent case)	140
4-6	Times for the serial spectral model	155
4-7	Sizes for varying spectral truncations	155
4-8	Times for the parallel spectral model	156
4-9	Extra computations required by the parallel model	157
4-10	Effects of increasing task numbers	158

5-1	Values of $N(c, I_R)$ for the grid-point model	176
5-2	Values of $k(c, I_R)$ for the grid-point model	176
5-3	Timing results for grid-point model with a fixed number of processors	178
5-4	Timing results for grid-point model	180
5-5	Times for grid-point model with FFTs	193
5-6	Times for grid-point model with software multiplexing . . .	199
5-7	Times for FFT load-balancing	201

Chapter 1

Introduction

The first attempt at a numerical weather prediction was described in a book by Richardson (1922). In that book, in a flight of fantasy, he envisaged using massive parallelism to build a computational engine to carry out operational weather forecasting. The nodes of this engine were people, and he planned on using many of them as the following quote illustrates.

After so much hard reasoning, may one play with a fantasy? Imagine a large hall like a theatre, except that the circles and galleries go right through the space usually occupied by the stage. The walls of this chamber are painted to form a map of the globe. The ceiling represents the north polar regions, England is in the gallery, the tropics in the upper circle, Australia on the dress circle and the antarctic in the pit. A myriad computers are at work upon the weather of the part of the map where each sits, but each computer attends only to one equation or part of an equation. Numerous little "night signs" display the instantaneous values so that neighbouring computers can read them. Each number is thus displayed in three adjacent zones so as to maintain communications to the North and South on the map. From the

floor of the pit a tall pillar rises to half the height of the hall. It carries a large pulpit on its top. In this sits the man in charge of the whole theatre; he is surrounded by several assistants and messengers. One of his duties is to maintain a uniform speed of progress in all parts of the globe. In this respect he is like the conductor of an orchestra in which the instruments are slide-rules and calculating machines. But instead of waving a baton he turns a beam of rosy light upon any region that is running ahead of the rest, and a beam of blue light upon those who are behind.

Four senior clerks in the central pulpit are collecting the future weather as fast as it is being computed and dispatching it by pneumatic carrier to a quiet room. There it will be coded and telephoned to the radio transmitting station.

Messengers carry piles of used computing forms down to a storehouse in the cellar.

In a neighbouring building there is a research department, where they invent improvements. But there is much experimenting on a small scale before any change is made in the complex routine of the computing theatre. In a basement an enthusiast is observing eddies in the liquid lining of a huge spinning bowl, but so far the arithmetic provides the better way. In another building are all the usual financial, correspondence and administrative offices. Outside are playing fields, houses, mountains and lakes, for it was thought that those who compute the weather should breathe of it freely.

Richardson's fantasy captures the main ideas of massive parallelism: he has each one of his computers responsible for a small region of the atmosphere and his computers communicate with their nearest neighbours by sending messages to one another (the night signs). In order to produce an

operational forecast Richardson estimated that 64,000 human computers would be required. However Richardson's attempt at Numerical Weather Prediction (NWP) was flawed. The timestep he used was too long and so his method was numerically unstable. This problem is discussed later in this chapter. The first successful NWP using a digital computer was carried out by Charney *et al.* (1950). One of the authors of that paper, von Neumann, was involved in many early developments in computing including the von Neumann architecture where the processor has a single memory for instructions and data and fetches one instruction or datum at a time. The processor carries out one instruction on one datum and is, in essence, the serial computer in widespread use today. In contrast to the von Neumann architecture there now exist parallel architectures with several processors.

Since the mid 1960's, numerical models have been used to forecast the future behaviour of the atmosphere. The complexity, resolution and predictive power of those models has been increasing ever since. Bengtsson (1991) states that the five day prediction of the 500 mbar height field is as accurate in 1991 as the one day forecast was in the early 1950's. In the paper by Simmons *et al.* (1989) it is shown that the impact of a higher resolution spectral model on forecast skill in the one to ten day period is positive. One important user for weather forecasts is the civil aviation industry; the paper by White *et al.* (1987) shows how increased model resolution and complexity has had a positive impact on forecast skill. However weather forecasts are only useful for a limited period of time, therefore the computations for the numerical weather prediction must be done quickly. Thus in order to utilize more complex models and higher resolution models for NWP it is necessary to use increasingly powerful computers. The development of these trends at the U. K. Meteorological Office (UKMO) is shown in Houghton (1991).

Juckes and McIntyre (1987) investigated a single layer high resolution model and concluded that an accurate representation of the steep gradi-

ents necessary to understand some important stratospheric dynamical phenomenon required high resolution models.

Extended range prediction of future weather in the timescale of 10-30 days, in which several forecasts from slightly different initial conditions were integrated forwards in time have been described by Murphy (1988); Branković *et al.* (1990) and Tracton *et al.* (1989). Branković *et al.* carried out extended range forecasts by integrating nine forecasts from different initial conditions. The initial conditions were consecutive analyses with six hour separations between them. The forecasts are all integrated forwards in time until they have all reached a point 30 days in the future from the latest analysis. In order to carry out this process significant amounts of computing power are required.

A significant problem in atmospheric modeling is predicting the effect of increasing atmospheric carbon dioxide and other trace gases from man-made sources on the world's climate. Interest in the possibility of climate change lead to the formation, in 1988, of the International Panel on Climate Change (IPCC). The scientific assessment has been published (Houghton *et al.*, 1990) and amongst the conclusions reached are ^{that there is} a need for greater resolution in atmospheric models and that available computer power is a serious limiting factor on coupled atmosphere-ocean models (see Chapters 5 and 6 of Houghton *et al.* (1990)). Climate models are integrated forwards in time for many decades, unlike the six days of a typical NWP model. Traditionally the resolutions of climate models have been significantly lower than that of NWP models. Amongst the conclusions, in the paper by Tibaldi *et al.* (1990), was that a very low resolution version of the then ECMWF model failed to correctly simulate the non-linear dynamics of the extratropical regions. Climate modelling therefore poses a great challenge to computing in that it requires considerable amounts of computing resource.

Having shown that increased computing power is required, how will this be achieved? The present generation of supercomputers, of which the Cray Y-MP is one example (discussed in Subsection 1.2.1), are vector computers with small numbers of processors. Existing supercomputers have a single shared memory which all processors can access. It is unlikely that this technology can be pushed significantly further forward. Unless a radical change in micro-technology occurs, the clock speed¹ is not likely to get much greater than one per nano-second (Hack, 1989). One way in which the need for ever greater computing resources can be satisfied is to use large numbers of processors.

In response to this perceived need for greater computing power, there has been an increased interest in parallel computing in the meteorological community. The European Centre for Medium Range Weather Forecasting (ECMWF) has held four biennial workshops in the use of parallel processors in meteorology. By 1991 two publications, based on these workshops, have appeared in print (Hoffmann and Snelling, 1988; Hoffman and Maretiš, 1990).

The problem with having many processors share a single memory is that the connections between the processors and the memory become saturated as the number of processors increase. One solution to this problem is to give each processor its own memory. These processors are connected together by communications channels by which the processors can exchange data, as necessary, with one another. This method of communications is often called message-passing. It is very expensive to build such a computer with a connection from every processor to every other processor and therefore each processor can only communicate directly with a limited sub-set of processors.

¹The rate at which a processor executes instructions.

In this thesis, the use of such computers in meteorology will be explored, focusing mainly on global atmospheric models. The thesis will show the importance of communications in parallel algorithms for methods used in present atmospheric models. Work must be distributed over the available processors as equally as is possible. In the paper by Kauranne (1990) some general remarks on inherent parallelism in weather models are made. In this thesis, specific parallel algorithms are described and the results of their implementation on a parallel computer are presented.

1.1 Thesis Overview

This thesis examines the utility of computers built from many processors, each with their own memory and own program. Communications between the processors is via message-passing. A study of the problems involved in implementing some parameterization schemes is made and two schemes are implemented and benchmarked. Two methods for simulation of the large-scale dynamics are examined and implemented. The only major component of forecast models which is not examined is the assimilation scheme, which converts observations into an initial state suitable for the model.

The following section examines, in a broad manner, a number of different parallel architectures. First, the parallel machines presently used in meteorological centres are examined. These computers have small numbers of very powerful vector processors and have a shared memory. The Cray Y-MP is a typical example of such a computer. Second, there follows a discussion on array or data-parallel computers. These computers are naturally programmed using the Fortran 90 array model which is described by Metcalf and Reid, 1989. The final architectural type that will be considered is the MIMD computer. In particular the focus is on the Inmos Transputer

from which the Edinburgh Concurrent Supercomputer (ECS) is built, although the algorithms considered in this thesis could apply to many MIMD computers.

The remaining two sections in this chapter are concerned with a discussion of the meteorological background and the numerical methods used to solve them. One section concentrates mainly on the primitive equations and is intended to provide some background for the other, longer, section on numerical methods. The discussion of numerical methods will provide sufficient detail to understand how the parallel algorithms for them are constructed.

The second chapter outlines the principles used in designing parallel algorithms computers, similar to the ECS, in which processors interact by message passing. The first part of the chapter considers the various different decomposition strategies that can be used to divide a problem so that it can be computed in parallel by many processors. Following on from this, the way in which the separate components of a parallel program communicate and interact with one another is discussed. The machine's topology and the interaction with the algorithm are also considered here. The use of associative operators, of which addition and maximum are examples, is considered. An extended discussion on efficiency for parallel computers is then followed by a short section on input and output from the computer.

Chapter Three examines the problems involved in implementing parameterization schemes on parallel computers. For computers like the ECS this is fairly straightforward as long as the scheme is designed to process single atmospheric columns. Load balancing of these schemes is also considered.

Chapters Four and Five examine methods used to model the atmospheric dynamics. Chapter Four is concerned with the spectral method. The main component of this method, and also that part which causes the difficulties

in a parallel scheme, is the transformation from spectral space to grid-point space and *vice versa*. A section each, in that chapter, is devoted to the fast Fourier transformation (FFT) and the Legendre transform which together make up the spectral transformation. Prior to these, the concept of a pipeline is introduced. Next, a section shows how the components of the spectral transformation are connected together. Following this, the details of the implementation on the ECS are described. The results of this implementation are presented in the penultimate section. The final section of the chapter draws some conclusions and makes some suggestions for further work.

Three-dimensional grid-point models are examined in the Fifth chapter. Results of the implementation of a three-dimensional grid-point model are also discussed in this chapter. For limited area or regional models, the approach taken of using horizontal geometrical decomposition is efficient. In order to increase the number of processors that can be used, algorithms for limited vertical decomposition are presented. Global grid-point models are considered next in the chapter. It is necessary to carry out Fourier transforms in these models and this extra requirement makes it impossible for the ECS to obtain satisfactory speeds. The effects of load-balancing on the Fourier transforms are considered and results of two implementations showing increased efficiencies for the Fourier transforms when load-balancing is used are shown.

The final chapter presents some conclusions and considers what atmospheric modellers require from parallel computers.

Table 1-1: Parallel computing taxonomy

		Instructions	
		Single	Multiple
Data	Single	SISD	MISD
	Many	SIMD	MIMD

1.2 Parallel Architectures

One way in which parallel architectures can be classified was introduced by Flynn (1972), shown in Table 1-1 reproduced from Trew and Wilson (1991)

No example of the MISD (Multiple Instruction, Single Data) has been built as the utility of carrying out several operations on one set of data values is not apparent. The SISD (Single Instruction, Single Data) architecture is the well understood von Neumann processor design. SIMD (Single Instruction, Multiple Data) computers perform the same operation on many data values, vector and array processors are examples of this class. MIMD (Multiple Instruction, Multiple Data) computers can carry out different instructions on many data values, these are machines with both multiple processors and multiple programs.

Parallel computers can also be classified by the way in which their memory is distributed. Existing supercomputers have a small number of processors and a global memory which all processors can access. Access to this memory is normally achieved through a single bus. This architecture has limits, in that access to the single memory will become a bottleneck as the number of processors increases.

An alternative approach is to provide each processor with its own local memory. Rather than information passing between the processors by one processor writing it to memory and another processor reading it from memory, one processor makes a copy of the information and sends that as a message to the other processor.

This section gives a broad overview of a small number of different parallel architectures and where appropriate refers to work done by other authors on parallel algorithms for atmospheric modelling. It is divided into three parts. The first, under the heading of modestly parallel architectures examines the Cray Y-MP, an example of a shared memory multi-processor vector machine. The following subsection examines data-parallel SIMD machines, the examples considered are the AMT DAP (Distributed Array Processor) and the Connection Machine from Thinking Machines Corp. (TMC). Finally the Edinburgh Concurrent Supercomputer (ECS) which is constructed from 400 T800 transputers is discussed as an example of a distributed memory MIMD computer. The final subsection will go into more detail than the rest as the implementations described in the thesis were carried out on this computer. There are, of course, many more computers than those listed above. For a survey of parallel computers as of 1991, see Trew and Wilson (1991).

1.2.1 Modestly Parallel Architectures

One approach taken to obtain greater computing performance, which has been successful in the 1980's, is to build computers with small numbers of very powerful processors. These processors interact through a single shared memory. An example of such a computer is the Cray Y-MP. The Cray Y-MP can have up to 8 vector processors and it is a development of the Cray-1. There has been a further development along this line, resulting in the Y-MP C90 with up to 16 processors.

In 1976 the first Cray-1 was delivered to the Los Alamos National Laboratory. It was the first computer to offer vector chaining and pipelined vector units in hardware. Vector computer architectures are covered in many undergraduate computer science text books, see for example Ibbet (1982). A vector machine is an example of a SIMD machine in which the same instruction is applied to many data, in this case the components of the vector.

The next development of the Cray-1 by Cray Research Inc. (CRI) was the Cray X-MP. The X-MP is a shared memory multi-processor computer, with up to 4 vector processors. The individual processors are developed from the Cray-1 with some other enhancements. An example of a spectral model implemented on an X-MP is described by Dent (1988). The Cray Y-MP is a further development of the Cray X-MP with up to 8 vector processors and a faster clock time allowing peak speeds of approximately three Gigaflops (3×10^9 floating point operations per second). For the U.K. Meteorological Office's (UKMO) forecast model, sustained rates of approximately one Gigaflops have been obtained using 8 processors (Dickinson, 1990).

Many scientific programs have repeated operations on multi-dimensional arrays; in Fortran this is usually implemented by "do" loops. Utilization of the fast vector units in machines such as the Cray-1 is normally done by "vectorising" certain do loops. Compilers have been designed that can recognise the loops which can be safely vectorised. These compilers allow many Fortran programs to run on these machines without requiring modifications. However, to use these computers effectively, the code does need to contain large vectorisable loops. Some restructuring or rewriting of the program may be required so that the program has these large vectorisable loops.

Cray Research Inc. have, with time, developed increasingly more sophisticated compilers for the machines described above, which can recognise and safely vectorise increasingly complex loop constructions, and requiring

less programmer intervention. The present compiler can also divide loops up over multiple processors². A master/slave approach is taken in which the master processor finds out if there are free processors and, if there are, partitions the loop over any free processors and itself. The development of these compilers is described by Fourtney (1990).

Cray technology is mature, providing a compiler which can efficiently run code on multi-processor machines without requiring much programmer intervention. The efficiency of their compilers is made possible by the shared memory nature of the machines; the transfer of information for the loop partitioning between processors is very cheap. The master processor just needs to pass information to the slave processors telling them which part of the loop they should compute and where in memory is the data they require. Unfortunately there are problems with shared memory machines as the number of processors increases.

1.2.2 Data-Parallel Architectures

Now, an architecture with distributed memory but with one source of control is considered. In these architectures there is one program whose statements are executed simultaneously by all processors. These machines are similar to the vector processors considered previously in that they are SIMD machines. However, because of the distributed memory, more attention has to be paid to how the work is divided over the processors. The distribution of the work of the processors is normally termed decomposition. Unlike vector processors, where data is read from memory, processed and then written back, in data-parallel computers, data resides in the memory of each (of

²In addition to vectorising them on each processor.

the many) processors and all processors carry out the same operation on their data. In many algorithms, for example finite difference, it is necessary for processors to access data in neighbouring processor's memories. This is implemented by shift operators, in which the whole array is moved, for example $A(i, j, k)$ would be shifted to $A(i+1, j, k)$. All existing computers of this class allow some processors to ignore any given instruction. Examples of such computers are the AMT Distributed Array Processor (DAP) and the Connection Machine (CM) from Thinking Machine Corporation.

The DAP is an array of 32×32 or 64×64 one bit processors called processing elements (PE). Each processor can have up to 256Kbit of local memory. The PEs are connected to each other in a grid topology with wrap around at the edges of the array i.e. the processors at the north edge of the grid have as northerly neighbours the processors on the south edge. In addition to the PEs the DAP has a Master Control Unit (MCU) which decodes instructions and broadcasts them to the PE's for processing if the operations are parallel or carries them out itself if they are serial operations. For parallel operations, each PE can effectively ignore an operation, depending on the state of a flag on that PE. The DAP is normally hosted by another computer on which compilation and I/O is done. Carver (1990) in his thesis explored the use of the DAP for meteorological modelling and for the dynamics-only models he looked at, reported positive results.

The Connection Machine was originally developed for Artificial Intelligence applications. It is like the DAP in that it is constructed out of one-bit processors though with up to 65,536 of them. Nodes of the machine are constructed from 16 of these and the nodes are connected together in a hypercube topology where each node is the corners of a N-dimensional cube. In a 3 dimensional hypercube each node has 3 links to other nodes and has 8 nodes in total. In the CM-200, pairs of nodes (i.e. 32 of the one-bit processors) can have a floating point co-processor which allows the nodes to have

a respectable computing speed. In this case the programmer would think of the machine as consisting of up to 2048 processing nodes. Like the DAP each processor can carry out a broadcast instruction or effectively ignore it. Like the DAP the CM relies on the host computer for I/O, however unlike the DAP it does not have a MCU, instead it relies on the host computer to decide which instruction should be done in parallel on the CM and which should be done serially on the host computer. A far more detailed description of the CM can be found in the technical document released by Thinking Machines Corp. (1991).

1.2.3 MIMD Architectures

The third class of Flynn's taxonomy is the MIMD computer. One example of this computer has already been mentioned, the shared memory Cray Y-MP. This section will consider MIMD computers with their own local memory which communicate by message-passing. The example considered is the Transputer designed by Inmos.

Inmos was founded in 1977 by the then Labour government. It is presently owned by SGS-Thompson. In 1985, Inmos released the T414 Transputer. Inmos coined the word Transputer to mean a computer on one piece of silicon. On one piece of silicon the T414 has 2K bytes of memory, four bi-directional communications links and a central processing unit (CPU). The chip also has an interface to external memory which allows extra memory chips to be used. Each link can transfer data concurrently with CPU operation. However the CPU needs to initiate the transfers. The processor is designed to support many processes running in parallel. A process will run for some time and then halts and another process is run (if it is ready). This context switching is very fast on the Transputer and is one of its strengths.

Initially the only language available for the Transputer was Occam. Occam has features in the language to support message-passing and running of parallel processes. See Jones (1987) and Bowler *et al.* (1987) for a description of the language. In fact the Transputer was originally designed as an Occam processor. More standard languages such as Fortran were released later; in these languages, input and output of messages is done by subroutine calls.

Floating Point Systems (FPS) designed a supercomputer built from many computing nodes joined together in a hypercube network. The compute nodes consisted of a T414 Transputer and a vector processing unit. This architecture was examined by Snelling and Tanqueray (1988) and Grønas (1988) for its usefulness for meteorological modelling. The system was unsuccessful for two reasons; first the node had communications that were too slow for the fast vector unit; second FPS did not initially provide a Fortran compiler which made acceptance by many organisations difficult.

In 1987, Inmos released the T800 Transputer, a development of the T414. It had a Floating Point Unit (FPU) and an additional 2K bytes of on-chip memory (see Figure 1-1). The Central Processing Unit (CPU) sends data to the FPU, which processes it and returns the data to the CPU. The CPU can do other tasks while the FPU is busy.

Each of the transputer's links can be joined to another transputer's links, the networks formed can be quite complex. For more information on the Transputer hardware see the technical documentation from INMOS Limited (1988).

The Edinburgh Concurrent Supercomputer (ECS) is a machine built from 400 T800 Transputers. The machine was built by Meiko Scientific and delivered in 1987. Each processor has at least 4 Mega-bytes (4M bytes) of memory in addition to the on-chip memory of the Transputer. The processors

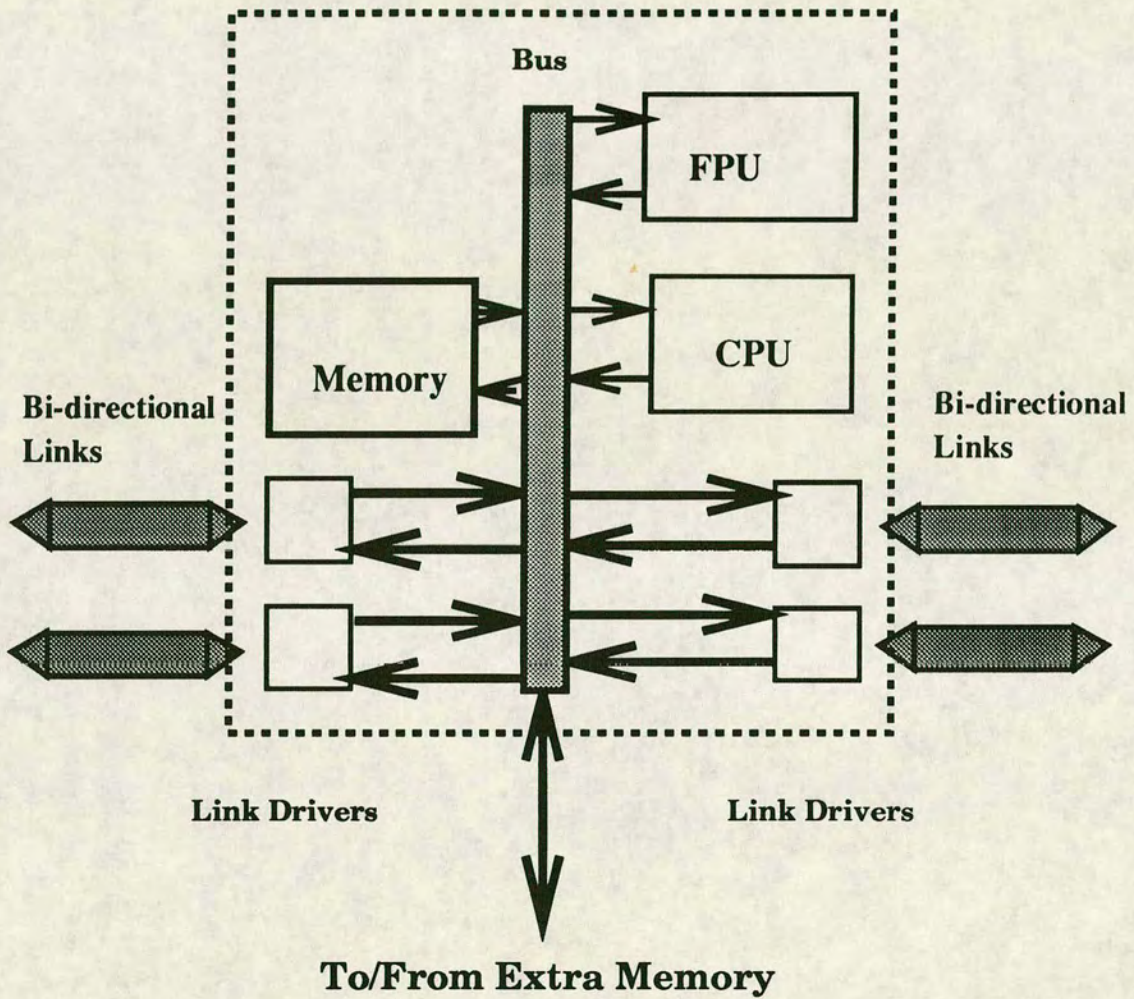


Figure 1-1: The T800 Transputer

A schematic diagram of the T800 processor showing the major components.

are connected to switching chips which allow a wide range of inter-processor connections to be made. The machine is divided into several partitions called domains. Each domain is allocated to a single user who has sole use of all the processors in the domain. He or she may choose to use any number of these processors in the domain. These domains range in size from 1 to 131 processors, with some domains having hardware to support graphics. The system has its own filing system and runs a version of Unix³. See Bowler *et al.* (1989) for a description of the ECS and some of the applications as of 1988; by 1990 the list of applications running on the ECS had increased and the ECS had reached the form described above, see Wallace (1990) for more details. Also see Wallace (1988) for some early applications on the ECS and on the ICL DAP at Edinburgh.

Table 1-2 shows the number of operations per second for two typical patterns in a grid-point model with the data being stored in external memory. The table shows that the T800 can manage about a half a million floating point operations per second (a half Mflops). Figure 1-2 shows the communications speed in bytes per second for various different message lengths and different configurations. The T800 is capable of communication speeds of up to 2 million bytes per second. These results are discussed in the relevant chapters.

³Unix is a trademark of AT&T.

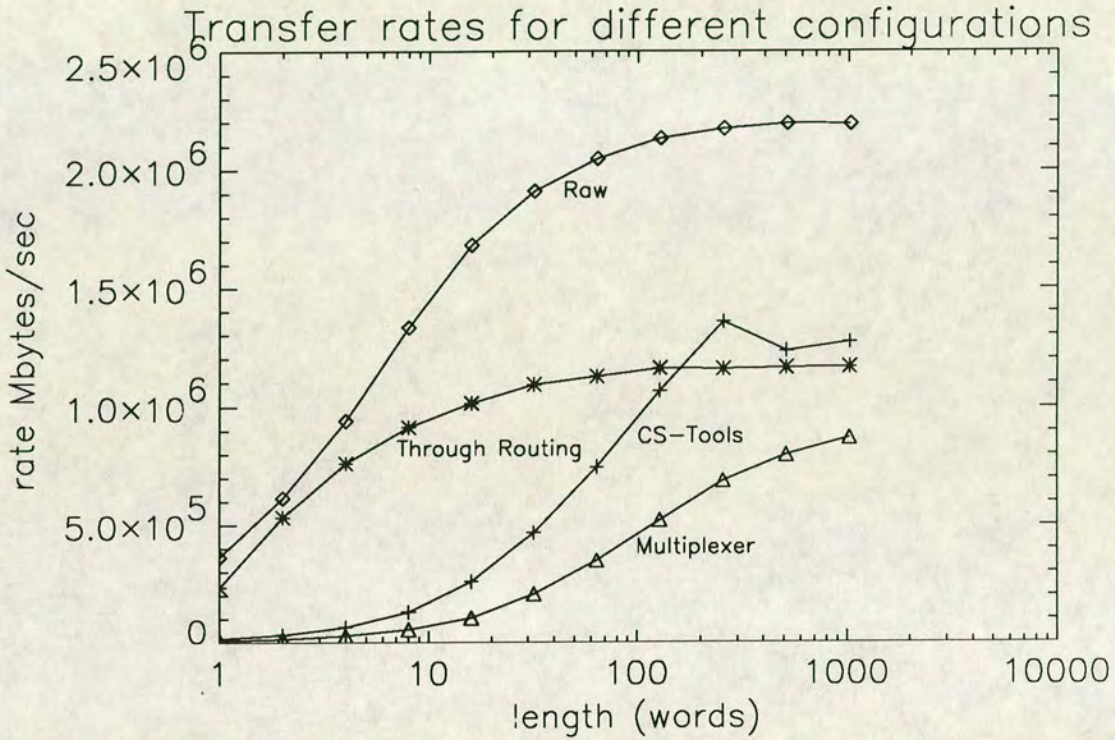


Figure 1-2: T800 communicating speeds

Four different results are shown here. The line labelled “raw” is the speed between two Transputers. The line labelled “cs-tools” is the speed achieved by a Meiko supplied library in a Fortran program. This is discussed in Chapter 4. The line labelled “through routing” is when a message is routed through another processor, that processor is doing nothing else. The line labelled “multiplexer” is the speed when a software multiplexer/demultiplex is used. See Chapter 5 for an explanation of the latter two cases.

Table 1-2: T800 compute speeds

Operation	$A(I) = B(I) + C(I)$	$A(I) = r * (B(I) + C(I))$
No. per Sec	297619	240384
Flops	297619	480768

This table shows the speed at which the T800 can compute. The speeds for two different, indexed, operations are shown. One is when the element by element sum of two vectors is formed, while the other is when this sum is multiplied by a scaler before being stored back in memory.

1.3 Physical Meteorological Background

This section will set out the physical basis to the meteorological work, while the following section will discuss the numerical methods used in meteorology. First the dry primitive equations will be detailed. Following this the effects of water vapour on the atmosphere will be discussed. General introductions to atmospheric physics can be found in the books by Holton (1979) and Houghton (1986).

1.3.1 Dry Primitive Equations

This subsection will present the primitive equations. For details of the derivations the reader is referred to Haltiner and Williams (1980) and Gill (1982), in particular Chapter 1 of the former and Chapters 3 and 4 of the latter. Many basic meteorological textbooks show that,

$$\frac{DV}{Dt} = -1/\rho \nabla p - 2\Omega \times V + g + F. \quad (1.1)$$

where V is the velocity, ρ the density of air, p the pressure, F any frictional forces, Ω is the rotation vector of the planet, and g is $g_a - (\Omega \times (\Omega \times r))$, g_a

being the gravitational acceleration and $\Omega \times (\Omega \times r)$ is the centrifugal force due to the rotation of the planet. The above equation is in a Lagrangian frame of reference where the frame of reference moves with fluid particle. When the frame of reference is fixed ^{in a Cartesian co-ordinate system} the operator $\frac{D}{Dt}$, in Equation 1.1 is equivalent to $\frac{\partial}{\partial t} + u\frac{\partial}{\partial x} + v\frac{\partial}{\partial y} + w\frac{\partial}{\partial z}$. See Batchelor (1967) Chapter 2 for an extended discussion of these two ideas.

Conservation of fluid mass is expressed by,

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho V) = 0. \quad (1.2)$$

This can be interpreted as the change of mass in a small volume being the same as the total inflow or outflow into this small volume.

The atmosphere is normally approximated by a perfect gas which satisfies the following equation of state,

$$\rho = \frac{p}{RT} \quad (1.3)$$

with R being the gas constant for dry air and T the temperature. The first law of thermodynamics is,

$$Q = c_v \frac{DT}{Dt} - 1/\rho \frac{Dp}{Dt} = c_p T \frac{D \ln \theta}{Dt}. \quad (1.4)$$

In this equation, θ is the potential temperature given by $(p/p_0)^{\kappa} T$ and Q is the rate at which heat is added to a fluid parcel. For adiabatic processes $Q = 0$ and thus $\frac{D\theta}{Dt} = 0$.

For the dry atmosphere there are six variables and six equations, thus the system is complete. In practice the observations are not perfect and so the initial state, from which the model is started, has errors. As the model is integrated forwards in time these errors grow and cause the model state to diverge increasingly from reality. The growth of the initial errors in the ECMWF forecast model was investigated by Lorenz (1982).

This set of equations support a great variety of wave behaviours, in a wide variety of length and time scales. In addition to the linear, and reversible, wave behaviour there are non-linear processes occurring in the atmosphere, for example the flux of momentum, moisture and other quantities from tropical regions to polar regions by eddies in the atmospheric flow.

The behaviour of these linear wave solutions can be obtained by linearising the equations about some state and considering small perturbations to that state. As will be discussed in the next section, an explicit numerical scheme will be unstable if $\left| \frac{c\Delta t}{\Delta x} \right| > 1$, where c is the phase speed of the fastest modelled travelling waves in the problem, Δx and Δt are the time and space intervals between points in the numerical approximation respectively. This is the Courant-Friedrichs-Levy (CFL) condition. If there exist wave solutions to the linearised equations with high phase velocities, a short timestep will be needed for numerical stability. In many atmospheric conditions the fastest moving waves (sound waves) have very little energy and so have negligible impact on the solution.

The co-ordinate system usually chosen for atmospheric modelling is one in which the direction of gravity is locally vertical. The horizontal surfaces are then surfaces of equal gravitational potential. These surfaces are almost spherical and it is a good approximation to treat them as such. The atmosphere is a thin layer compared to the radius of the earth and another approximation that can be made is to neglect the variation in radius (the shallow atmosphere approximation). Considering the Coriolis term ($2\Omega \times V$) in equation 1.1, it can be shown (Holton, 1979) that if the geopotential surfaces are assumed to be spherical and the shallow atmosphere approximation is made then terms proportional to $\cos(\theta)$ in the Coriolis terms must be neglected in order to conserve angular momentum. This is valid as the terms proportional to $\cos(\theta)$ are small. Gill (1982), sections 6.4 and 6.5, shows that for plane progressive gravity waves, with the effects of rotation

being ignored that; $\frac{\partial p'}{\partial z} = -(m^2/(k^2 + l^2 + m^2))g\rho'$, with p' and ρ' being the pressure and density perturbations from a hydrostatic reference state. k, l and m are the x, y and z components of wavenumber. For the perturbations to be hydrostatic, $k^2 + l^2 \ll m^2$. This implies that $(L^2/H^2) \gg 1$, with L and H being the horizontal and vertical scales respectively. For synoptic scales this is true and therefore the vertical component of acceleration ($\frac{Dw}{Dt}$) in equation 1.1 can be neglected. When all these approximations are carried out, equation 1.1 can be rewritten as two equations:

$$\frac{DV_H}{Dt} = -1/\rho \nabla p - f\mathbf{k} \times V_H + F_H \quad (1.5)$$

$$\frac{1}{\rho} \frac{\partial p}{\partial z} = -g. \quad (1.6)$$

In the above equation, f is $2\Omega \sin \theta$ with θ being the latitude. The horizontal component of velocity (V_H) will from now on be referred to as V .

1.3.2 Hydrodynamic Cycle

The effects of water vapour on the atmospheric flow are considerable. The large releases of energy through latent heat when water vapour changes phase are responsible for driving many processes in the atmosphere. The accurate prediction of rainfall is one of the public's requirements from weather forecasting.

Water vapour is normally measured by specific humidity, q , which is the mass of water vapour per unit mass of air, and it is conserved following fluid motion except for condensation/evaporation and molecular diffusion. For most of the atmosphere the effects of molecular diffusion can be neglected. This gives,

$$\frac{Dq}{Dt} = E \quad (1.7)$$

where E is a term representing condensation/evaporation from the atmosphere. A modification to the equation of state in Equation 1.3 is necessary. The density is now given by,

$$\rho = \frac{p}{RT_v} \quad (1.8)$$

T_v is the virtual temperature which is $T_v = T(1+0.61q)$ and is the temperature that dry air would have if it were the same density as the wet air.

A source of heat energy occurs when water vapour condenses to form precipitation, while a sink is the energy required to evaporate liquid water or ice. These will form one of the sources of energy on the left hand side of Equation 1.4. Condensation will occur when the humidity is greater than the saturation humidity. Occasionally the atmosphere can become supersaturated although many models neglect this.

1.3.3 Co-ordinates

Computations cannot be done with primitive equations in the general vector form, some co-ordinates need to be introduced. This subsection will do that. First the vertical co-ordinate will be examined.

Introduce a generalised vertical co-ordinate ζ which is monotonic and single valued with respect to z ,

$$\zeta = \zeta(z, x, y, t) \quad \text{or} \quad z = z(\zeta, x, y, t). \quad (1.9)$$

It can be shown that in this co-ordinate Equation 1.5 becomes,

$$\frac{DV}{Dt} = -1/\rho \nabla_{\zeta} P - \nabla_{\zeta} \Phi - f \mathbf{k} \times V + F \quad (1.10)$$

with Φ being the geopotential height ($g \times z$). The hydrostatic approximation then becomes,

$$1/\rho \frac{\partial p}{\partial \zeta} + \frac{\partial \Phi}{\partial \zeta} = 0, \quad (1.11)$$

which implies that

$$\Phi = \Phi(0) - \int_{\zeta_0}^{\zeta} \frac{1}{RT} \frac{\partial \ln P}{\partial \zeta} d\zeta. \quad (1.12)$$

The Earth's surface occurs when $\zeta = \zeta_0$ and $\Phi(0)$ is the height of the surface, above some constant reference level times the gravitational constant. The continuity equation, 1.2, becomes

$$\frac{d(\ln \frac{\partial p}{\partial \zeta})}{dt} + \nabla_{\zeta} \cdot V + \frac{\partial \dot{\zeta}}{\partial \zeta} = 0. \quad (1.13)$$

Various vertical co-ordinates have been used by meteorologists; these include θ (potential temperature), $\log p$ and p . One popular choice is σ which is defined as $\sigma = \frac{p}{p_*}$, where p_* is the surface pressure. This choice of co-ordinates has the advantage that co-ordinate surfaces do not intersect

the surface of the Earth. More recently hybrid co-ordinates have been used. These define ζ as,

$$\zeta = A(\eta) \frac{p}{p_*} + B(\eta) p_0. \quad (1.14)$$

If values of A and B are chosen correctly then this co-ordinate will behave like σ near the surface and like p high in the atmosphere. These co-ordinates are used in the grid-point model in Chapter 5.

Haltiner and Williams (1980) show that Equation 1.5 in spherical co-ordinates is,

$$\frac{Du}{Dt} = -\frac{1}{a \cos \theta \rho} \frac{\partial p}{\partial \lambda} + \left(f + \frac{u \tan \theta}{a}\right)v + F_\lambda \quad (1.15)$$

$$\frac{Dv}{Dt} = -\frac{1}{a \rho} \frac{\partial p}{\partial \theta} - \left(f + \frac{u \tan \theta}{a}\right)u + F_\theta \quad (1.16)$$

where λ and θ are the longitude and latitude respectively.

1.4 Numerical Methods For Meteorology

Global atmospheric models can be conceived as having two parts; a simulation of the large-scale atmospheric flow using a numerical approximation to the equations of motion, normally the primitive equations, and the simulation of the effect on the resolved variables by the unresolved flow termed parameterization. Existing NWP models have a length scale of approximately 100km in mid-latitudes while the length scale for climate models is approximately 500km.

All numerical methods used in meteorology make some kind of approximation in that model variables are stored at discrete values of space and time co-ordinates. The partial differential equations with continuous space and time fields and infinite degrees of freedom are replaced by variables

with finite degrees of freedom. There are two methods widely used in the meteorological community: the grid-point and spectral methods, described in Subsections 1.4.2 and 1.4.1 respectively. However, because of the non-linear nature of the equations, unresolved scales of motion will have effects on the resolved scale. This will be illustrated by considering the one dimensional advection equation,

$$\frac{\partial A}{\partial t} + u \frac{\partial A}{\partial x} = 0. \quad (1.17)$$

Here A is an arbitrary scalar. For a numerical model, the values will be averaged over a length scale L to form \bar{A} and \bar{u} and the numerical equations will be used to predict the time evolution of these quantities. Formally then

$$\bar{A} = \int_0^L A(x) dx, \quad \bar{u} = \int_0^L u(x) dx. \quad (1.18)$$

If A' and u' are defined as the difference from the means, then using this definition, $A = \bar{A} + A'$ and $\bar{A}' = 0$, and similarly for u . Using these definitions Equation 1.17 can be rewritten as,

$$\frac{\partial(\bar{A} + A')}{\partial t} + (\bar{u} + u') \frac{\partial(\bar{A} + A')}{\partial x} = 0. \quad (1.19)$$

If the mean of this equation is taken to obtain an equation to predict the evolution of \bar{A} and \bar{u} , the following is obtained,

$$\frac{\partial \bar{A}}{\partial t} + \bar{u} \frac{\partial \bar{A}}{\partial x} + \overline{u' \frac{\partial A'}{\partial x}} = 0. \quad (1.20)$$

Equation 1.20, in addition to terms involving the meaned variables, has a term to take account of the effects of sub-grid fluctuations on the mean flow. The sub-grid scale needs to be parameterised in terms of the mean variables⁴. In large-scale models this effect is normally parameterised by

⁴Which is all the numerical scheme will have any information about.

“eddy” diffusion which in many models for a variable A , is modelled by $k_A \nabla^{2n} A$. The value of the constant k_A can be different for different variables. The value of n can also be different for different variables. For some criticisms of this parameterization see McIntyre (1989).

This \square occurs in many atmospheric models; processes whose length or time scales are unresolved by the model have effects on resolved variables. These unresolved processes need to be modeled in terms of the resolved variables in order to simulate their effects on them. This process is called parameterization. Some processes that require this are convection, radiation and surface exchanges.

1.4.1 The Spectral Method

This subsection describes the spectral method as it is used in meteorology. The aim is not to go into great detail. Instead, the salient features of the method from the point of view of implementing it on a parallel computer will be stressed. See Machenhauer (1979) for an extended discussion on the method.

In the spectral method, variables are represented by a truncated sum of orthogonal functions. For global meteorological models, the natural functions to choose are the spherical harmonics, $Y_l^m(\mu, \lambda)$, where μ is the sine of latitude and λ is the longitude. These harmonics are eigenfunctions of the spherical Laplacian; a description of their properties can be found in Arfken (1985). A variable Z is given by the following relationship,

$$Z(\lambda, \mu) = \sum_{m=-M}^M \sum_{l=|m|}^N Z_l^m Y_l^m(\lambda, \mu). \quad (1.21)$$

For an exact representation of the field, M and N would need to be equal to infinity. In meteorology two different truncations are used.

Triangular $N = M$. This truncation is denoted by TM, i.e. T21 is a triangular truncation with $M = 21$. The truncation is called triangular as when the allowed positive values of l and m are plotted they form a triangle.

Rhomboidal $N = M + |m|$. This truncations is denoted by RM, i.e. R15 is a rhomboidal truncation with $M = 15$. When the allowed positive values of l and m are plotted they form a rhombus.

The spectral harmonic, Y_l^m is given by the following relationship.

$$Y_l^m(\lambda, \mu) = P_l^m(\mu)e^{im\lambda}. \quad (1.22)$$

$P_l^m(\mu)$ is the associated Legendre function normalised to unity. The spherical harmonic has the following properties,

$$\begin{aligned} \frac{\partial Y_l^m}{\partial \lambda} &= imY_l^m \\ (\mu^2 - 1) \frac{\partial Y_l^m}{\partial \mu} &= l\epsilon_{l+1}^m Y_{l+1}^m - (l+1)\epsilon_l^m Y_{l-1}^m \\ \epsilon_l^m &= [(l^2 - m^2)/(4l^2 - 1)]^{\frac{1}{2}} \\ \frac{1}{4\pi} \int_0^{2\pi} \int_{-1}^1 Y_l^m Y_{l'}^{m'*} d\mu d\lambda &= \delta_{l,l'} \delta_{m,m'} \end{aligned} \quad (1.23)$$

with $\delta_{m,m'}$ being the Kronecker delta which is 1 if both indices are the same and 0 otherwise. The last property in Equation 1.23 is the orthogonality relationship. In order to show how the spectral method works in practice the non-divergent barotropic ^{∇_σ · \hat{r} = 0} equation will be considered and the triangular truncation will be used. This is a very simple equation but illustrates the important features of the method. For a parallel algorithm the details of the transformation, for this simple equation, will be the same as that for the primitive equations. The treatment that follows is based on that of Washington and Parkinson (1986).

The basic equation is,

$$\frac{\partial \zeta}{\partial t} = -V \cdot \nabla(\zeta + \psi). \quad (1.24)$$

Here ζ is the vorticity, $k \cdot \nabla \times V$. Furthermore the horizontal wind components U and V are defined as $u \cos \theta$ and $v \cos \theta$ (with θ being the latitude). In addition to this the non-divergence approximation is made, ($\nabla \cdot V = 0$) and setting $\zeta = \nabla^2 \psi$, with ψ being the streamfunction, the diagnostic equation for V is,

$$V = k \times \nabla \psi. \quad (1.25)$$

The prognostic equation, using these approximations, for ψ is

$$\frac{\partial \nabla^2 \psi}{\partial t} = -\frac{1}{a \cos^2 \phi} \left[\frac{\partial U \nabla^2 \psi}{\partial \lambda} + \cos \phi \frac{\partial V \nabla^2 \psi}{\partial \phi} \right] - \frac{2\Omega V}{a}. \quad (1.26)$$

From Equation 1.21 the following values are obtained,

$$\begin{aligned} \psi &= a^2 \sum_{m=-M}^M \sum_{l=|m|}^M \psi_l^m Y_l^m \\ U &= a \sum_{m=-M}^M \sum_{l=|m|}^M U_l^m Y_l^m \\ V &= a \sum_{m=-M}^M \sum_{l=|m|}^M V_l^m Y_l^m. \end{aligned} \quad (1.27)$$

The linear parts of Equation 1.26 are straightforward to compute. The difficult part is the computation of the non-linear terms ($U \nabla^2 \psi$ and $V \nabla^2 \psi$). Using Equation 1.27 to carry out this computation would involve multiplying two series together—an extremely expensive procedure in terms of computer time. Orzag (1970) observed that transforming from spectral space to grid point space and computing the non-linear products there, would be more efficient than multiplying the two series together.

The computation of $V(\lambda, \mu)$ can be done in two stages, the first being to calculate $V^m(\mu_k)$ at latitudes μ_k using,

$$V^m(\mu_k) = \sum_{n=|m|}^M V_n^m P_n^m(\mu_k). \quad (1.28)$$

When this computation has been carried out, values of V are calculated at longitudes λ_j using,

$$V(\lambda_j, \mu_k) = \sum_{m=-M}^M V^m(\mu_k) e^{im\lambda_j}. \quad (1.29)$$

As the values of $V(\lambda_j, \mu_k)$ are real, being values in grid-point space, then $V^m = V^m(\mu_k)^*$, and so the computation of $V^m(\mu_k)$ need only be done for positive values of m . The set of values λ_j are chosen to be equally spaced and the number of values are such as to permit a fast Fourier transform to be carried out.

Before doing this it is necessary to compute the values of U_l^m and V_l^m . They can be diagnosed using Equation 1.25 and the relationships given in Equation 1.23 to give, where a is the radius of the earth,

$$V_l^m = a m i \psi_l^m \quad (1.30)$$

$$U_l^m = a [(l-1)\epsilon_l^m \psi_{l-1}^m - (l+2)\epsilon_{l+1}^m \psi_{l+1}^m]. \quad (1.31)$$

The diagnostic relationship for U_l^m (Equation 1.31) involves values of ψ_l^m at different values of l from U_l^m . This is discussed on page 142 of Chapter 4. An alternative formulation is possible (private communication, M. Blackburn) and this will now be discussed as it is used in Chapter 4 to remove the need for some communications and in the model considered in that chapter.

The problem is to compute the Legendre transform of a derivative. There are two derivatives to consider:

$$A(\lambda, \mu) = \frac{\partial F(\lambda, \mu)}{\partial \lambda}$$

$$B(\lambda, \mu) = (1 - \mu^2) \frac{\partial F(\lambda, \mu)}{\partial \mu}.$$

From Equation 1.21, for the triangular truncation, the following is obtained,

$$F = \sum_{m=-M}^{m=M} \sum_{l=|m|}^M F_l^m Y_l^m(\lambda, \mu)$$

$$= \sum_{m=-M}^{m=M} \sum_{l=|m|}^M F_l^m P_l^m(\mu) \exp(im\lambda).$$

Using this it can then be shown that

$$\begin{aligned} A(\lambda, \mu) &= \frac{\partial F}{\partial \lambda} \\ &= \sum_{m=-M}^{m=M} \sum_{l=|m|}^M im F_l^m Y_l^m(\lambda, \mu). \end{aligned} \quad (1.32)$$

Therefore A_l^m is equal to imF_l^m .

Now considering the $B(\lambda, \mu)$ term.

$$\begin{aligned} B(\lambda, \mu) &= (1 - \mu^2) \frac{\partial F}{\partial \mu} \\ &= (1 - \mu^2) \sum_{m=-M}^{m=M} \sum_{l=|m|}^M (F_l^m \frac{\partial P_l^m(\mu)}{\partial \mu} \exp(im\lambda)). \end{aligned} \quad (1.33)$$

So, to compute the transformation of the μ derivative from spectral space to grid-point space, multiply by the *derivative* of the Legendre polynomial rather than by the Legendre polynomial. As will be shown in Chapter 4, a model using this method will require less communications than one that computes U and V using Equations 1.31 and 1.30.

Most of the details of the computations in grid point space are irrelevant to the needs of this thesis; the important point is that all computations at a point (λ_j, μ_k) involve only values defined at that point and not at other points. The vertical interactions are more complex and will be discussed briefly in Chapter 4.

Having transformed the variables into grid point space, the various products will need to be transformed into spectral space. This transform can be carried out using the orthogonality relationship given in Equation 1.23 to obtain (for any variable X_l^m),

$$X_l^m = \frac{1}{4\pi} \int_{-1}^1 \int_0^{2\pi} X(\mu, \lambda) Y_l^{m*}(\mu, \lambda) d\mu d\lambda. \quad (1.34)$$

It is possible to rewrite this equation as,

$$X_l^m = \frac{1}{2} \int_{-1}^1 P_l^m(\mu) \underbrace{\left[\frac{1}{2\pi} \int_0^{2\pi} X(\mu, \lambda) \exp(-im\lambda) d\lambda \right]}_{\text{Fourier}} d\mu. \quad (1.35)$$

The Fourier part of this integral can be evaluated exactly by a discrete Fourier transform with at least $3M + 1$ points at equally spaced longitudes. If the number of points in a longitudinal row is chosen to be a power of 2 or a product of a small numbers of prime numbers then this transformation can be done very efficiently by using the fast Fourier transform. Defining the result of this integral as $X^m(\mu)$, it is now necessary to evaluate,

$$X_l^m = \frac{1}{2} \int_{-1}^1 P_l^m(\mu) X^m(\mu) d\mu. \quad (1.36)$$

This integral can be evaluated exactly for the quadratic terms (in grid-point space), except for rounding error, by Gaussian quadrature. For this to be done the latitudes (μ_k) must be chosen to be certain special values and there will be, at least, $(3M + 1)/4$ of them. These values are normally termed Gaussian latitudes and they are roots of Legendre functions. See Machenhauer and Rasmussen (1972) for details of this.

Having transformed the tendencies into spectral space, the computation of the values of the spectral coefficients can then be done.

To summarise, the spectral method involves several stages.

1. Compute values of U_l^m and V_l^m from the prognostic variables. The computation of U_l^m will require spectral coefficients at $l + 1$ and $l - 1$. This stage is optional.
2. Compute $X^m(\mu_j) = \sum_{l=|m|}^m P_l^m(\mu_j) X_l^m$ for all variables that require transforming. If the velocities have not been computed, the computation of $X^m(\mu_j) = \sum_{l=|m|}^m \frac{\partial P_l^m(\mu_j)}{\partial \mu} X_l^m$ will be required for some of the variables.

3. Compute $X(\mu_j, \lambda_j) = \sum_{m=1}^{(3M+1)} X^m(\mu_k) e^{im\lambda_j}$. The values of $X(\mu_k, \lambda_j)$ are real.
4. In grid point space, compute the set of non-linear products, $\{Z\}$ from the transformed set of variables $\{X\}$. If required, compute any of the parameterised processes, an example of which is convection.
5. Compute $Z^m(\mu_j) = \sum_{k=1}^{(3M+1)} Z(\mu_j, \lambda_k) e^{-im\lambda_k}$
6. Compute $Z_1^m = \sum_{j=1}^{(3M+1)/2} G(\mu_j) Z^m(\mu_j) P_1^m(\mu_j)$. $G(\mu_j)$ is the Gaussian weight.
7. Using the set of the time tendencies, $\{Z_1^m\}$ compute new values of the prognostic variables.

1.4.2 Grid-point Methods

In grid point methods, a variable is given values at only a finite set of points. The value of the variable is predicted only at these points. Compare this to the previous subsection where variables were defined at all points on the globe but in terms of a finite number of orthogonal functions. Much of the analysis for grid point methods is based on the Taylor series,

$$f(x \pm \Delta x) = f(x) \pm f'(x)\Delta x + f''(x)\frac{\Delta x^2}{2!} \dots (\pm 1)^n f^{(n)}(x)\frac{\Delta x^n}{n!}. \quad (1.37)$$

The derivative $f'(x)$ can be approximately computed using this series by the centered difference.

$$\frac{f(x + \Delta x) - f(x - \Delta x)}{2\Delta x} = f'(x) + \frac{f^{(3)}\Delta x^2}{3!} + \dots \quad (1.38)$$

This equation can be rewritten to show that the approximation is second order accurate in Δx ,

$$f'(x) = \frac{f(x + \Delta x) - f(x - \Delta x)}{2\Delta x} + O(\Delta x^2) \quad (1.39)$$

There are many other ways of approximately computing derivatives, some of them more accurate. The CFL condition will be obtained by analysing the one-dimensional linear advection term. Following this, the von Neumann method for stability analysis will be explained. Most grid-point meteorological models use staggered grids, these will be discussed. Much of the discussion here can be found and considerably amplified in Haltiner and Williams (1980).

Turning now to the advection equation,

$$\frac{\partial F}{\partial t} + c \frac{\partial F}{\partial x} = 0. \quad (1.40)$$

The analytic solution to this equation is $F = f(x - ct)$ and is the initial disturbance at time $t = 0$ propagating with speed c . Next, a numerical approximation to Equation 1.40 is made. Using the centered difference in Equation 1.39 to approximate the derivative, and introducing the notation $F_{m,n}$ for $F(m\Delta t, n\Delta x)$, gives the following approximation to Equation 1.40,

$$\frac{F_{m+1,n} - F_{m-1,n}}{2\Delta t} + \frac{c}{2\Delta x} [F_{m,n+1} - F_{m,n-1}] = 0 \quad (1.41)$$

To analyse stability, a single harmonic component is substituted for $F_{m,n}$ i.e. $F_{m,n} = B^{m\Delta t} \exp(in\Delta x)$. If $|B|$ is greater than 1 then the solution exponentially grows with time and is unstable. Making this substitution gives, after a little manipulation.

$$B^{2\Delta t} + 2iB^{\Delta t}\sigma - 1 = 0 \quad \sigma = \frac{c\Delta t}{\Delta x} \sin(\Delta x\nu). \quad (1.42)$$

This has a solution for B , $B^{\Delta t} = -i\sigma \pm (1 - \sigma^2)^{1/2}$. If $|\sigma| \leq 1$ then the root is real and the magnitude of B is 1, therefore the solutions are nonamplifying and stable. There are two solutions for B , giving two wave solutions to the finite difference scheme, the analytical solution has only one solution. When

$|\sigma| > 1$, then $|B^{\Delta t}| > 1$ for at least one of the two solutions, the solution is unstable as it grows exponentially with time. For σ to be less than 1 then $|\frac{c\Delta t}{\Delta x}| \leq 1$. This is normally called the CFL condition. Physically only points within a distance $c\Delta t$ of a point can interact with it. If a numerical scheme violates this condition then the scheme is no longer representing physical reality and problems are likely to occur.

The CFL condition above was obtained by examining the behaviour of a single harmonic. If the amplification factor (B) has a magnitude greater than one, then the magnitude of the harmonic will grow exponentially with time. If a numerical scheme has any harmonic components whose amplification factor is greater than one then this scheme will be unstable. The von Neumann method consists of analysing the behaviour of all harmonic components and computing amplification factors for them; if any are greater than one then the scheme is unstable. When applying this method to non-linear schemes, there will be interactions between the harmonics. Such interactions are complex and difficult to compute. Normally the analysis is simplified by linearising the scheme and then doing the von Neumann analysis on the resultant linearised scheme.

In a global longitude-latitude grid, as the meridians approach the poles the physical distance between the grid-points becomes smaller. The CFL criterion would then require a very short time-step in order that the scheme stays stable over the entire globe. One way to avoid this short time-step and thus allow a longer time-step is to carry out the von Neumann analysis on a linearised version of the model to compute the amplification factor for all wavenumbers in any latitudinal row due to the numerical scheme. If any wavenumber in a row is unstable then it is necessary to transform from grid-point space to wavenumber space, using the Fourier transform. When in wavenumber space wavenumbers with amplification factors greater than one have the value of the wavenumber divided by the amplification factor

or set to zero depending on the specifics of the scheme being used. After this has been done, the transform from wavenumber to grid-point space is carried out.

The primitive equations, 1.5 and 1.6 can be divided into two parts; adjustment and advection. Advection refers to the left hand part of Equation 1.5, the non-linear parts of the primitive equations. Equation 1.6 and the rest of Equation 1.5 is the adjustment part. The two parts have different stability criteria. In the advection equation the maximum speed is approximately 100ms^{-1} while in the adjustment there exist wave solutions (gravity waves) with phase speeds of approximately 300ms^{-1} . Accurate numerical schemes for the advection stage are complex and time consuming. Gadd (1978); Gadd (1980) showed how a numerical scheme could solve the two parts of the equation separately, using different timesteps for each part. Normally the adjustment timestep is three times as short as the advection timestep. Therefore the adjustment numerical scheme is carried out three times before computing the effects of the advection scheme.

To complete this subsection, staggered grids will now be considered. Consider the linear equations,

$$\frac{\partial u}{\partial t} = -g \frac{\partial h}{\partial x} \quad ; \quad \frac{\partial h}{\partial t} = -H \frac{\partial u}{\partial x} \quad (1.43)$$

where g and H are constants. Equations 1.43 are the shallow water equations neglecting the non-linear terms. If centered differences are used to compute approximately the differential equations then the following equations are obtained,

$$\frac{\partial u_j}{\partial t} = -g \frac{h_{j+1} - h_{j-1}}{2\Delta x} \quad , \quad \frac{\partial h_j}{\partial t} = -H \frac{u_{j+1} - u_{j-1}}{2\Delta x} \quad (1.44)$$

The grid contains two solutions which do not interact with each other. One solution will consist of the values of u_j at even sites and the values of h_j at

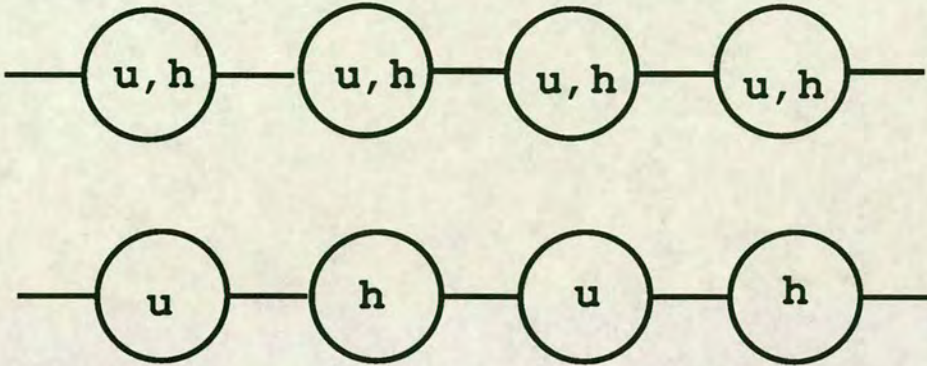


Figure 1-3: One dimensional grids

The top grid is the original grid with two separate solutions. The lower grid is the staggered grid with only one solution.

odd sites. As the two solutions are independent, nothing is lost by dispensing with one and a saving of half the computation time will be obtained. A new grid will be obtained with u at even values of j and h at odd values of j . Figure 1-3 shows the original grid and the staggered grid.

For large-scale atmospheric flows, a two dimensional grid is needed. Figure 1-4 reproduced from Messinger and Arakawa (1975) shows various possible grids labeled A to E.

When implementing a staggered grid on a computer there are several ways in which the $j + \frac{1}{2}$ and $i + \frac{1}{2}$ sites could be mapped into the program arrays. The model implemented in Chapter 5 used the "B" grid and sites labelled by $(i + \frac{1}{2}, j + \frac{1}{2})$ (the points where the velocities were kept) were stored in an array at points labelled by (i, j) . The model kept all the remaining variables at points labelled by (i, j) and they were also stored in arrays at points labelled by (i, j) . To compute the centred differences, an example of which

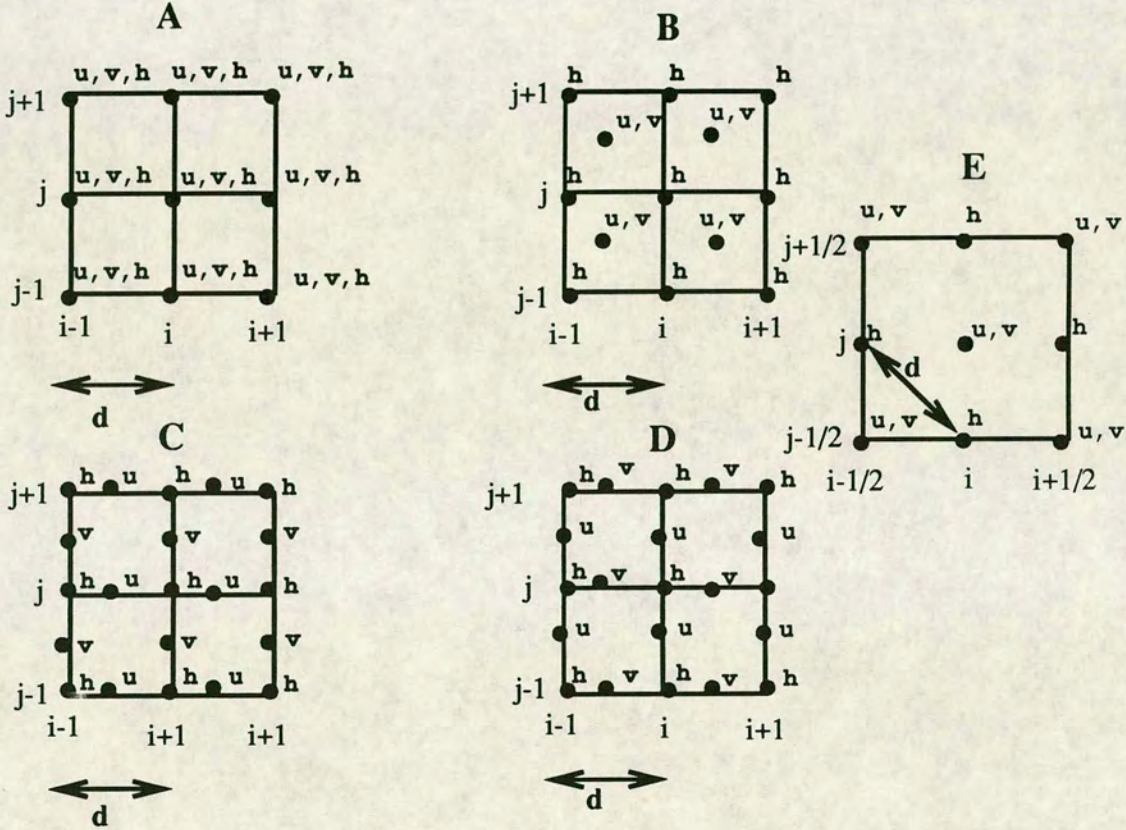


Figure 1-4: Arakawa grids

Five different grids are shown labeled A to E. The arrow shows the grid separation and the horizontal grid point labels are also shown.

is $u_{(i+1/2, j+1/2)} - u_{(i+1/2, j-1/2)}$, it is necessary to compute $U(i, j) - U(i-1, j)$ ⁵. This point will be returned to in Chapter 5.

1.4.3 Fast Fourier Transform

As has already been discussed, in grid-point models the Fourier transform allows larger timesteps and in the spectral method the Fourier transform is used as part of the transform. The derivation that follows is based on that of Hockney and Jesshope (1988).

The aim here is to compute the discrete Fourier transform for 2^q values. For this it is necessary to compute,

$$F(k) = \sum_{j=0}^{2^q-1} \omega_{2^q}^{jk} f(j) \quad j, k \in \{0, 1, \dots, 2^q - 1\} \quad (1.45)$$

with $\omega_{2^q} = \exp(\pm 2\pi i / 2^q)$. The sign depends on the direction of the transform, but is arbitrary as long as the signs are different for the forward and reverse transformation. Note that ω_2 is -1 .

To provide some motivation consider

$$F(k) = \sum_{j=0}^{2^q-1} \omega_{2^q}^{jk} f(j) \quad (1.46)$$

This could be computed by a matrix-vector multiplication with the elements of $\omega_{2^q}^{jk}$ forming a matrix M_{jk} and therefore the time to compute the $F(k)$ will be $O(2^{(2q)})$.

Now separate the sum in Equation 1.46 into odd and even values of j to obtain,

$$F(k) = \sum_{j=0}^{2^q-1-1} \omega_{2^q}^{2jk} f(2j) + \sum_{j=0}^{2^q-1-1} \omega_{2^q}^{(2j+1)k} f(2j+1). \quad (1.47)$$

⁵Using a Fortran notation.



This then implies

$$F(k) = \underbrace{\sum_{j=0}^{2^{q-1}-1} \omega_{2^{q-1}}^{jk} f(2j)}_{\text{Even}} + \omega_{2^q}^k \underbrace{\sum_{j=0}^{2^{q-1}-1} \omega_{2^{q-1}}^{jk} f(2j+1)}_{\text{Odd}}. \quad (1.48)$$

The transform has been separated into two parts called “even” and “odd”. The two parts are the Fourier transform of the even sites and odd sites. It will now be shown that this procedure can be repeated indefinitely and that the time to compute the Fourier transform will be reduced to $O(q2^q)$. Introduce the following definition;

DEFINITION 1.1

$$\overline{F}_i^k{}^{(l)} = \sum_{j=0}^{2^{l-1}-1} \omega_{2^l}^{jk} f(2^{(q-l)}j + i), \quad j, k \in \{0, 1, \dots, 2^l - 1\}$$

$$i \in \{0, 1, \dots, 2^{(q-l)} - 1\}.$$

Each of the \overline{F}_i^k are $2^{(q-l)}$ different Fourier transforms labelled by the index i with the index within the transform being labelled by k . From this definition it can be seen that the following holds,

$$\overline{F}_i^{(0)} = f(i)$$

$$\overline{F}_0^{(q)} = \sum_{j=0}^{(2^q-1)} \omega_{2^l}^{jk} f(j).$$

i.e. the $\overline{F}_i^{(0)}$ are the original values and that $\overline{F}_0^{(q)}$ are the final transformed values. It will now be shown that the following pair of equations are true.

$$\overline{F}_i^{(l+1)} = \overline{F}_i^{(l)} + \omega_{2^{l+1}}^k \overline{F}_{i+\frac{N}{2^{l+1}}}^{(l)} \quad (1.49)$$

$$\overline{F}_{i+2^l}^{(l+1)} = \overline{F}_i^{(l)} - \omega_{2^{l+1}}^k \overline{F}_{i+\frac{N}{2^{l+1}}}^{(l)} \quad (1.50)$$

From Definition 1.1,

$$\overline{F}_i^{k(l+1)} = \sum_{j=0}^{2^{(l+1)}-1} \omega_{2^{(l+1)}}^{jk} f(2^{q-(l+1)}j + i). \quad (1.51)$$

If this is then separated into odd and even parts then the following is obtained,

$$\begin{aligned} \overline{F}_i^{k(l+1)} &= \sum_{j=0}^{2^{l-1}} \omega_{2^l}^{jk} f(2^{q-l}j + i) + \sum_{j=0}^{2^{l-1}} \omega_{2^{(l+1)}}^{(2j+1)k} f(2^{q-l}j + 2^{(q-l)} + i) \\ \Rightarrow \overline{F}_i^{k(l+1)} &= \overline{F}_i^{k(l)} + \omega_{2^{(l+1)}}^k \sum_{j=0}^{2^{l-1}} \omega_{2^l}^{jk} f(2^{q-l}j + i + 2^{(q-l)}) \\ \Rightarrow \overline{F}_i^{k(l+1)} &= \overline{F}_i^{k(l)} + \omega_{2^{(l+1)}}^k \overline{F}_{i+2^{(q-l)}}^{k(l)} \end{aligned}$$

To show Equation 1.50 consider the first term from left hand side of the first line and the second term from the second in the proof above, and substitute $k + 2^l$ for k in that line to obtain

$$\begin{aligned} \overline{F}_i^{k+2^l(l+1)} &= \sum_{j=0}^{2^{l-1}} \omega_{2^l}^{j(k+2^l)} f(2^{q-l}j + i) + \omega_{2^{(l+1)}}^{(k+2^l)} \sum_{j=0}^{2^{l-1}} \omega_{2^l}^{j(k+2^l)} f(2^{q-l}j + i + 2^{(q-l)}) \\ \Rightarrow \overline{F}_i^{k+2^l(l+1)} &= \overline{F}_i^{k(l)} + \omega_2 \omega_{2^{(l+1)}}^k \overline{F}_{i+2^{(q-l)}}^{k(l)} \\ \Rightarrow \overline{F}_i^{k+2^l(l+1)} &= \overline{F}_i^{k(l)} - \omega_{2^{(l+1)}}^k \overline{F}_{i+2^{(q-l)}}^{k(l)} \end{aligned}$$

Having considered the transform of a complex variable, the transform of real numbers is now considered. See Press *et al.* (1986) for a broader discussion of this topic. In many applications of the FFT the input values are real. The transformed complex values, F_m , of the input real values, f_j , then satisfy, if there are M input values,

$$F_{M-m} = (F_M)^* \quad (1.52)$$

and, if the input values, g_j , are purely imaginary then the following is satisfied,

$$-i G_{M-m} = (G_M)^*. \quad (1.53)$$

The Fourier transform is a linear operator and the transform of the sum of two arrays is the sum of the transforms. Using this, two real input arrays, f_j and g_j can be formed into one complex array h_j by $h_j = f_j + i g_j$.

The transform of the complex array, h_j , H_k , will be a linear superposition of the two complex arrays which satisfy the relationships in equations 1.52 and 1.53. Using these relationships it is easy to show that

$$\begin{aligned} F_m &= \frac{1}{2} (H_m + H_{(M-m)}^*) \\ G_m &= \frac{1}{2i} (H_m - H_{(M-m)}^*) \end{aligned} \quad (1.54)$$

Consider now the reverse process where two arrays, F_m , G_m , both satisfy relationship 1.52. The inverse transform can be done by forming $H_m = F_m + iG_m$ and carrying out the inverse Fourier transform. As the transform is a linear operator then f_j and g_j will be the real and imaginary parts of the transform.

1.5 Summary

To complete the chapter a brief summary will now be given. The first unsuccessful Numerical Weather Prediction (NWP) was made by Richardson. He imagined that NWP would be carried out, on an operational basis using several thousand human computers. The ever increasing demands by atmospheric modellers for computer power were next examined and it was concluded that the present technology of shared memory vector computers was inadequate for modelers' needs. The only likely alternative is parallel computers with distributed memories. The theme of this thesis is what algorithms or modifications or both to existing methods will be required to utilize MIMD computers effectively. The next chapter will introduce some ideas used in designing algorithms for those computers.

Chapter 2

Parallel Algorithm Design

2.1 Overview

The following three chapters will describe how different meteorological models are implemented on a parallel computer. This chapter will explain the principles that lie behind the design of the algorithms.

Part of the “folklore” of parallel computing is Amdahl’s law which, as restated by Lazou (1988) is:

When a computer has two distinct modes of operation, a high speed and a low speed, the overall operation is dominated by the low speed mode unless the fraction of work processes in the low speed mode can be virtually eliminated.

In particular, if an algorithm has a serial part, (which cannot be speeded up) and a parallel part (which can) then, no matter how many processors are used, it will not be possible to obtain a speed greater than that of the serial part.

The model of parallel programming and design that was used here was that of Communicating Sequential Processes (CSP) described by Hoare (1985). For the purposes of this thesis, this can be considered as a set of sequential programs which communicate with one other. Communications causes synchronisation, in that if a process desires to receive a message from or send a message to another process it must wait until that process is ready to send or to receive that message. When both processes are ready then the exchange takes place and only then can both processes proceed. A similar model is the calculus of communicating systems (CCS) (Milner, 1989).

When using MIMD computers like the Edinburgh Concurrent Super-computer (ECS) machine, thought must be given to the decomposition of the problem over many processors. Once the problem has been decomposed, the communications between the processors needs to be considered. The first section in this chapter will introduce some decomposition strategies. Following on from that is a discussion of how the components (or *processes*) of a parallel program interact through communications and the importance of the mapping of these processes to the machine hardware.

Having introduced the idea of process topology, a discussion of the conditions which may cause a program to stop, namely deadlock and livelock, follows. When the program stops in these circumstances it is difficult to detect that it has stopped. Some general ideas on associative operators are presented in Section 2.5. These are used in Chapters 5 and 4. The question of how the efficiency of a parallel program is measured and a case for looking at scaled speedup is made. To complete the chapter a very brief discussion of the input/output problem is given.

2.2 Decomposition Strategies

Classically there are three possible decomposition strategies for using parallel computers. They are,

Geometrical The computational domain is divided into contiguous pieces and each piece is given to a processor. The mapping of the pieces to the processors should preserve the topology of the entire domain, with adjacent pieces being mapped to adjacent processors. This paradigm is widely used in physical applications. Chapter 5 shows how this can be used for grid point models.

Algebraic If a complex operation requires computation, then it may be possible to break it down into various sub-components with each component executed in parallel by a different processor. Sometimes the operation will have no sub-components which can be computed in parallel. Identifying the sub-components may be difficult and the computations on them are likely to take different amounts of time. The speed at which the parallel computer will work at is determined by the slowest part and if this unbalanced condition does occur then the efficiency of the implementation will be reduced. Another point that should be raised is that for many meteorological applications there will only be a limited amount of parallelism available from this method. It is therefore unlikely that many meteorological applications will find this a particularly useful strategy.

Task If many *independent* tasks need to be done, and the number of tasks is greater than the number of processors, then each processor will compute at least one task. When a processor has finished with a task it will

process the next one. In the meteorological context, examples where this approach would be helpful include ensemble forecasting, where an ensemble of forecasts is integrated forwards in time (Murphy, 1990; Branković *et al.*, 1990), and climate experiments where several experiments could be run in parallel (Mitchel *et al.*, 1989). Chapter 3 shows how this approach is used to build a load-balancer for the parameterization schemes.

Algebraic decomposition can be done by building a pipeline, in which many tasks move through the pipeline, being processed sequentially by the sub-components. Normally there will only be a limited number of sub-components that can be profitably identified and used. In order to use the parallelism in the problem a geometric decomposition has also to be used. In Chapter 4 this combined approach is used to implement a spectral model.

2.3 Topology

A program is considered as a collection of processes. A process, at least for the purposes of this thesis, can be considered as a sequential program with the addition of communications. See Hoare (1985) or Milner (1989) for a more formal definition of a process. The communications between them form a pattern. This pattern is termed the *process topology*. Analogous to this are the connections between the processors of a parallel computer, the *computer topology*. In this section, the topology of the process network and its relationship to the topology of the parallel architecture will be discussed. The communications pattern between the processes of a parallel program can be expressed as a directed graph (Gelernter, 1981). Processes form the nodes of this graph while the communications between the processes are represented as directed edges. The neighbours of a process are those

processes connected to it by only one edge. This directed graph will form the topology of the program.

The topology of the hardware is also important. Many existing computers use a hypercube topology, where the 2^n processors form a n -dimensional hypercube. Each processor has n neighbours and is separated from any other processor by at most n edges. Machines using such topologies include the Connection Machine (Hillis, 1984) and the Intel iPSC/2. A description of a fluid dynamics application on the latter can be found in Chamberlain and Chesshire (1990).

Other computers use a mesh topology, where the processors are connected in a two dimensional grid. An example of such a computer is the AMT DAP (Associative Memory Technology, 1990).

Many Transputer based machines are reconfigurable and can be connected into many different topologies, although, since the processors only have a fixed valency, they cannot in general form a hypercube. Valency means the number of input/output links the processor has, with the T800 having a valency of 4. The symbol v will frequently be used to mean the valency of the processor.

One problem with fixed valency computers is the mapping of the process topology to the machine topology. Ideally neighbours in the process topology should be mapped to neighbouring processors in the machine topology, although this is not possible if the machine valency is too low. This could occur if, for example, a process needs to communicate with 5 other processes while the machine allows a maximum of 4 processors to be connected to any one processor. However, Prior *et al.* (1990) shows that random graphs of fixed valency nodes can have good properties and in particular for a random network of N processors it is possible to send a message between any two processors in an average time of $O(\ln N)$.

On many existing machines, communication between processors is significantly slower than the computation speed and through-routing through processors is expensive. Many existing architectures pass messages by “store and forward” in which, for a message to be through-routed by a processor on its way to another processor, the intermediate processor receives the entire message before sending it towards its destination. In this case the time for a message to pass along N links, through $N - 1$ processors, is proportional to N . The T800 works by this mechanism, with the through-routing being implemented in software. The efficiency of the computer will be increased if processes that communicate with one another are mapped to neighbouring processors.

An alternative to store and forward routing is wormhole routing; once an intermediate processor receives the first part of a message it starts routing that message through the correct links. If the message is large enough then the first part of it could arrive at the destination processor before the last part leaves the originating processor. Contrast this with store and forward routing where a processor must receive the entire message before sending this message onto the next processor. If wormhole routing exists on an architecture, then through-routing a message will not significantly increase the time taken by the message to pass between the two processors. However the N links being used by the message cannot be used by any other processor until the message has arrived. In effect the total bandwidth of the machine has been repartitioned with a larger fraction being given to the single message (Dally, 1990). If many processors wish to communicate across several links then there may not be sufficient bandwidth to enable them all to complete rapidly. Again, machine efficiency will be improved by mapping neighbouring processes as close to each other as possible.

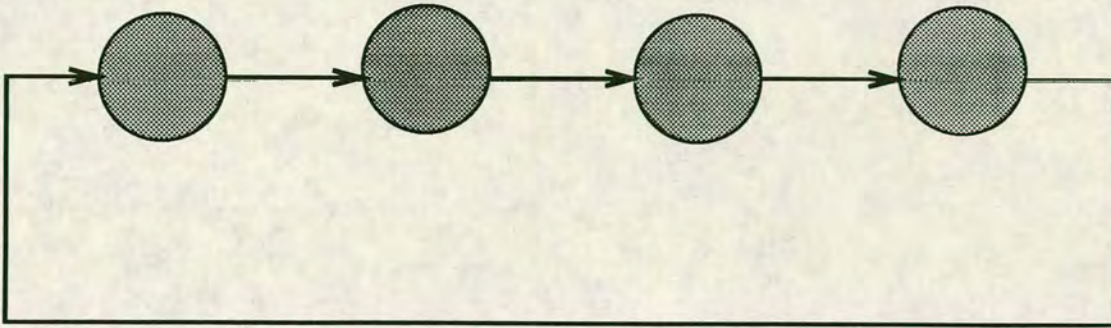


Figure 2-1: Deadlock in a ring

Arrows show how the processes (represented by the shaded circles) wish to send a message.

2.4 Deadlock and Livelock

When designing and writing parallel programs it is imperative that the designer ensures that the program never enters into either of two states called “deadlock” and “livelock”.

As soon as two parallel processes can interact, there exists the possibility of deadlock. In a deadlocked set of processes, each process in the set is waiting for another process, also in the set, to interact with it. In the CSP model of parallel computing, the interaction is a communication and the wait is caused by one process starting a send or receive and waiting for the other process to receive or send data as appropriate. Deadlock may occur when a cycle exists in the directed graph of the processes. Figure 2-1 shows an example of deadlock where all the processors in a ring wish to send data to their right hand neighbour. Deadlock is a property of the topology of the communications pending at any one time.

Livelock is a less common problem and can occur if a process will stay active and use processor resources (i.e. CPU time) until it receives a signal

from another process. If the other process is on the same processor and the first process is consuming all the CPU's time, then the second process will never become active, and so the first process will never receive its signal to stop it being active. In practice this situation may not arise, but one process could waste an arbitrarily large fraction of the CPU's time.

An example of this occurs with a simple task farmer, where the processes are configured as a ring. A task will continue moving around the ring until it finds a processor which has no work to do, whereupon that processor will do some work on the task. There are two processes on each processor as shown in Figure 2-2, with the communications design being such as to avoid deadlock.

1. A compute process which does the tasks.
2. A communications process which moves tasks to the next processor in the ring or sends them to the compute process on the same processor.

If each processor is processing a task and all of the processors have another task, then each processor will send this second task to its neighbour. This passing of tasks around the ring will continue until there is no processor with more than one task¹. At this point there will be no tasks to transfer to neighbouring processes. However the system may never get into this state; the CPU time taken to move tasks from processor to processor may not leave sufficient time for the tasks to be done and so the system will continuously keep moving tasks around the ring. For this to happen the communications process would have to take all the CPU time, it is more likely that it would take a large fraction of the time and in that case the computations process

¹That task will be being processed.

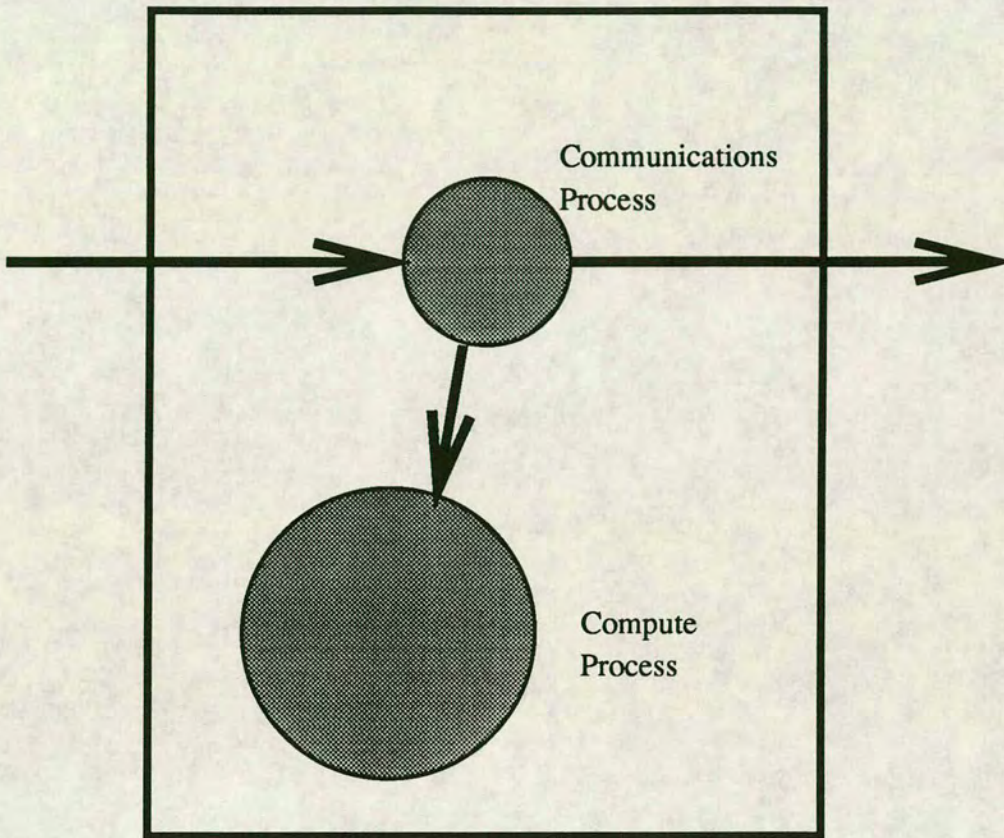


Figure 2-2: Process design that leads to livelock

The box represents one processor with two processes, arrows show communications between processes. The arrows entering and leaving the processor come from and go to the communications process on the next processor. The design of the communications process is such as to avoid deadlock.

would eventually finish but it would take a much longer time than would be expected.

2.5 Associative Operators

The aim of this section is to present some useful results about associative operators. A large class of problems, known as reduction problems, consist of reducing many values to a single value. A typical example is adding together many data values to produce their sum. Related problems involve the computation of iterative forms, for example $F_{n+1} = \max(F_n, x_n)$, to which the techniques described here may be applied. This section discusses algorithms in terms of a general associative operator, which might be $+$ or \max . It is a generalisation of Section 5.2.2 of Hockney and Jesshope (1988) on cascade sums. Also see Brent (1974) for similar ideas for arithmetic operators. Figure 2-3 shows a routing diagram for a cascade sum, or for that matter any associative operator. Reference to this figure should be made during the following discussion.

Consider a binary operation \circ , not necessarily associative, on elements $a_1, a_2, \dots \in A$ of type $A \times A \rightarrow A$. Examples of such operators are \max , \times , matrix multiplication and vector cross product. The operator \circ can be extended to the $N + 1$ -ary operator $\Omega_{i=0}^N a_i$ (the generalised product) which is defined by the following recursive equations.

DEFINITION 2.1

$$\begin{aligned}\Omega_{i=0}^0 a_i &\equiv a_0 \\ \Omega_{i=0}^N a_i &\equiv (\Omega_{i=0}^{N-1} a_i) \circ a_N.\end{aligned}$$

Now assuming that \circ is associative i.e. $a_1 \circ (a_2 \circ a_3) = (a_1 \circ a_2) \circ a_3$ then the following is true,

$$\Omega_{i=P}^Q a_i = \Omega_{i=P}^R a_i \circ \Omega_{i=R+1}^Q a_i \quad \forall P + 1 \leq R \leq Q - 1. \quad (2.1)$$

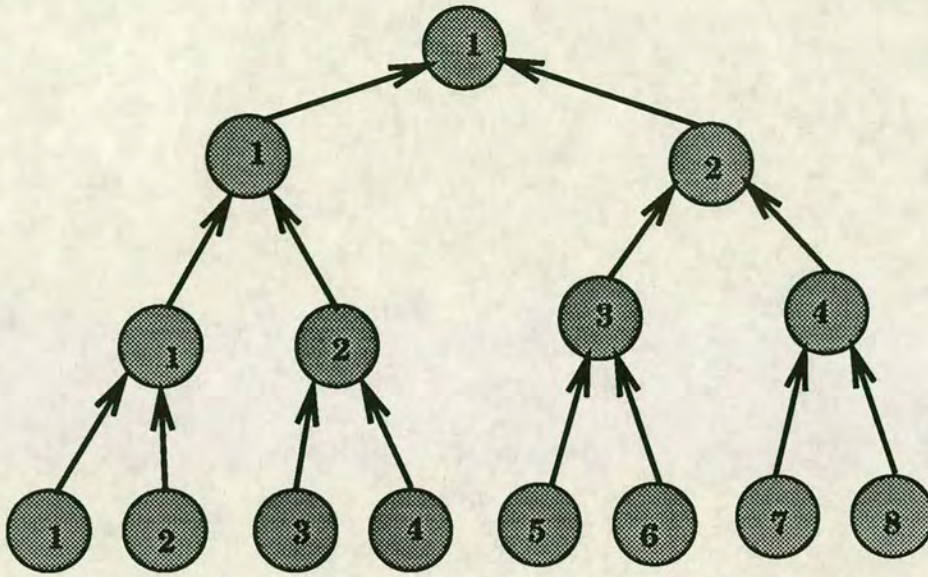


Figure 2-3: Topology for cascade sum

In this diagram the shaded circles represent processes and the communications between them are represented by arrows. The numbers label the processes at each level of the tree.

In several meteorological applications, it is necessary to compute $A = \Omega_{i=0}^k a_i$, where the values of i will range across some large computational domain. Examples of this are the computations of total energy, mass and the zonal averaging that occur as part of the diagnostics of many models. The values of a_i will be held on many different processors. The problem is how to compute A efficiently. To begin with, a tree topology, as in Figure 2-3, is built with the a_i being mapped to leaf processors of this tree such that a_i and a_{i+1} , i being the site labels, are on the same processor or on processors whose processor labels are P and $P + 1$.

For a processor P , the minimum value of the site label is denoted by \mathcal{P}_{\min} , while the maximum is \mathcal{P}_{\max} . Each leaf processor's sub-domain will consist of the contiguous site labels from \mathcal{P}_{\min} to \mathcal{P}_{\max} . Note that if both processor labels P and $P + 1$ exist, then $\mathcal{P}_{\max} + 1 = (\mathcal{P} + 1)_{\min}$.

To compute A , each leaf processor will compute $A(P) = \Omega_{i=\mathcal{P}_{\min}}^{\mathcal{P}_{\max}} a_i$. All leaf processors can do this in parallel. These values of $A(P)$ are passed on to the nodes in the next level of the tree, connected to the leaf processors. These node processors compute $A(P)$ from the input values of $A(P)$, in the order of lowest input processor label to highest input processor label. Again these labels form a contiguous set. These nodes pass their computed values up to the next level of nodes, which will again compute a value of $A(P)$ from their input values and so on. Finally the desired value of A appears in the root processor.

Having shown how to compute the reduction operation, the question of the iterative form is examined. In this case it is necessary to compute something of the following form,

$$A_i = a_i \circ A_{i-1} \quad (2.2)$$

with $A_0 = a_0$. It is easy to see that A_k is equal to $\Omega_{i=0}^k a_i$. Again this can be done in parallel, with a similar algorithm to that described earlier in the section.

The algorithm begins by all leaf processors computing values of $\Omega_{\mathcal{P}_{\min}}^k$ for all values of k that are on that leaf processor i.e. $k \in \{\mathcal{P}_{\min}, \dots, \mathcal{P}_{\max}\}$. In order to compute Ω_0^k the value of $\Omega_0^{(\mathcal{P}-1)_{\max}}$ will need to be input to the leaf processor. Once this is done then the leaf processor can compute Ω_0^k using $\Omega_0^{(\mathcal{P}-1)_{\max}} \circ \Omega_{\mathcal{P}_{\min}}^k$.

In order to arrange that $\Omega_0^{(\mathcal{P}-1)_{\max}}$ is input to the leaf processor the following needs to be done.

Each leaf processor should send the value of $\Omega_{\mathcal{P}_{\min}}^{\mathcal{P}_{\max}}$ to the node processor above it in the tree. The behaviour of the node processors is different in this case from the reduction operation considered previously. Rather than just passing values to the nodes above them, each node will compute generalised

products from their input values. A node, \mathcal{P} , receives inputs from processors labeled $\mathcal{P}(1), \mathcal{P}(2), \dots, \mathcal{P}(v-1)$ with values, $\Omega_{\mathcal{P}(1)_{\min}}^{\mathcal{P}(1)_{\max}}, \dots, \Omega_{\mathcal{P}(v-1)_{\min}}^{\mathcal{P}(v-1)_{\max}}$. From these the node processor computes $\Omega_{\mathcal{P}(1)_{\min}}^{\mathcal{P}(1)_{\max}}$, with I ranging from 1 to $v-1$. All node processors can then send $\Omega_{\mathcal{P}(1)_{\min}}^{\mathcal{P}(v-1)_{\max}}$ to the node processor above them in the tree². If a node processor has the smallest label (it is on the left hand edge in Figure 2-3) then $\Omega_{\mathcal{P}(1)_{\min}}^{\mathcal{P}(1)_{\max}}$ is $\Omega_1^{\mathcal{P}(1)_{\max}}$ for all values of I . To the processors, below it in the tree, labeled $\mathcal{P}(I)$ with I ranging from 2 to $v-1$ it sends $\Omega_1^{\mathcal{P}(I-1)_{\max}}$. All other processors, labelled I , in a row wait until they receive values of $\Omega_1^{\mathcal{P}(I-1)_{\max}}$ from the node processor above them. They can use this to compute $\Omega_1^{\{\mathcal{P}'(I)\}_{\max}}$, with the notation $\{\mathcal{P}'(I)\}$ meaning the set of processor labels that are below $\mathcal{P}(I)$ in the tree. Having computed these values the node processors can send them to the processors below them in the tree. Figure 2-4 provides a schematic illustration of this.

Similar ideas to the ones presented above have been extensively developed in the computer science community, mainly in the field of systolic algorithms (Kung, 1979).

2.6 Parallel Efficiency

This section will define parallel efficiency, then discuss how throughput and efficiency are linked and how to measure them. The time taken by N processors will be denoted by $\tau(N)$ throughout. There are two contexts in which efficiency is typically considered and they require slightly different ways of measuring it. These contexts are discussed below.

²For the node processor with the largest label this value will not be used by the node processor above it

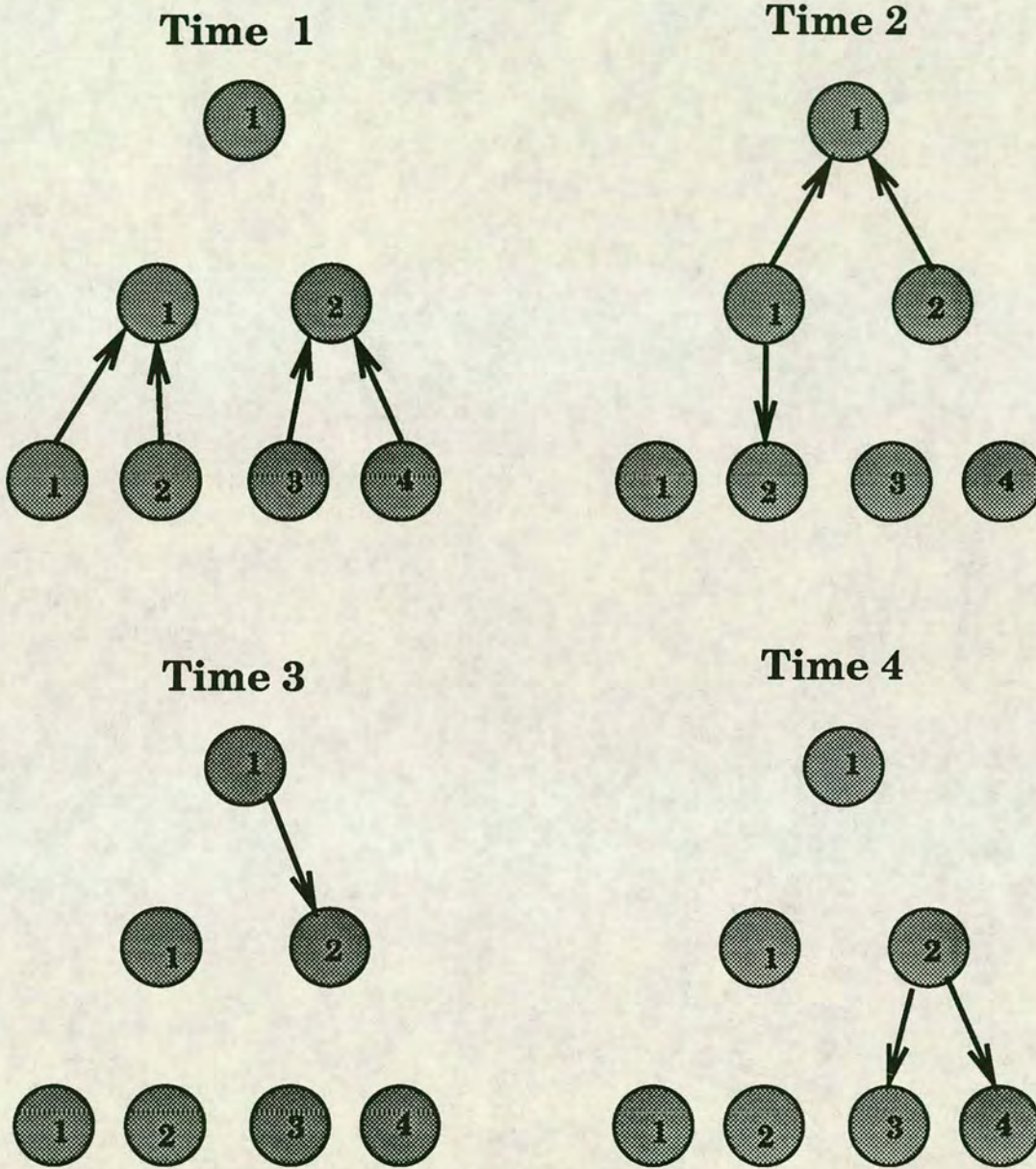


Figure 2-4: Schematic illustration of parallel iterative operations

At time 0 all leaf processors (the bottom row in the tree) compute $\Omega_{P(\min)}^{P(\max)}$. At time 1 all the leaf processors send this computed value to the node processors above them in the tree. At time 2 node processor 1 sends $\Omega_1^{1(\max)}$ to leaf processor 2 and also sends $\Omega_1^{2(\max)}$ to the root processor (the processor at the top of the tree). Also, at time 2, node processor 2 sends $\Omega_{3(\min)}^{4(\max)}$ to the root processor. At time 3 the root processor sends $\Omega_1^{2(\max)}$ to node processor 2 which uses it to compute $\Omega_1^{3(\max)}$. At time 4 node processor 2 sends $\Omega_1^{2(\max)}$ and $\Omega_1^{3(\max)}$ to leaf processors 3 and 4 respectively

Fixed Problem Size Here the efficiency is examined as the number of processors solving a fixed size problem increases. Amdahl's law implies that the efficiency of the algorithm problem will decrease with increasing number of processors. For example, the relative cost of the loop overheads will increase as the loop size on an individual processor decreases. This will be termed the "fixed problem" in the following discussion. A perfectly efficient parallel algorithm running on 1 processor would take N times as long as it would take using N processors. This leads to the following definition for efficiency, denoted by e .

DEFINITION 2.2

$$e = \frac{\tau(1)}{N\tau(N)}.$$

Scaled Problem Size In many meteorological problems scientists would like to run larger models. As more powerful computers become available they will use these computers to explore larger problems. In this context the problem size grows linearly with the number of processors available. Here the relative loop overhead will stay constant but other effects may cause an efficiency decrease. From here on this will be termed the "scaled problem". For the "scaled problem", a perfectly parallel algorithm should require the *same* time on N processors as on 1 processor (since the problem is N times as big on N processors). Here the efficiency is defined by the following.

DEFINITION 2.3

$$e = \frac{\tau(1)}{\tau(N)}.$$

Having defined the efficiency for the two cases, consider the problem of computing the total efficiency of a program given the efficiency of each of its sub-components.

Consider a program with k components; each part labelled by i ranging over $\{1, \dots, k\}$, and having efficiency $e_i(N)$ and taking a time $\tau_i(N)$ on N processors. For both definitions of efficiency, the total efficiency of the whole program is given by the following expression,

$$e(N) = \frac{\sum_{i=1}^k e_i(N) \tau_i(N)}{\sum_{i=1}^k \tau_i(N)}. \quad (2.3)$$

This can be seen as follows. For the scaled case, from Definition 2.3 $e_i(N) = \tau_i(1)/\tau_i(N)$ therefore the right hand side of Equation 2.3 can be rewritten to give,

$$e(N) = \frac{\sum_{i=1}^k (\tau_i(1)/\tau_i(N)) \tau_i(N)}{\sum_{i=1}^k \tau_i(N)} = \frac{\sum_{i=1}^k \tau_i(1)}{\sum_{i=1}^k \tau_i(N)}. \quad (2.4)$$

But $\sum_{i=1}^k \tau_i(1)$ is equal to $\tau(1)$ for the whole program and similarly $\sum_{i=1}^k \tau_i(N) = \tau(N)$ so this is exactly the scaled efficiency given in Definition 2.3.

For the “fixed problem”, a similar argument applies. Use Definition 2.2 and substitute it in Equation 2.3 to obtain,

$$e(N) = \frac{N \sum_{i=1}^k \tau_i(i)}{\sum_{i=1}^k \tau_i(N)} \quad (2.5)$$

which is the required expression.

2.6.1 Throughput versus Efficiency

In this subsection the tradeoffs required between throughput and efficiency will be examined. All else being equal it will be more efficient to run several programs independently on different processors than to run them one after the other with each one being decomposed over the entire machine. This will now be shown.

Consider the case of a “fixed problem” which has a parallel part, taking time P , on one processor and P/N on N processors. It has in addition a serial part which takes a time S on any number of processors. On one processor the time taken will be $P + S$ while on N processors the time taken will be $P/N + S$. By Definition 2.2 the efficiency is

$$e(N) = \frac{(P + S)/N}{P/N + S}. \quad (2.6)$$

Furthermore let S' and P' be the fractions of the total problem taken by the serial and parallel parts respectively, i.e. $S' = S/(S + P)$ and $P' = P/(S + P)$. The efficiency for one task running on a N processor machine is

$$e_1(N) = \frac{1}{P' + NS'}. \quad (2.7)$$

Consider now the case of two such tasks running on the machine with each task using half the total processors. The efficiency is the same as the efficiency of one task using a $N/2$ processor machine.

$$e_2(N) = e_1(N/2) = \frac{1}{P' + N/2S'}. \quad (2.8)$$

As $e_2 > e_1$ it is therefore true that the efficiency of M independent tasks running on N processors³ is greater than running the M tasks sequentially.

This analysis has ignored two things. First, it assumes that there is unlimited memory available on the machine. There may not be sufficient memory on all processors to enable some tasks to be run. The second point that has been ignored is that for many meteorological applications there is a maximum time in which the model must run. The best example of this is a numerical weather prediction (NWP) model at a forecast centre. If the model does not run within this time then its results will be of no interest to the intended users.

³As long as all the processors are used.

2.6.2 Measuring Parallel Efficiency

To complete this section, methods of measuring parallel efficiency will be discussed. Some justification for the idea of scaled speedup will first be given. In the introduction it was shown that model complexity and size has increased along with increasing computing power. If this trend continues, then the concept of scaled speedup is useful. Here the problem size is allowed to grow in proportion to the number of processors. The time taken by N processors is directly proportional to the efficiency.

For some problems scaled speedup is not an appropriate measure. In this case the problem size remains fixed and the efficiency of the program will be measured as the number of processors increase. What is actually measured is the time taken by the problem on N processors. Defining the scaled time as the time taken by N processors multiplied by N , it is straightforward to show that the scaled time is ^{inversely} proportional to the efficiency.

2.7 Input/Output

Parallel computers will not be processing in isolation. The data they produce will need to be transferred to other computers for further processing, to storage for later analysis and archive, and to be disseminated to users who require the information contained in the forecast (whether a climate or a NWP model).

Models also require data. They require initial data and, depending on the type of model, they may need further data as they integrate forwards in time. A forecast model while assimilating data observations, see for example Lorenc *et al.* (1991), will need these observations at regular model time intervals. Climate models may need some ancillary fields. For example,

an atmosphere-only model will need values of the sea-surface temperature fields at regular model time intervals.

The scaling behaviour of I/O is examined to see if this may be a bottleneck and cause the parallel computer to spend as much time transferring data out of the computer as it does doing the computations to produce this data. The computational complexity of a three-dimensional model is $O(N^4)$, with $O(N^3)$ coming from the number of grid-cells in the computational domain⁴ and $O(N)$ from the time-step required for numerical stability or computational accuracy.

For a serial model, of data set size $O(N^3)$ the wall-clock time⁵ to integrate the model forwards through one model day will be $O(N^4)$. In this time the amount of data to be transferred out of the model is $O(N^3)$. Therefore the ratio of I/O time to compute time for a serial model will be $1 : O(N)$. This assumes that the model users would not wish to examine data every model timestep.

Now consider the behaviour of a parallel computer, where the number of processors is assumed to increase in proportion to the horizontal problem size. As will be shown in later chapters, the amount of vertical decomposition is limited. Thus the time to compute a model day is $O(N^2)$, assuming perfect scaleup. As in the serial case the amount of data that needs to be transferred out from the computer is $O(N^3)$. Therefore the ratio of I/O time to compute time is $O(N) : 1$. If this situation were allowed to remain then the time taken by I/O would dominate the computing time. In order to make the I/O balance, the parallel-computer would require $O(N)$ channels out of the machine. Note that this argument is a scaling argument and that the

⁴Assuming that vertical resolution increases as horizontal resolution does.

⁵That time measured on a clock in the computer room.

constant which determines the number of I/O channels required may be quite small.

Having presented some general ideas on the design of parallel algorithms, the next three chapters will show how these are used in practice.

Chapter 3

Subgrid Processes

3.1 Introduction

In the previous chapter, parallel algorithms were considered in a general way. This chapter will consider the parameterization schemes required by many atmospheric models. The approach taken in this chapter, in contrast to the next two chapters where dynamics schemes are examined, is mainly empirical. Approximately 50% of a Numerical Weather Prediction or Climate model's time is spent computing these parameterization schemes. In these, physical processes whose length or time scales cannot be explicitly resolved by the model are parameterized in terms of the model variables. Examples of these are gravity wave drag, radiation, convection and large-scale rainfall.

In this chapter, two particular schemes are investigated by implementing them on the Edinburgh Concurrent Supercomputer (ECS). The particular schemes investigated are: (1) a large-scale rainfall scheme, used by the U.K. Met. Office (UKMO) as a benchmark code; (2) a convective rainfall scheme described by Gregory and Rowntree (1990). The second scheme

Table 3-1: Minimum and maximum times for two schemes

Scheme	Min	Max
Rainfall	0.27	0.41
Convection	0.20	0.93

Times are in seconds for 100 trials of the convection scheme and 500 trials for the rainfall scheme. These maximum and minimum times were taken from an experiment described later in the chapter.

is used by the UKMO in its present combined forecast and climate model. Neither of these schemes require any communication between columns of the atmosphere although considerable communication is required within these columns. This is a consequence of the vertical resolution in atmospheric models being much greater than the horizontal resolution.

Parameterization schemes typically have many conditional statements and different pathways through the code. This causes the computational time for the schemes to vary in different columns of the atmosphere, depending on the values of the model variables. Table 3-1 shows the maximum and minimum calculation times taken for the large-scale rainfall scheme and the convection scheme from the experiment described later in this chapter.

This chapter will examine parameterization schemes in the context of a grid-point model. In the grid-point model, for reasons discussed in Chapter 5, it is necessary to map the atmospheric columns to the processors such that columns that are geographically neighbouring are mapped to the same or adjacent processors. Furthermore the columns should be shared out as equally as possible between all the processors. Throughout this chapter the atmospheric columns are assumed to be mapped such that one or more entire columns are on a single processor.

In order to put the work of this chapter into a more general context a

brief discussion on how these schemes are implemented on a vector machine is given. In addition the problems that require solving, before an implementation on a data-parallel machine could be successfully carried out are examined.

The normal technique used on vector machines is “gathering and scattering”. In each branch of a conditional statement a vector will be constructed by gathering all the points where the branch condition is true. This vector will then be acted on by the vector units of the computer. After the computation is completed the points are scattered back to their original locations. The model code may need some re-ordering from a purely serial/non-vectorised state. Consider a simple large-scale rainfall model where, if the humidity is greater than the saturated humidity, water vapour will precipitate out. If there is a falling rain at a single point in the column and the humidity there is less than the saturated humidity then some of the falling rain will evaporate.

```
DO over all points
  DO over all levels
    IF (relative humidity > sat. humidity)
      add contribution to rainfall from condensing vapour
    IF (relative humidity < sat. humidity AND falling rain)
      evaporate falling rain
    ENDIF
  ENDDO
ENDDO
```

On a vector machine this will become,

```
DO over all levels
  GATHER all points WHERE (humidity > sat. humidity)
```

```
DO over all gathered points
  compute rainfall increments
ENDDO
SCATTER rainfall increments back
DO over all points
  add rainfall increment to falling rain
ENDDO
GATHER all points WHERE (humidity > sat. humidity
                        AND falling rain)
DO over all gathered points
  evaporate rainfall
ENDDO
SCATTER rainfall back
ENDDO
```

On data-parallel machines this technique of “gathering and scattering” has two problems:

1. The movement of points across the processor array so that all processors have the same thing to do at the same time involves many communications between processors. If, as is true for many parameterization schemes, the computational work in a single branch of the model compared to the communications time is small then this approach may be extremely inefficient. Communications between processors is, of course, more expensive than memory access on the same processor.
2. Data-parallel computers tend to have many processors and so consequently the number of points per processor is small. See for example the Connection Machine (Hillis, 1984). It is possible that the number of points where one branch is taken is significantly less than the number of processors in the computer. As a result of this if there are

many conditional branches, then the efficiency of this approach may be small.

These points were also raised in the paper by Swarztrauber and Sato (1990) who also pointed out that the coding of the physical parameterizations on such computers could require creative reformulation of the computations. Such a reformulation (S. Reddaway, private communication) is to identify from all branches of the conditional expression the common time-ordered set of operations that are required. It is imperative that the time ordering is preserved. A simple, contrived, example of this is the following;

```
IF (a < 0) THEN
  b=2.0 * SQRT(-a)
ELSE
  b=SQRT(a) / 2.0
ENDIF
```

is transformed to

```
IF (a < 0)
  c=2.0
  a=-a
ELSE
  c=0.5
ENDIF
b=c*SQRT(a)
```

Existing parameterization schemes are written in the language Fortran 77 using the technique of “gathering and scattering” for vector supercomputer. Existing data-parallel computers, such as the DAP and the CM-2 require the use of a specialised language to manipulate elements of arrays

in parallel (Trew and Wilson, 1991). The array manipulation features of these languages are similar to the proposed Fortran 90 standard. Existing codes of parameterization schemes would need extensive modifications so as to use these array features. In addition to the cost of recoding the efficiency of existing schemes could be quite low for the reasons outlined above. It is possible that the efficiency could be improved by a major restructuring of the codes that implement the parameterization schemes.

3.2 Properties of Parameterization Schemes

This section will describe some of the properties of the two schemes investigated. It will mainly concentrate on the work distribution of the schemes. Both schemes were written in standard Fortran and both were designed to run on a vector machine by using the "gathering and scattering" approach described previously. It is straightforward to make both of them process a single column by making the vector length one.

A brief description of both schemes, operating in a single column mode, will now be given, starting with the large-scale rainfall scheme. In this scheme computing starts in the highest grid-cell and proceeds down through the atmospheric column until it reaches the lowest levels. For each grid-cell, if the relative humidity is greater than the saturated humidity then the excess humidity is precipitated out and added to any falling precipitation. If the relative humidity is less than the saturated humidity then any falling precipitation partially evaporates. In both cases temperatures in the grid-cell are adjusted using the Clausius-Clapeyron equation. See any basic textbook on thermodynamics (e.g. Zemansky and Dittman (1981)) for details. Any remaining precipitation then falls through to the next grid-cell down in the column. As precipitation falls it can change phase from "snow"

to "rain" or *vice versa*. The phase changes of the precipitation have an effect on the temperature of the grid-cell in which they occur.

The convection scheme will now be described. In this scheme computing commences in the lowest grid-cell and works up. The scheme finds the lowest layer where there is slight positive buoyancy and initiates convection of a parcel. This rising parcel represents an ensemble of convective plumes. In each layer the parcel detrains parcel air and entrains environmental air. When the parcel is no longer buoyant a fraction of the parcel air is detrained so as to allow the remainder to have slight positive buoyancy. The process continues until the entire parcel has completely detrained.

On a serial machine the time taken to compute the effect of a parameterization scheme will be dependent only on the mean time and the size of the data-set. On a parallel computer, the performance is determined by the slowest processor, therefore in addition to the the mean time, the distribution of times taken must also be considered. Consider first the work required to compute the effects of a scheme on a single atmospheric column. It is expected that the state of the column would affect the time taken. In order to measure this, data-sets for both schemes were obtained. The data-sets and codes for the schemes were all supplied by the U.K. Meteorological Office. For the rainfall scheme, this data-set was for a limited area of the globe; while for the convection scheme, a global data-set was supplied. The surface values of these data-sets are shown in Figures 3-1 and 3-2 respectively. There were approximately 3000 atmospheric columns in the rainfall data-set and approximately 6000 columns in the convection data-set.

For each atmospheric column in the data-set the scheme was run several times and the total time taken measured. After each run of the scheme on a single column, the values of that column were reset to their original values. For the convection scheme each column was processed 100 times, while for the rainfall scheme each column was processed 500 times. These

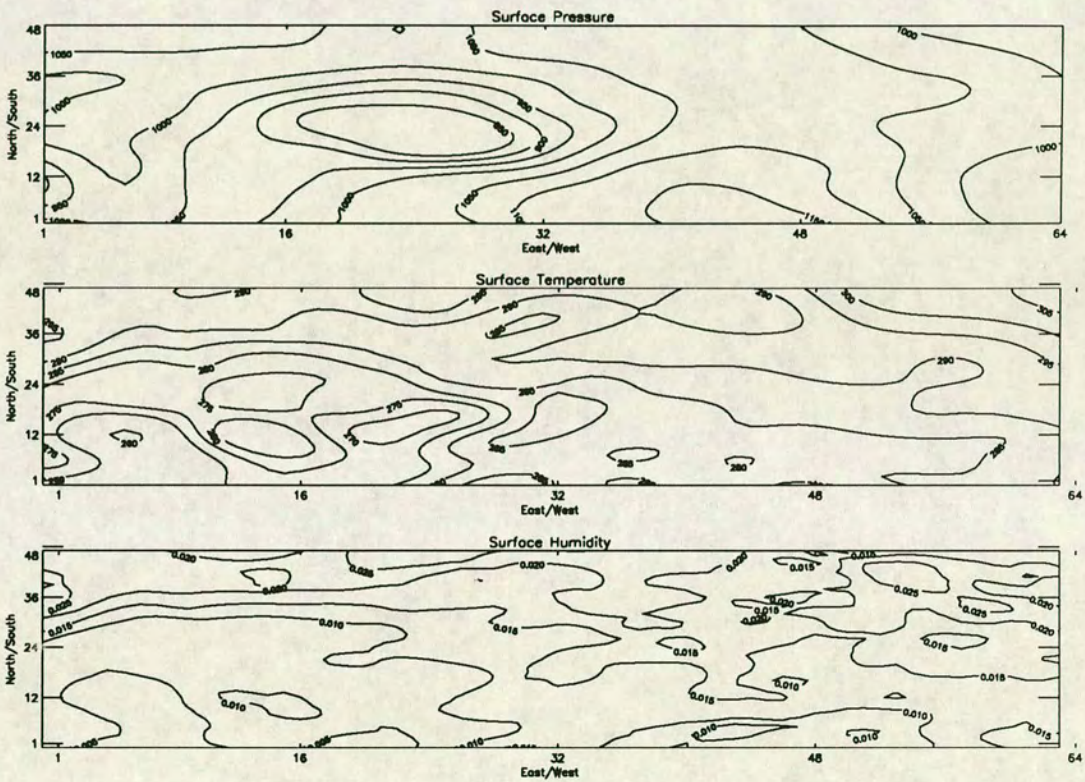


Figure 3-1: Surface data used for the large-scale rainfall parameterization scheme

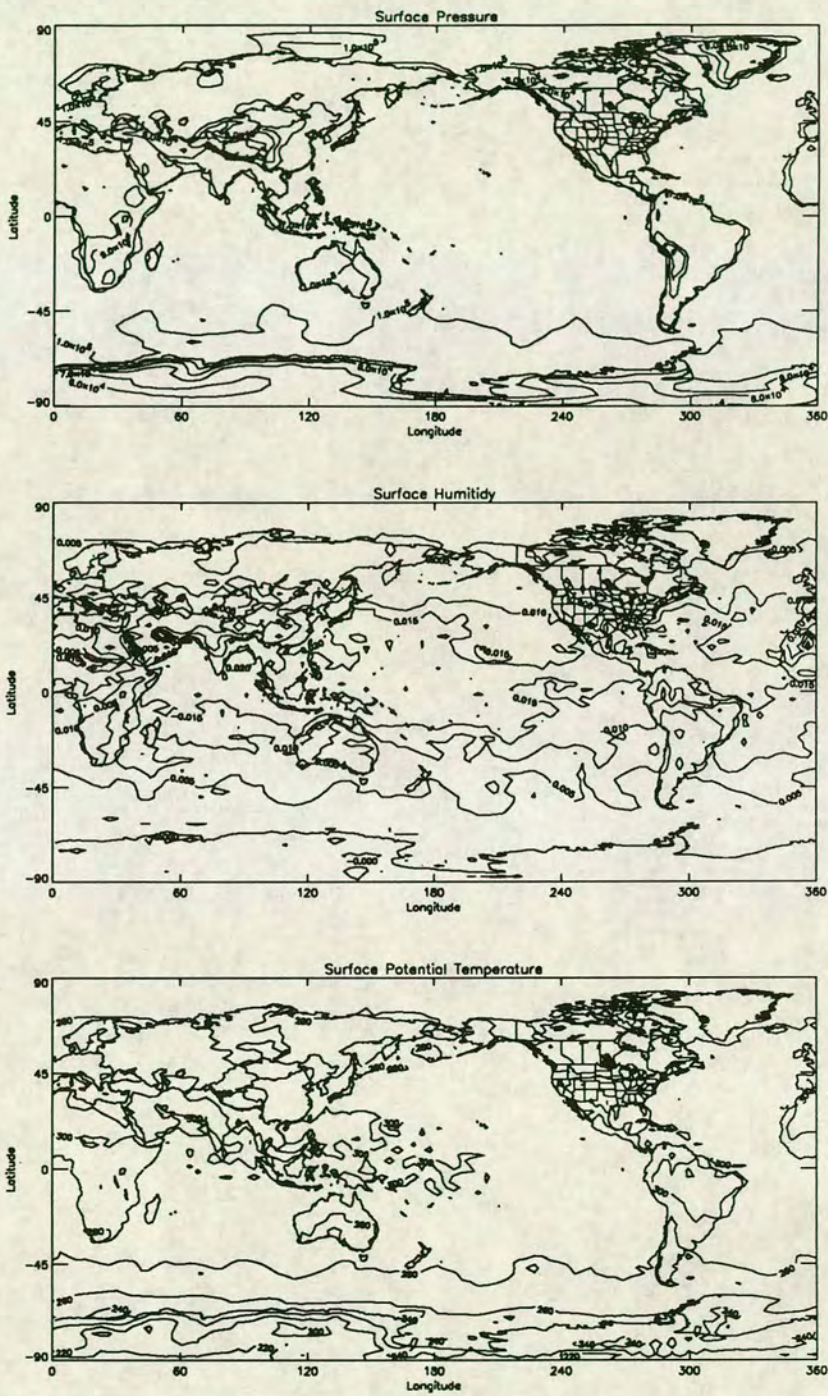


Figure 3-2: Surface data used for the convection scheme

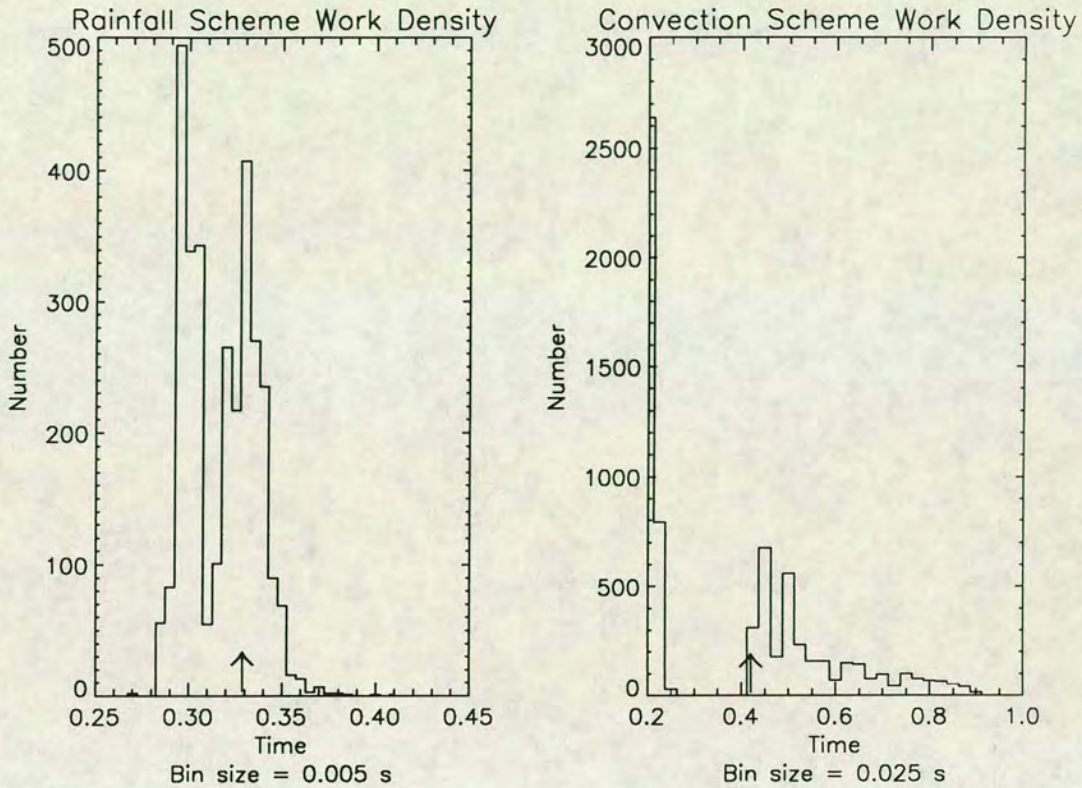


Figure 3-3: Time distribution functions for convection and rainfall schemes. The times shown for the two schemes were obtained by running the convection scheme 100 times and the rainfall scheme 500 times. The times were obtained on a Sun Sparc 2 computer. Modified data values were reset after each call to their initial value. Mean values are shown with an upward pointing arrow.

experiments were carried out on a work station. Figure 3-3 shows the time density functions for both schemes from these experiments. Upward pointing arrows denotes the mean time taken for that scheme.

For the rainfall scheme the variation in time taken is not very large and it is centred around the mean. For the convection scheme there is a large variation in the times taken. Furthermore the distribution is not centred

about the mean; there are a large number of columns which take half the mean time and the distribution has a long tail of large times.

The parameterization schemes act only on a single column and, for the computation of these schemes, there is no interaction between columns. However consideration of the dynamics requires that the atmospheric columns should be mapped to the processors such that two neighbouring columns are on the same processor or on adjacent processors. This is discussed in greater detail in Chapter 5. Figure 3-4 shows the geographical distribution of times taken to compute a column 100 times for the large-scale rainfall scheme. Figure 3-5 shows the times for the convection scheme. Both fields are quite noisy and have been smoothed prior to plotting. For the rainfall schemes the smoothing is a nearest neighbour averaging while for the convection scheme averaging is done with all points that are 2 or fewer grid-points away.

Having examined some properties of the geographical distribution of the schemes' work, the implementation on the ECS and the results of this implementation will now be described. In both cases it was straightforward to take the schemes, coded in Fortran 77, and run them on the ECS. No communications were required and the original code could be used almost as it was. Both codes had a few Cray specific sub-routines, requiring re-writing; this work would be required for any porting of the code.

In a full atmospheric model, the parameterization schemes are run in conjunction with a dynamics stage. One of the effects of the dynamics would be to cause processors to synchronise with their neighbours. Therefore after each processor has computed all the changes to its atmospheric columns it will exchange messages with all its neighbours. This prevents processors with little work to do from "running ahead" of the rest. In addition to these worker processors, which do the computations, there is also an additional processor which will read in the original data, distribute it to the workers,

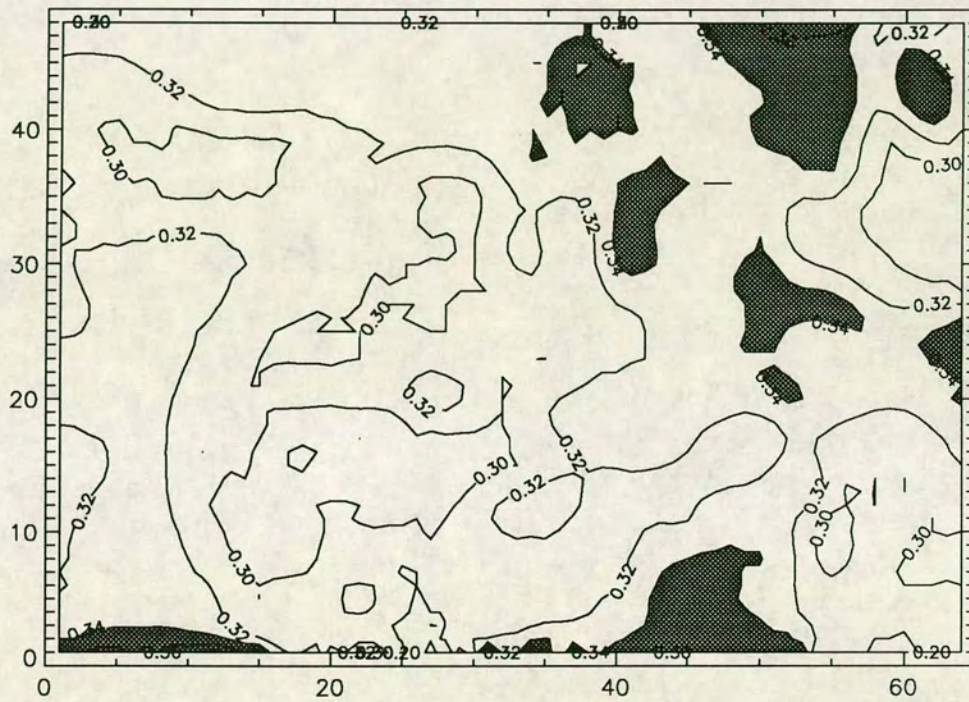


Figure 3-4: Distribution of work for the large-scale rainfall scheme
 Contours are at 0.2,0.3, 0.32, 0.34,0.36 and 0.38 seconds, shading is for times
 greater than 0.34 seconds.

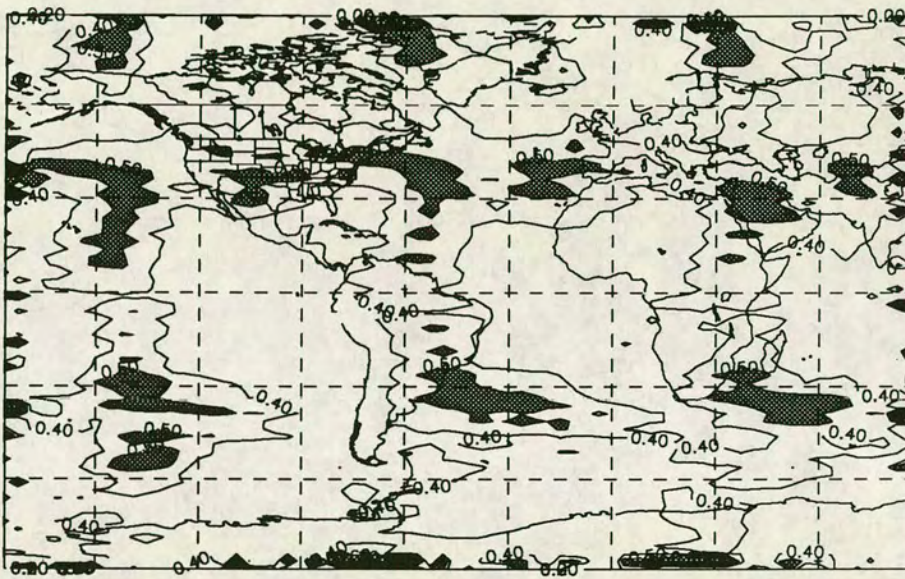


Figure 3-5: Distribution of work for the convection scheme
Contours are 0.2,0.4, 0.5, and every 0.1 of a second, shading is for times greater than 0.5 seconds.

and receive the computed results back again after the worker processors have done all the work required.

Two experiments were carried out for each scheme. The initial data-sets used were the same as those shown in Figures 3-1 and 3-2 for the rainfall and the convection schemes respectively. The data was distributed to the processors with neighbouring columns being on the same processor or on adjacent processors. In addition the number of columns per processor was kept as equal as was possible. Each processor repeatedly carried out the scheme being tested on all atmospheric columns that it contained, synchronised with its neighbours and then reset the data to its initial values. This was done 100 times for the convection scheme and 500 times for the rainfall scheme, in order to increase timing accuracy and reduce startup costs such as data initialization. After this was done the data from all the processors was then sent back to the manager processor. The times measured were from the beginning of the data transmission to the processors until the final piece of data had been received from all processors. In addition to this, a second experiment was carried out in which the overhead associated with sending data out, synchronising and returning data was also measured. These experiments were carried out for differing numbers of processors. For the rainfall scheme these experiments were carried out using up to 64 processors while for the convection scheme up to 128 processors were used. It was not possible, for the convection scheme, to use 4×2 or 2×2 processors due to memory limitations.

The results are shown in Table 3-2. In this table the number of processors is shown as the number of processor columns \times the number of processor rows. In order to isolate the costs of the parameterization schemes the times are corrected for the times taken to transfer data and synchronise between the processors. A plot of these corrected times, multiplied by the number of processors used (the scaled times), is shown in Figure 3-6.

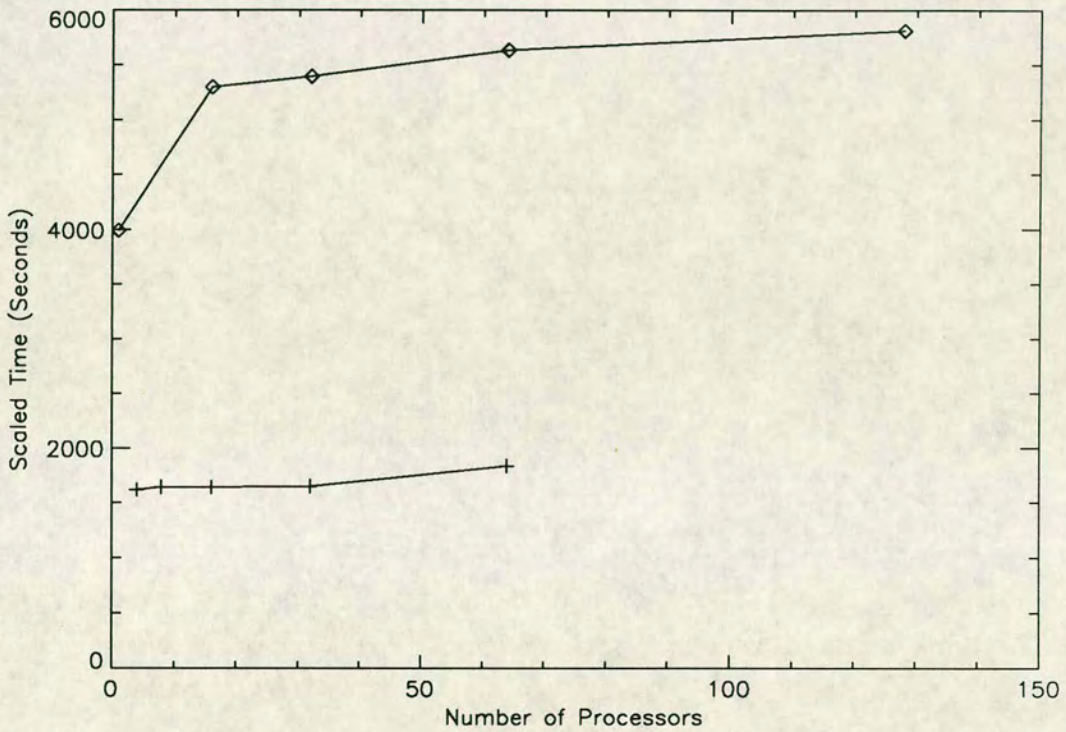


Figure 3-6: Scaled time for parameterization schemes

The above figure shows the corrected scaled times for both parameterization schemes. Diamonds denote the convection scheme (with 100 trials) while the rainfall scheme (with 500 trials) is denoted with crosses.

Table 3-2: Parameterization schemes on the ECS

Number of Processors	Rainfall	Rainfall Correction	Convection	Convection Correction
16 × 8	–	–	55.2	9.8
8 × 8	32.7	4.0	97.6	9.6
8 × 4	58.1	6.5	178.8	9.7
4 × 4	109.0	6.3	340.3	9.2
4 × 2	211.9	6.2	–	–
2 × 2	411.3	6.4	–	–
1 × 1	–	–	3992.9	0.0

Times shown are for 100 trials for the convection schemes and 500 for the rainfall scheme. Times are measured in seconds.

Examining first the results of the convection scheme, shown in Figure 3–6, the difference in scaled speedup when using one processor and using 16 processors is 30%. Beyond this point, as the number of processors increase, the scaled speedup rises slowly to a value approximately 50% greater than the single processor case. This represents an acceptable efficiency for the scheme. Also shown in Figure 3–6 are the results of the large-scale rainfall scheme. For this scheme the scaled time is approximately constant with increasing processor number. There is an increase in scaled time when using 64 processors but it is not clear whether this part of a trend or not.

For both schemes implementation was straight forward and the efficiencies obtained are adequate for operational purposes. The reason why the scaled time dramatically increases, in the convection scheme, when using 16 processors compared to 1 processor may be due to load imbalance, in which some processors have more work to do than other processors. The next section explores this issue.

3.3 Load-Balancing

This section examines the effects of load-balancing on the scaled times of the parameterization schemes used in the previous section. Smoothed plots of the work distribution were shown in Figures 3–4 and 3–5. These can be considered as showing the averaged work over a small region. As can be seen from these figures the spatial coherence of the work distribution, for the convection scheme, is high – that is columns close to one another are likely to require similar amounts of work, and as a consequence of this some processors will have more work to do than others. The aim of load-balancing is to move tasks from processors which have much work, to processors which have little work to do.

Fox *et al.* describe an alternative technique called scattered decomposition. For the meteorological parameterization schemes examined in this chapter, this would involve transferring atmospheric columns to processors such that columns that began close to one another are now far away. The hope is that all processors would now have approximately equal amounts of work to do. However this conflicts with the requirement for the dynamics stage, that neighbouring atmospheric columns should be on the same or neighbouring processors. Thus to carry out scattered decomposition, data would need to be moved large distances across the computer, the computation of the parameterization schemes carried out, and then the data needs to be transferred back to the original processors. The communications cost required make this approach unattractive.

Other authors have carried out some work on load-balancing through task movement, and reported positive results, although for other disciplines (Boillat *et al.*, 1991; Boillat, 1989; Smith and Wilson, 1991). Some work has been done for hypercube architectures (Cybenko, 1989) on load-balancing.

For the purposes of this chapter, a task will consist of a parameterization scheme acting on a single column of the atmosphere. Data will mean the variables for this column and any associated control data. This control data could include, for example, the identifier of the processor from whence the column came.

The load-balancer will consist of the following three modules all running in parallel on the same processor.

1. A compute module; this module will carry out computations on a single task.
2. An exchange module which will exchange tasks with neighbouring processors as well as providing the compute module with tasks.
3. A general router module, which will route tasks so that they return to the correct processor. In many current and proposed parallel computing systems this module is provided as part of the system/library software (Clarke and Wilson, 1990; Meiko, 1991).

Figure 3-7 is a schematic diagram of such a load-balancer. The dynamics module which runs in parallel with these three modules is also shown. There is a possibility of deadlock as a cycle exists between the dynamics module, the exchange module and the general router. This deadlock can be avoided by the dynamics module transferring *all* its tasks to the exchange module before accepting any tasks.

One of the objectives of the implementation of a load-balance/task-exchanger was to reuse the existing serial code easily. The same two parameterization schemes examined earlier in this chapter were used in the work described in this section.

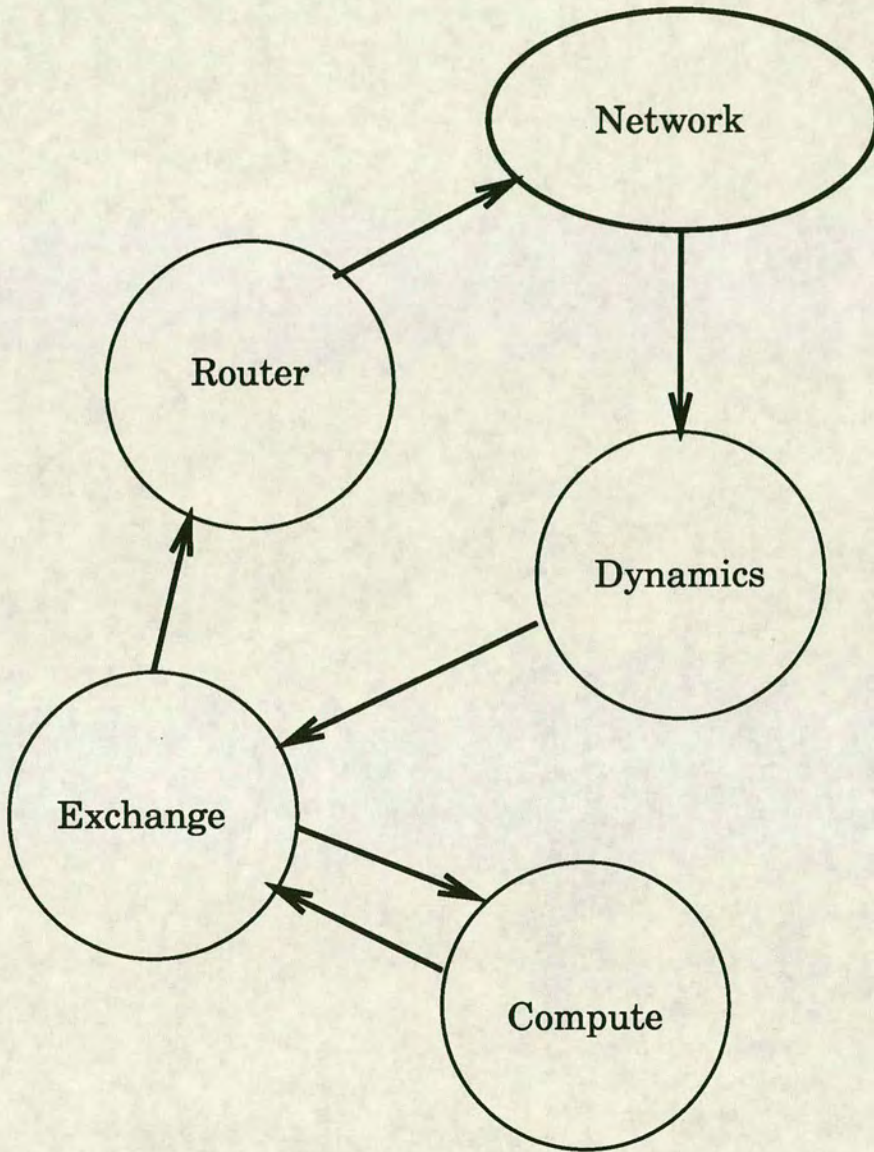


Figure 3-7: Schematic diagram of load-balancer

This figure shows the components of a load-balancer. In addition to the three components of the load-balancer and the dynamics module, the general network is also shown, this comprising the router modules on all other processors.

In order to reuse these existing serial codes, the task-exchanger should treat the parameterization schemes as “black boxes” and assume only that they act on a single column of the atmosphere with no communications between columns. Therefore the task exchanger will see the tasks as merely data to be moved, but which have varying length. This length could change after the compute module is called if some extra data is returned, for example, in the convection scheme, the atmospheric column will have its structure changed and the amount of convective rain and snow will be returned.

There is no general *a priori* method of determining how long a task will take except by carrying out the computations (*cf* the “Halting Problem” Turing (1936)). It is for this reason that the work is measured by the number of tasks a processor has left to compute. Processors on which the tasks require little work, after some time, will have less tasks to do than processors where the tasks require much work. Exchanges of tasks between processors will be done periodically. The natural time scale for the exchange module is the time taken by the computations on the task. In this time, a neighbouring processor could be polled to see if an exchange of tasks would improve the balance of tasks locally. The number of tasks, and thus the estimate of the work left to do, per processor would be more ^{nearly} equal after the exchange than before.

There are two possible ways to carry out the polling of a neighbour. A processor can poll a neighbour when it has *less* than a critical number of tasks left to do. This is called a request strategy as it is likely that the processor will receive tasks from its neighbours. The alternative is to poll a neighbour when the processor has *more* than a critical number tasks left to do. This is a send strategy as it is likely that the processor will send tasks to its neighbours.

The exchange module will carry out the following set of alternatives

repeatedly, the first one ready will be selected. If more than one is ready at the same time then the highest numbered one will be taken.

1. A processor has sufficient tasks. It then chooses at random a neighbouring processor to poll or to not poll any of its neighbours. The reason for the possibility of polling none of the neighbours is to avoid live-lock in which a processor repeatedly polls its neighbours. If a neighbour is selected then that neighbour is told how many tasks the processor has, and then the processor waits for the neighbour to reply with the number of tasks it has. Both processors know how many tasks each has. The one with the greater number of tasks then transfers tasks to the other to balance up the number of tasks. The number of tasks transferred is limited so that the time to exchange tasks is not too great. This is done in order that the compute module has no work to do while the task exchange is done.
2. Receive a message from a neighbouring processor telling the processor how many tasks the neighbour has, then tell the neighbour the number of tasks this processor has and proceed as in item 1.
3. Receive some tasks from the dynamics module. This alternative will happen infrequently compared to the previous two.

After one of these alternatives have occurred the balancer will be in one of four states.

1. It has tasks and the computational module is processing a task. In this case the processor will wait for the computational module to complete its task, at which point the task exchanger will receive the computed task and pass it on to the router. It will then give a task to the computational module.

2. It has no tasks and the computational module is processing a task. The balancer will then proceed as in item 1.
3. The computational module is empty and the task exchanger has tasks. In this case a task will be given to the compute module.
4. The computational module is empty and the task exchanger has no tasks. Nothing will happen.

After one of the above, the probability distributions for selecting a neighbour or not polling a neighbour will be adjusted to represent the increased information about its neighbours. In the implementation, the probability of selecting the neighbour just polled would be set to zero and the probabilities of selecting any another neighbour would be relaxed towards their mean values. The probability of not polling a neighbour would be set to whatever was required to make the sum add to 1.

Under certain circumstances, the processor could live-lock. This could happen if the processor has zero tasks, is following a request strategy and all its neighbours have no tasks. It would poll a neighbour and exchange no tasks (as its neighbour has none). After this it would be in state 4 and do nothing, before starting the cycle again.

This cycle would only be broken when it received some tasks from either its neighbours or the dynamics module. In this live-lock state, the exchange module could consume much of the processors CPU time. If all the processors have completed the sub-grid tasks and so none have any tasks, then all of them could be in this live lock state. If they all consumed all the processors CPU time polling each other then this state would last indefinitely as the dynamics module would never be able to complete, and pass tasks to the exchange module.

For this reason a request strategy is not as suitable as a send strategy. However a remedy is possible: if the exchange module is ever in state 4, then it should suspend itself for some time, thus allowing the rest of the processor to do useful work. After this has been done the suspension time should be doubled. If the processor gets into any of states 1–3, the suspension time should be reset to its original value.

Times for the load-balancer were measured in the same way as was done in Section 3.2. As well as load-balancing after each scheme had been computed on all columns on a processor, all processors would then synchronise with their neighbours in order to stop “running” ahead. The balancer described in this section was implemented in Occam and data was copied between processes on the same processor. This made the implementation easier and also made modifications easier. However the cost of this data transfer is high and a operational implementation should not do this. As only two processes, the compute process and the exchange process, could modify the data, pointers to the data are all that should be manipulated by all the other processes. For each scheme two experiments were carried out, one in which the times were measured using the load-balancer and another in which the total times taken to synchronise between processors and copy data between the dynamics process and the worker process were measured. The times from the second experiment were used as corrections to the first times in order to estimate what times would have been taken by an optimised version of the balancer software. The number of processors used in these experiments was the same as for the experiments in the previous section. The results of the experiments are shown in Table 3–3.

Figure 3–8 shows a plot of the scaled time for these corrected results. The times taken when no load-balancing was used is shown as dotted lines, in order to make comparison easier. For the convection case a very small gain is observed for up to 32 processors but beyond this the load-balancer

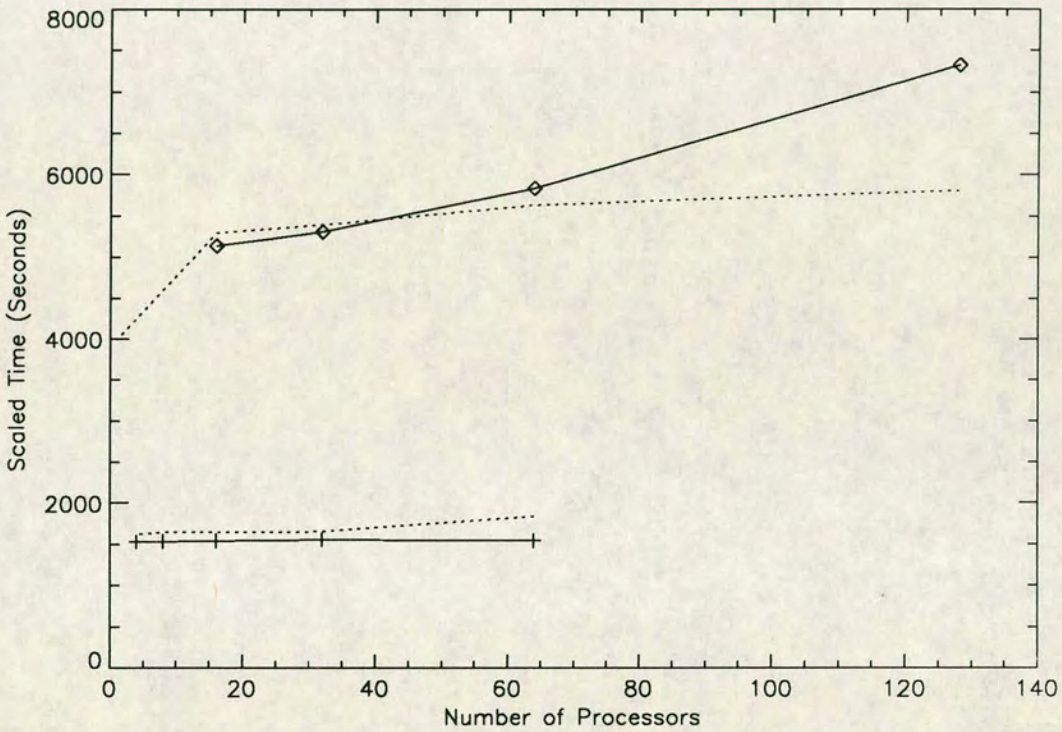


Figure 3-8: Scaled speedup with load-balancing

The results for the convection scheme are denoted by diamonds while the rainfall scheme is shown with crosses. Dotted lines show the results for the non load-balanced cases. Times taken are for 100 trials for the convection scheme and 500 for the rainfall scheme.

Table 3-3: Balancer results

Processors	Convection Scheme		Rainfall Scheme	
	Balancer	Copy correction	Balancer	Copy correction
16 × 8	74.9	17.7	—	—
8 × 8	116.2	25.0	55.3	31.2
8 × 4	206.2	40.2	103.8	55.3
4 × 4	392.4	71.1	200.4	104.2
4 × 2	—	—	393.4	201.9
2 × 2	—	—	779.8	397.1

causes the times taken to increase. For the rainfall case the load-balancer produces a small gain.

It seems likely that the differences between the two cases are due to the different distributions of work over the computational domain. For the convection case work is distributed more unevenly than in the rainfall case. In the former case the load-balancer needs to move tasks farther than in the latter to equalise the distribution. The time to do this is too great. However if the work was distributed more evenly, as in the rainfall scheme, the processors are already in approximate balance and therefore load-balancing will be of little benefit.

From this study it is concluded that load-balancing is unlikely to be of much benefit for the parameterization part of atmospheric models.

3.4 Conclusion and Discussion

This chapter examined parameterization schemes in the context of their implementation on parallel computers. First it concluded that it was unlikely that the technique of "gathering and scattering" would be efficient on data-parallel computers. In addition it was concluded that rewriting of existing codes would be needed to use data-parallel machines. This is in contrast to MIMD computers, such as the ECS, where implementation of existing codes is straightforward and should be trouble free if the code is written in a standard language, and using no extensions, such as Fortran 77.

Two schemes were ported to the ECS; a convection scheme, and a large-scale rainfall scheme. Measurements of their scaled times were made. For the convection scheme the scaled time was about 50% higher than the 1 processor case when using 128 processors. The scaled time increased by about 20% from 16 processors to 128 processors. It was postulated that this was due to load-imbalance. For the rainfall scheme the scaled time was approximately constant or increasing slowly with increasing numbers of processors.

A prototype load-balancer was designed and implemented. After various corrections had been made to the scaled times observed it was found that the load-balancer caused an increase in time taken for large numbers of processors or a small decrease for small numbers of processors. It was therefore concluded that load-balancing was unlikely to be of any benefit for a realistic model on a massively parallel computer.

The load-balancer is in effect a process which moves tasks between processors; there are other circumstances where task movement between processors may be needed. The size of the compiled model for each scheme is

quite large and likely to increase as the model complexity increases. Moreover it may be extremely expensive to provide sufficient memory on each processor to allow all the parameterization schemes to be present on every processor. In these circumstances, it may not be possible to have a copy of each parameterization scheme on every processor. If this is correct then it could be envisaged that each processor would contain only some of the parameterization schemes and that data would need to be moved between processors to allow all the schemes to act on any single column of the atmosphere. The number of processors having a particular scheme should be inversely proportional to the mean time that the scheme takes in order to minimise the load imbalance problem in what is in essence a pipeline. See Section 4.2 for a discussion on pipelines. In practice this may not be so easy. The main problem will be the expense of the additional communications to move the tasks between the processors.

In addition, in order to ensure repeatability, a task should visit the different schemes in a *deterministic* order. If the order is the same as the serial model being considered then verification would be made easier. In many parameterization schemes there is, on physical grounds, a pre-determined order in which they should be carried out. For example the radiation scheme requires information on cloud properties which are partially generated by the convection scheme; so the convection scheme must be called prior to the radiation scheme.

Chapter 4

Spherical Harmonic Methods

4.1 Introduction

This chapter and the next examine two different methods used to simulate large-scale atmospheric flow. The next chapter considers grid-point methods; this one will consider the spectral method.

The spectral method is widely used by the Meteorological community. Its use ranges from low resolution climate models (James and Hoskins, 1990; Bourke, 1988) to high resolution numerical weather prediction models (Girard and Jarraud, 1982; Hogan and Rosmond, 1991). Courtier and Geleyn (1988) have suggested that the spectral method could be suitable for local area models if a suitable conformal transformation of the globe is chosen. Their use is not likely to be discontinued in the near future. Therefore an efficient parallel algorithm for this method is required. This chapter will detail such an algorithm. The details of the spectral method have already been described in Chapter 1. The spectral method requires data to be transformed from grid-point space to spectral space and *vice versa*. These transforms are global transformations and require that data be communicated across the

entire computational domain. Some models also require some local communications in parts of the method, an example of this is described in Bourke (1974). The model whose implementation is described in this chapter has been in use for some time and was first described in Hoskins and Simmons (1975) and will be referred to throughout this chapter as the Reading Model. This model uses the three-dimensional primitive equations.

An overview of this chapter will now be given. The next section will provide a description of the method used to carry out a parallel implementation, namely, a pipeline. The following two sections explain in detail how the two components of the spectral transformation, the fast Fourier transform and the Legendre transform are performed. Section 4.5 shows how the components are joined together and gives an analysis of the algorithm's scaling behaviour. Next, details of the implementation of the algorithm on the ECS are given. Results of this implementation are then presented and finally some conclusions are made.

Swarztrauber and Sato (1990) described an implementation of the pseudo-spectral method for the shallow water-equations for the CM-2. In the pseudo-spectral method the basis functions are complex exponentials rather than the spherical harmonics used in global spectral methods. Carver (1988) implemented the shallow water equations on the DAP computer using the spectral method. The problem Carver (1988) solved was in effect how to pack the data onto the processors of the DAP. His algorithm is more inefficient for the inverse transform than for the forward transform. MIMD computers like the ECS are more flexible than data-parallel computers such as the DAP and the algorithm described in this chapter is equally efficient for both the forward and inverse transformations. The algorithm is for a MIMD computer with fixed valency, which is configurable to any desired topology consistent with this requirement.

4.2 Pipelines

The approach that was chosen for implementing the spectral method was a pipeline because several of the sub-components of the spectral transform could be implemented naturally in that way. A pipeline can be thought of as algebraic decomposition in which a complex task is broken into several tasks. If a sequential operation consists of several functional units and this operation is done several times on different pieces of data then it may be efficient to split up the operation into its component functional units. Each unit will operate on a piece of data, pass this computed data out to the next part and take in another data element to act on. Figure 4-1 is a simple analogy of this where a brick wall is built using a pipeline.

In this example the task of building a wall is broken into four functional units. A task for this case is adding one brick to the wall. The four units are in order;

1. Prepare the brick.
2. Apply mortar to the brick.
3. Place brick on wall.
4. Tamp brick down.

Bricks (tasks) will be taken from a pile and prepared by the first worker in stage 1. She will then pass this brick on to the next person in the pipeline. The process will continue. After, in this case, five time-steps the first brick will be added to the wall. From this time on a brick will be added to the wall every timestep. If each stage in the pipeline takes a different amount of

Building a Wall with a Pipeline

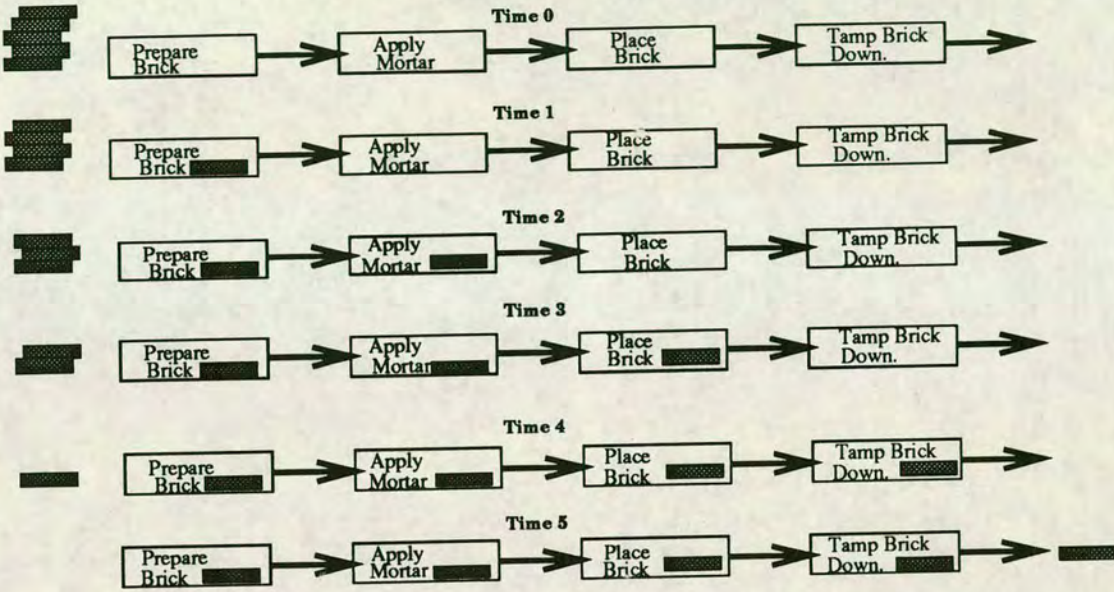


Figure 4-1: A simple pipeline

This figure shows, schematically, how a brick wall could be built using a pipeline. Note that the first brick will appear on the wall after 5 timesteps.

time then some workers will spend some of their time doing nothing useful. The time between bricks being added to the wall will be given by the time taken by the slowest worker or the time taken by the longest task..

4.2.1 General Pipelines

A few preliminary definitions will now be given. The timestep, T , is the time interval between data elements at the output end of the pipe. The startup time, S , is the time it takes for the first data element to travel the length of the pipeline. A pipeline has a very simple work and communications structure; each stage in the pipeline will take in input data, process it and output it to the next stage in the pipeline. Synchronisation between stages of the pipeline occurs at input or output operations. If one stage is not ready

to synchronise with another stage, then the later stage will be forced to wait until such time as the former stage is ready to proceed.

Consider now a pipeline with timestep \mathcal{T} . Another stage, with timestep \mathcal{T}' , is then added to this pipeline. The timestep of the composite pipeline is now the maximum of \mathcal{T} and \mathcal{T}' .

Having shown how to combine the timesteps of the components of a pipeline to obtain a timestep for the pipeline, a means of computing the timestep for any given component will now be considered. At this stage the hardware characteristics are relevant. Two models are considered; first, hardware like the Transputer where multiple communications and one set of calculations can proceed concurrently; second, hardware which can only perform one function at a time i.e. it can communicate along one link or compute. For future reference, these two cases will be referred to as concurrent and non-concurrent respectively. Figure 4-2 shows possible work, communication and idle time patterns for both the concurrent and non-concurrent cases. For the concurrent case, define \mathcal{T}_i^I , \mathcal{T}^W and \mathcal{T}_k^O as the time taken to do input on channel i , computation and output on channel k respectively, then \mathcal{T} is $\max(\{\mathcal{T}_i^I\}, \mathcal{T}^W, \{\mathcal{T}_k^O\})$ with labels i and k ranging over the possible inputs and outputs. In the example shown in Figure 4-2 there are three inputs and one output for stage four of the pipeline. For the non-concurrent case, in contrast, \mathcal{T} is $\sum_i \mathcal{T}_i^I + \mathcal{T}^W + \sum_k \mathcal{T}_k^O$.

One final complication will be considered before showing how the startup time for the pipeline is computed: it is possible that a stage will be receiving input from several different sub-pipelines which are all proceeding in parallel. The timestep for each of those pipelines is \mathcal{T}_i . For the concurrent case then the timestep for the composite pipeline will be $\max(\{\mathcal{T}_i\}, \mathcal{T})$. The argument that leads to this result is the same as that presented earlier. For the non-concurrent case the analysis proceeds as before with the time step being the same as before. The processing of inputs from the parallel sub-

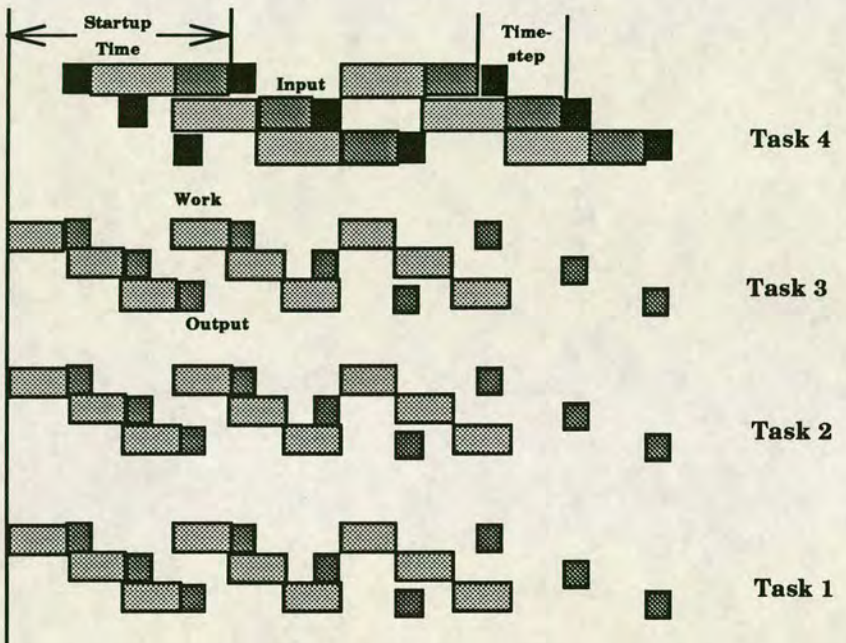
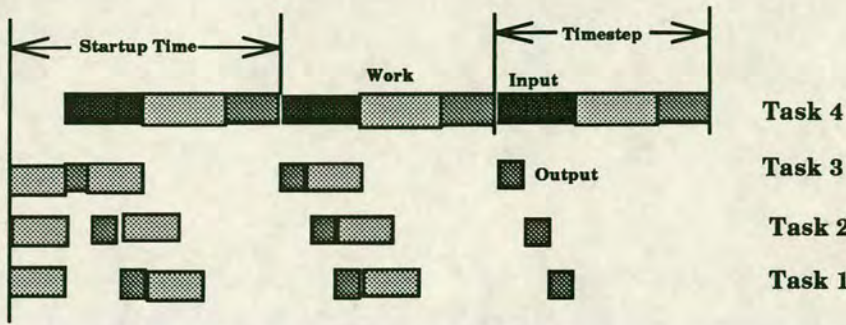


Figure 4-2: Task work patterns

The above figure shows the work and communication patterns for a simple pipeline. The top portion of the figure shows this for hardware which cannot carry out simultaneous communications and calculations, while the bottom portion of the figure shows this for hardware which can do concurrent communications and calculations. Time flows from left to right, computations (work) are shown by a light shaded region, input by dark shading and output by intermediate shading. The pipeline consists of three identical processes (tasks 1 to 3) which do some work and send the results onto another process (task 4). The process, when it has received messages from tasks 1 to 3 does some computations and then outputs the results.

In the concurrent case, communications and work can all proceed in parallel thus giving up to three streams for each task. For the non-concurrent case note the delay in the starting of the communications for tasks one and two, as only one communication can proceed with task four at a time.

pipelines must be done in the same order in each cycle or the time between inputs will be increased from $\sum_i T_i^I + T^W \sum_k \tau_k^O$.

Finally, considering the startup time, since output in a stage is synchronised with input in the next stage, the time taken is the sum of work and output times for the concurrent case. For the non-concurrent case, the analysis is much more dependent on the situation, as the extra communications introduce delays on inputs. If all the inputs are coming from sub-pipelines, the first input adds nothing to the startup time but the remaining ones do. To summarise, the computation of the timestep for the entire pipeline is the maximum of the timestep for each stage of the pipeline. The startup time for the pipeline is at least the sum of the output and work times for each stage.

The efficiency of the pipeline can be computed from the startup time and the timestep for the pipeline. This pipeline efficiency is given by the following expression

$$V_e = \frac{Td}{Td + S} \quad (4.1)$$

where d is the number of tasks that the pipeline will process.

4.2.2 Pipelining the Spectral Method

Next, considering the spectral method, a flowchart for one iteration is given in Figure 4–3. This flowchart is at the level of the different blocks of work within the spectral transformation. This flowchart can, at least conceptually, be turned into a pipeline by choosing the transformation of the different Gaussian latitudes to be tasks. Figure 4–4 shows this and the computational complexity of each stage in this pipeline.

In order that each stage be balanced, as the timestep is given by the slowest part within the pipeline, the number of processors in any stage

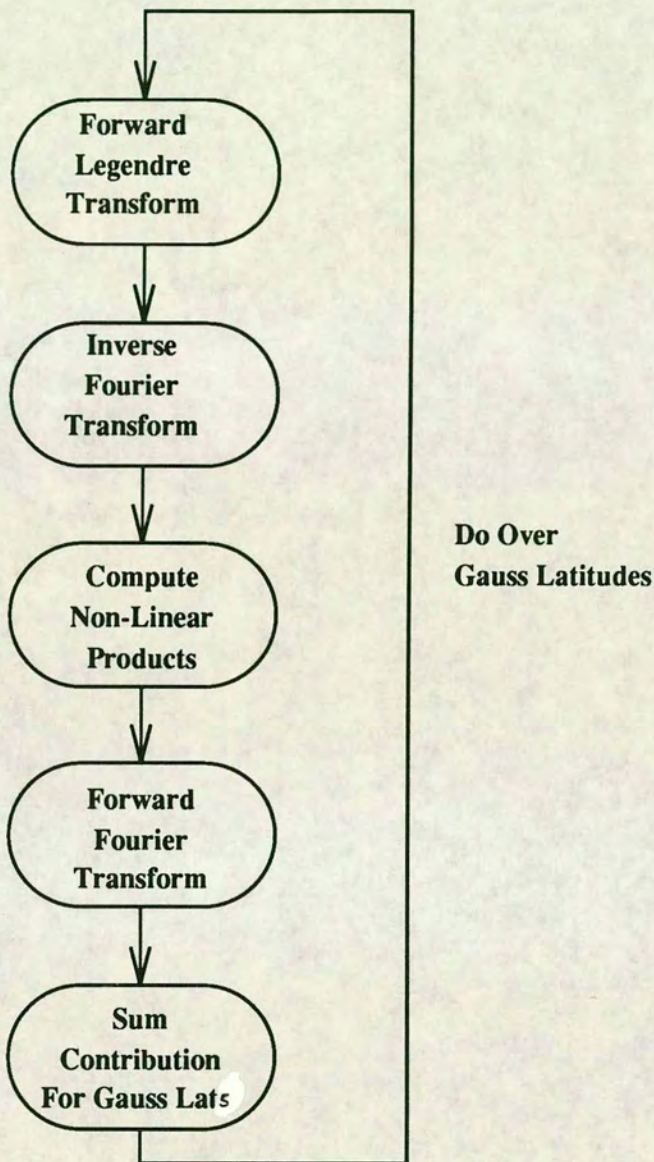


Figure 4-3: Flowchart of spectral transform

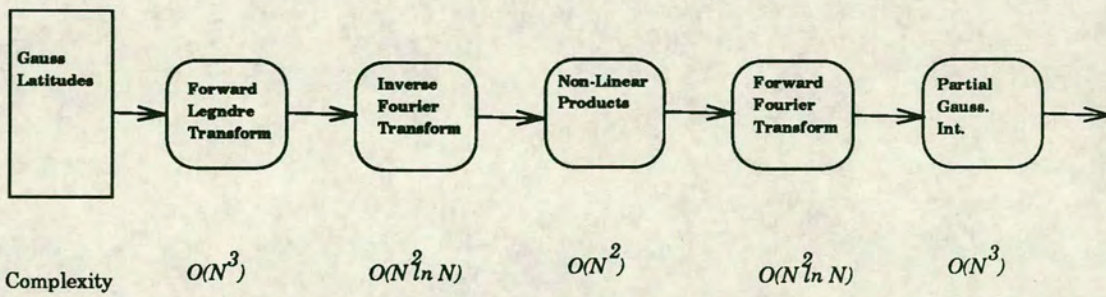


Figure 4-4: Pipeline and computational complexity for spectral method

Table 4–1: Required parallel complexity

Stage	No. of Processors
Time step Inc.	$O(N^2)$
Forward Lgnd.	$O(N^2)$
Inverse FFT	$O(N \ln N)$
Non-linear Products	$O(N)$
Forward FFT	$O(N \ln N)$
Gaussian Int	$O(N^2)$

N is proportional to the size of the spectral truncation (either Rhomboidal or Triangular). The table shows the number of processors that each stage in the pipeline should have in order that it should remain balanced. The numbers were obtained by taking the serial computational complexity and dividing by the number of tasks.

should be proportional to the computational complexity. Table 4–1 shows the constraints that are required on the number of processors at each stage. The chapter will proceed by examining the components of the pipeline and computing startup times and timesteps for each of them. These will be combined to compute a startup time for the entire pipeline, being the sum of all the individual startup times, and a timestep for the entire pipeline, being the maxima of each of the timesteps for the individual components.

4.3 The Fast Fourier Transform

This section will consider the unit of the pipeline that carries out the fast Fourier transform (or FFT) component of the spectral transform and in particular a FFT of radix two. This unit is itself broken down into a pipeline,

and there are three types of units within this sub-pipeline. The numerical details of the FFT were described in Chapter 1. In order to utilize as much of the inherent parallelism of the algorithm as possible it is necessary to use two-dimensional decomposition. Therefore a distributed FFT is required.

As will be shown later, this part of the algorithm is the bottleneck in the entire problem. The number of operations per site that the FFT requires is small, at most four per site. If the communications speed of the processor, relative to the computational speed, is slow then the fraction of time doing useful computations could be unacceptably low. If this is true then the speedup of the total algorithm will be very slow and the algorithm that will be presented in this chapter should not be used.

The important parts of the FFT, in terms of its implementation on a massively parallel computer, are the recurrence relationship which demands a particular kind of communications pattern and the fact that information stored on sites with labels in a particular order will need to move to sites whose labels are obtained by performing bit-reversal on the original labels. This is discussed below. If, as in the Reading model, no local communications are required in either grid-point space or spectral space, then it is not necessary to carry out a bit-reversal or reordering process. For future reference, the bit-reversing operator for λ bits will be denoted by $\widetilde{}^\lambda$ i.e. \widetilde{M}^λ is the number formed by reversing the bits of a binary representation, length λ , of M . M must satisfy the constraints $M \geq 0$ and $M < 2^\lambda - 1$. If M is equal to $\sum_{i=0}^{\lambda-1} m_i 2^i$ then \widetilde{M}^λ is $\sum_{i=0}^{\lambda-1} m_{\lambda-(i+1)} 2^i$ with $m_i = 0$ or 1 .

For each timestep of the spectral method the inverse transformation is carried out first, followed by computation of the sub-grid schemes and the effects of advection, followed by the forward transformation. In most present models no communications are required between sites in grid-point space with different horizontal co-ordinates. The use of semi-Lagrangian methods

to compute the advection step in grid-point space would change this. For a description of this method see, for example, Ritchie (1987).

The method used to compute the FFT is to form a pipeline of width N (N is an integer power of 2) and length $\log_2 N + K$, K being a constant number of processors, the value of which depends on the size of "internal" FFT. Details are discussed later. The width is the number of processors which will act on one Gaussian latitude. Each stage of the pipeline will compute the recurrence relationships using whichever one of eqns 1.49, 1.50 is appropriate. The number of points on a processor, L with $L = 2^p$, is M/N (M being the total number of points for the Fourier transform with $M = 2^q$, $q \geq p$)

The mapping of the \overline{F}_i^k to the processors is now considered. This takes the form of a mapping, dependent on l , from the i, k indices to a single index j . Manipulating this mapping of i and k makes it possible to reason about the communications necessary between the processors. The processor identifier is given by j/L using integer division.

The communications patterns required for Equations 1.49 and 1.50 in terms of the index j is,

$$\{j(l, k, i), j(l, k, i + 2^{q-(l+1)})\} \mapsto j(l+1, k, i) \quad (4.2)$$

$$\{j(l, k, i), j(l, k, i + 2^{q-(l+1)})\} \mapsto j(l+1, k + 2^l, i) \quad (4.3)$$

with the notation meaning that in order to compute values at a site labelled $j(l+1, k, i)$, sites labelled $j(l, k, i)$ and $j(l, k, i + 2^{q-(l+1)})$ are required. This notation is based on that introduced by Carver (1990). The mapping of l, k and i to j should have the property that all sites on a processor should all communicate with the same processor. This is essential for the target architecture (T800 Transputer), which has only a fixed valency. On other architectures with more general routing strategies there will be a gain in efficiency if this

restriction is followed, as the effect of communications startup time is reduced. This gives the constraint on the mapping for values of j such that $(j + 1)/L = j/L$ (using integer division),

$$\{j(l, k, i) + 1, j(l, k, i + 2^{q-(l+1)}) + 1\} \mapsto j(l + 1, k, i) + 1 \quad (4.4)$$

with $i \in \{0, \dots, 2^{q-(l+1)-2}\}$ and $k \in \{0, \dots, 2^l - 1\}$. It is also useful for the transform to not use any extra memory, this leads to the following two requirements for the transform

$$\begin{aligned} j(l + 1, k, i) &= j(l, k, i), & (4.5) \\ \forall k \in \{0, \dots, 2^l - 1\}, & \quad \forall i \in \{0, \dots, 2^{q-l} - 1\} \end{aligned}$$

$$\begin{aligned} j(l + 1, k + 2^l, i) &= j(l, k, i + 2^{q-(l+1)}), & (4.6) \\ \forall k \in \{0, \dots, 2^l - 1\}, & \quad \forall i \in \{0, \dots, 2^{q-l} - 1\} \end{aligned}$$

Appendix C.1.1 shows that the following expression satisfies these requirements

$$j(l, k, i) = 2^{q-l} \tilde{k} + i. \quad (4.7)$$

Values of k are given by $\overline{[j/2^{q-l}]^l}$ and i is given by $j \bmod 2^{q-l}$. These are obvious from the restrictions on the domains of i and j . This expression has the properties that $j(q, k, i) = \tilde{k}^q$ and that $j(0, k, i) = i$, that is the indices start ordered at the beginning of the transform and are bit-reversed at the end.

This mapping strategy leads to the following communications pattern for Equations 1.49 and 1.50 respectively.

$$\{j, j + 2^{q-(l+1)}\} \mapsto j \quad (4.8)$$

$$\{j, j + 2^{q-(l+1)}\} \mapsto j + 2^{q-(l+1)}. \quad (4.9)$$

Communications will be required if the difference between j and $j + 2^{q-(l+1)}$ is greater than the number of points on the processor. As there are 2^p

points on the processor this occurs when $q - (l + 1) \geq p$. The natural, and well-known topology (Hockney and Jesshope, 1988) that has this required communications pattern is shown in Figure 4-5.

When no communications are required between processors, then an internal FFT can be done. This internal FFT is a purely serial operation, involving no communications between processors, and thus all the well known optimisations for serial computers can be used, such as those described by Temperton (1983). It is possible to decompose the internal transform over processors in a pipeline fashion with, for example, the first processor computing the first two stages of the FFT, and the second the remainder.

The FFT as described here requires some extra operations; the computation of $\omega_{2^{l+1}}^k \overline{F_{i+2^q-(l+1)}^{k+2^l}}^{(l)}$ is done twice in the distributed FFT, once on each processor involved in computing the FFT, using whichever one of Equations 1.49 and 1.50 is relevant. This makes the efficiency of the FFT approximately 50% less than it would be in a serial implementation. Fox *et al.* (1988) proposed a more efficient algorithm, suitable for a computer with a hypercube architecture. In their algorithm an exchange of data between processors would be done such that each processor would do several stages of the FFT, none of which would require communications, and then another data exchange would be done and so on. In contrast to this, the algorithm presented here has each processor receive some input data, do one stage of the FFT and then output the results to the next processors in the pipeline.

All computations concerning the time-step and startup times for the pipeline in the remainder of this section and of this chapter are expressed per real word relative to the time to do one floating point operation. For the pipeline as described, with communications between processors, the timestep is $\max(\tau, 4)$ and $4\tau + 4$ for the concurrent and non-concurrent case respectively. The startup times are $\tau + 4$ and $4 + 2\tau$ for the two cases, where

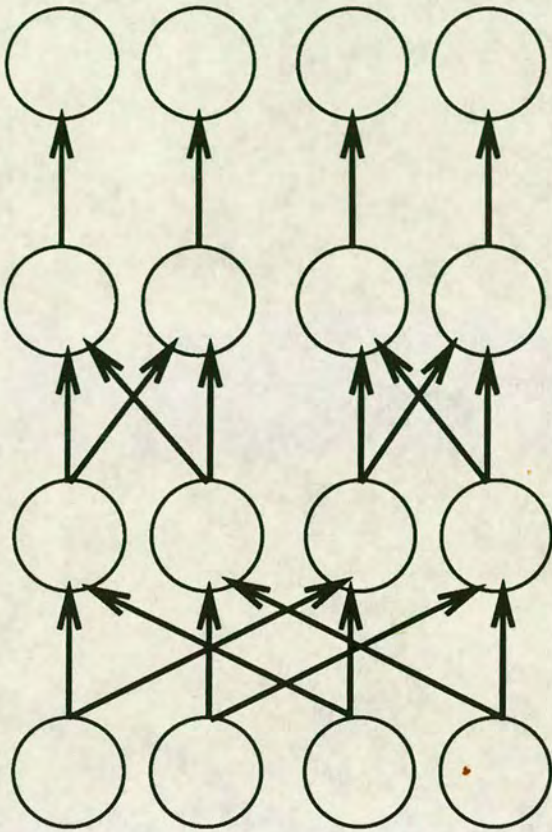


Figure 4-5: The processor topology for the fast Fourier transform

This diagram shows the processor topology required to compute the fast Fourier transform. The width of the pipeline is 4. The line of 4 processors at the base of the pipeline feed data into the remainder of the pipeline but do no computations for the FFT. This example of the topology will when the pipe is full, process 3 FFTs concurrently.

τ is the number of floating point operations that can be done in the time to transfer one real word to a neighbouring processor.

4.3.1 Balancing the Fourier Pipeline

At this point the balancing of the Fourier pipeline will be considered. All the distributed parts of the transform take the same time. The time per point

is fixed and cannot be varied. That is $\max(\tau, 4)$ for the concurrent case and $4\tau + 4$ for the non-concurrent case. The only free parameter available is the number of processors that will compute the internal FFT. This will require pc operations per site; c is the number of operations per site for one stage and 2^p is the number of points on a processor. The length¹, l , of the internal FFT is, in order that the pipeline be balanced,

$$l \leq \max(\tau, 4)/c \quad \text{and} \quad l \leq (4\tau + 4)/c \quad (4.10)$$

with the left hand expression referring to the concurrent case while the right hand one is the non-concurrent case. This expression can be obtained by considering the timestep for the two components of the FFT pipeline: the part where communications are required has a value of \mathcal{T} given by $\max(\tau, 4)$ or $\tau + 4$ depending on the ability to carry out concurrent communications and calculations: the “internal” FFT will have a time which depends on the number of stages, p , the number of processors, K , which these stages are distributed over, and the computational complexity, c , of each stage.

The number of extra processors that should be used is determined by the need to keep the pipeline balanced. If there are l stages on each processor then $lc \leq \max(\tau, 4)$ which requires that $l \leq \max(\tau, 4)/c$. There are p stages and for the number of processors, K , to be large enough to make Equation 4.10 true they should satisfy $K \geq pc/(\max(\tau, 4))$. A similar analysis for the non-concurrent case gives that $K \geq pc/(4\tau + 4)$.

The startup time for the internal FFT is $cl + K\tau$ for the both cases, as there is only one input and output from each stage. This result comes from the computations of the FFT and the communications between the K processors that make up the internal transformation.

¹Length here means the number of stages in the pipeline on one processor.

4.3.2 Inverse FFT

The previous discussion has only considered one part of the transformation, the inverse FFT. The forward transformation is also required. The traditional, serial method, is to explicitly carry out the bit-reversal after the transformation, do computations with the data thus ordered, then do the other FFT using the same algorithm as the first. Finally the data is again moved to bit-reversed locations to return it to an ordered state. The following quote from Press *et al.* (1986), with some text emboldened, by the present author, to show emphasis, illustrates this.

“You can use decimation-in-frequency algorithms (without its bit reversing) to get into the ‘scrambled’ Fourier domain, do your operations there, and then use an inverse algorithm (without *its* bit reversing) to get back to the time domain. While elegant in principle, the procedure does not in practice **save much computation time**, since the bit reversals represent only a small fraction of an FFT’s operations count ...”

In the distributed FFT, on the fixed valency architecture, bit-reversal will require many communications. Sites that need to be moved onto the same processor will need to come from processors that are widely separated. For example consider the site indices 0 and 1. Under the bit-reversal operation of length λ they will be mapped to 0 and $2^{\lambda-1}$ respectively. Thus the time taken by the algorithm is significantly increased. For the model being considered, it is not necessary to carry out the bit-reversal operation as there are no communications between adjacent points in grid-point space.

The forward part of the transform (or inverse if the indices start ordered in grid-point space) is very similar to the inverse transform. The main difference is that the transform starts bit-reversed and in this case an index

j' which starts with i bit-reversed is used. In this case k and i are given by the following expressions,

$$\begin{aligned} k &= (j^{\tilde{q}} \bmod 2^l) \\ i &= j^{\tilde{l}} \end{aligned} \quad (4.11)$$

The communications pattern that this leads to is,

$$\{j'(l, k, i), j'(l, k, i) + 2^l\} \mapsto j(l+1, k, i) \quad (4.12)$$

$$\{j'(l, k, i), j'(l, k, i) + 2^l\} \mapsto j(l+1, k, i) + 2^l \quad (4.13)$$

with $l = 0$ being grid-point space.

4.3.3 Real Fourier Transforms

Values in grid-point space are real not complex, and time can be saved in computing the transform by utilising this. This subsection explores these issues and provides algorithms for this class of Fourier transforms. When the result of the transform is real, then the complex variables satisfy the following relationship, where there are M complex values in the transform.

$$F_m = F_{M-m}^* \quad \text{or} \quad F_m = F_{-m}^* \quad (4.14)$$

Subsection 1.4.3 explained how the real transform could be done efficiently by packing two transforms into one transform. There are two distinct problems, which need slightly different algorithms. The first one, is the case where the complex values start in a normal ordering of their indices and the real values have bit-reversed indices. This is the problem that needs solving for the fast Fourier transform that has been described previously. The other problem is the case where the real values have a normal ordering of their indices and the complex values have their indices bit-reversed.

There are many models that require that the data is in a normal ordering in grid-point space. Any semi-Lagrangian method will require this. Finite difference models, where Fourier damping is used to control polar instabilities, are also in this category. First the case of bit-reversed index-ordering of the sites in grid-point space will be considered, that is the sites are normally ordered in Fourier space; this is what is required for the spectral transformation discussed previously.

In spectral space only half of the transform is stored and so after the Legendre transform, the F_{M-m} need to be generated from the F_m . The communication required to do this involves just the index m and can be done in place with the site indices $M - m$ being stored on the same processor as m . If this is done the sites $M - m$ and m will now have the same index.

After the generation of the F_{M-m} from the F_m the array will be folded such that m and $M - m$ have the same site label. This mapping can be formally stated as,

$$\begin{aligned} m &\mapsto M - m & m \in \{0, \dots, M/2 - 1, M/2 + 1, \dots, M - 1\} \\ M/2 &\mapsto 0. \end{aligned} \tag{4.15}$$

For the first stage of the FFT, the communications structure required, in terms of the unfolded mapping is,

$$\begin{aligned} \{m, m + M/2\} &\mapsto m \\ \{m, m + M/2\} &\mapsto m + M/2 \quad \forall m \in \{0, \dots, M/2 - 1\}. \end{aligned} \tag{4.16}$$

In terms of the folded mapping there are 2 cases that need consideration;

1. $m = 0$
2. $m \in \{1, M/2 - 1\}$.

For the first case the communications pattern required is $\{0, 0\} \mapsto 0$ for both communications. This is a mapping that requires no communications. For the second region $m + M/2$ in the folded mapping is $M/2 - m$. Using this gives for communications pattern in Equation 4.16

$$\begin{aligned} \{m, M/2 - m\} &\mapsto m \\ \{m, M/2 - m\} &\mapsto M/2 - m \end{aligned} \quad (4.17)$$

Introduce the following definitions, where $m \in \{0, \dots, L - 1\}$,

DEFINITION 4.1

$$\begin{aligned} \text{reverse}(m, L) &\equiv m \mapsto (L - (m + 1) \bmod L) \\ \text{shiftright}(m, L) &\equiv m \mapsto (m + 1) \bmod L \\ \text{shiftright}(m, L) &\equiv m \mapsto (m - 1) \bmod L \end{aligned}$$

For $\{m, M/2 - m\} \mapsto m$ a mapping from $M/2 - m$ to m is required for each m in $\{1, \dots, M/2 - 1\}$. There are two possibilities here for this mapping; either $\text{shiftright}(\text{reverse}(M/2 - m, M/2), M/2)$ or $\text{reverse}(\text{shiftright}(M/2 - m, M/2), M/2)$. A short proof of this statement is given in Appendix C.1.2. After these communications have been done the sites in the FFT will be ordered such that sites m and $m + M/2$ are on the same processor. Figure 4-6 shows the communications pattern required for this transformation.

For the remainder of the Fourier transform there is no interaction between sites whose labels differ by more than $M/4$ so the two folded parts of the transform can be considered to be two separate transforms by the rest of the pipeline. The unpacking prior to the grid-point computations requires no communications either and neither do the grid-point computations, though in grid-point space there are interactions between different variables.

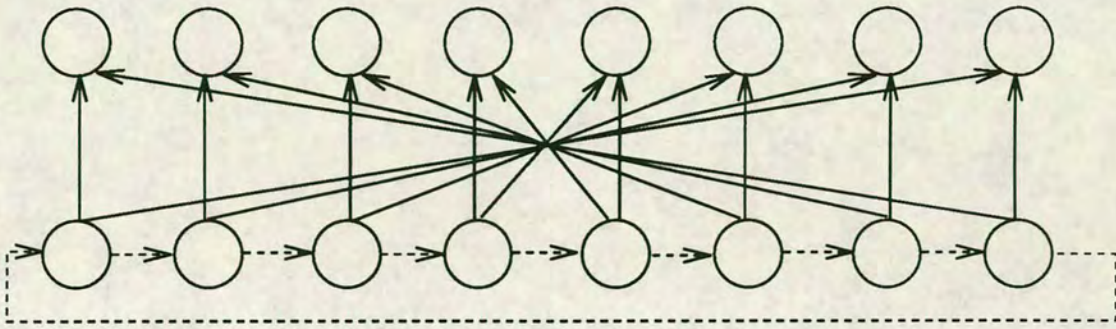


Figure 4-6: Communications pattern for computation of real fields
 Dashed lines show the shifting to the right of the data, while the non-dashed lines show the reversal required.

Considering the forward transform (from grid-point to Fourier space), the packing of the two real fields requires no communications. The Fourier transforms then proceed as before until communications are required between sites m and $m + M/2$.

The computation of the final stage of the FFT involves no communications as sites labeled m and $m+M/2$ are on the same processors. After doing this, it is necessary to unscramble the packed transforms. From Equation 1.54 the unscrambled values can be retrieved, the communications pattern required is,

$$\{m, M - m\} \mapsto m. \tag{4.18}$$

The generation of values for $M - m$ is not required due to the symmetry property of the completed transformation. The problem then is to carry out this mapping using the folded mapping, but this was already derived for the inverse transform. There are two possibilities, $shiftright(reverse(m, M/2), M/2)$ or $reverse(shiftleft(m, M/2), M/2)$.

Consider now the situation where data starts ordered in grid-point space.

It is then necessary to compute the complex values from the packed real values without doing any bit-reversal. Although not necessary for the Reading Model, the remainder of this subsection will be devoted to this topic because of its importance in other models.

The communications structure required, after bit-reversal is given by;

$$\{n, N - n\} \mapsto n. \quad (4.19)$$

Without doing bit-reversal this is

$$\{\tilde{n}^\lambda, \overline{N - n}^\lambda\} \mapsto \tilde{n}^\lambda \quad (4.20)$$

with $N = 2^\lambda$. The case where $n = 0$ does not require this communications pattern and requires no communications at all. The problem is to find an algorithm to generate $\overline{N - n}^\lambda$ in terms of \tilde{n}^λ for all remaining values of n .

The algorithm to compute $\overline{N - n}^\lambda$ from \tilde{n}^λ is to first find the most significant bit of \tilde{n}^λ and invert all lower bits (invert means $1 \mapsto 0, 0 \mapsto 1$). This has a communications pattern shown in Figure 4-7. This can be proved as follows.

Define \tilde{n}_r^λ to be $\sum_{i=0}^{r-1} n_i 2^i + 2^r$ with $n_i \in (0, 1)$ and $r \in (1, \lambda)$. This is a number whose most significant bit is $r + 1$.

$$\begin{aligned} \tilde{n}_r^\lambda &\in \{1, 2^\lambda - 1\} \\ n_r &= \tilde{n}_r^\lambda \\ \Rightarrow n_r &= \sum_{i=0}^{r-1} n_i 2^{\lambda-(r+1)} + 2^{\lambda-(r+1)} \\ \Rightarrow n_r &= 2^{\lambda-(r+1)} + \sum_{i=\lambda-r}^{\lambda-1} n_{(\lambda-(i+1))} 2^i \\ \Rightarrow N - n_r &= 2^\lambda - 2^{\lambda-(r+1)} - \sum_{i=\lambda-r}^{\lambda-1} n_{(\lambda-(i+1))} 2^i \\ \Rightarrow N - n_r &= 2^{\lambda-(r+1)} + \sum_{i=\lambda-1}^{\lambda-1} \bar{n}_{(\lambda-(i+1))} 2^i \end{aligned}$$

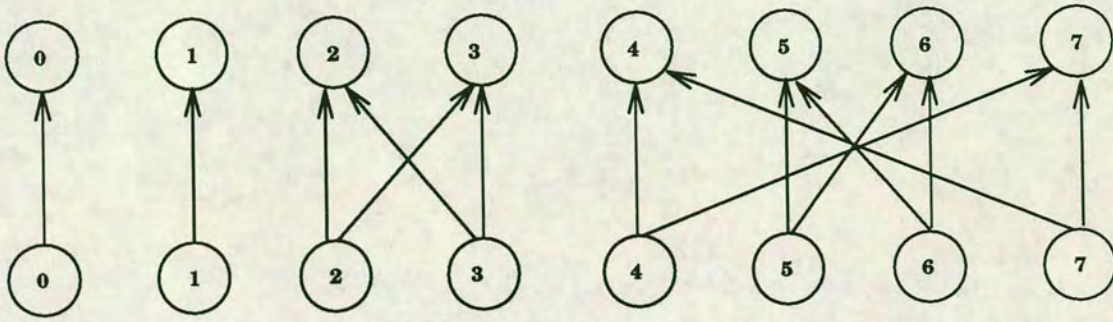


Figure 4-7: Communications pattern for computation of real fields from a transformed complex field.

The above figure shows the communications topology required to compute real fields from transformed packed fields. This figure can be interpreted in two ways: (1) it shows the communications between sites labeled 0 to 7; (2) it shows the process topology when 8 processes are used, also labeled 0 to 7, all with the same number of sites.

$$\Rightarrow \overline{N - n_r}^\lambda = \sum_{i=0}^{r-1} \overline{n_i} 2^i + 2^r$$

$P(\overline{N - n_r}^\lambda)$ (the processor label) can be formed from $P(\tilde{n}^\lambda)$ using the same algorithm that generated $\overline{N - n_r}^\lambda$ from \tilde{n}^λ . The inter-processor communications structure required for this algorithm when using 8 processors will be the same as Figure 4-7.

The parallel FFT algorithm and the mapping of the data to the processors has now been described. The remainder of the section will record what the startup time, S_{FFT} , and the timestep, T_{FFT} , are for the FFT part of the spectral

pipeline. The startup time, per real word² is

$$S_{\text{FFT}} = \underbrace{\frac{1}{2} + 1 + \tau}_{\text{Real packing}} + \underbrace{\log_2 N(4 + \tau)}_{\text{External FFT}} + \underbrace{cl + Kr}_{\text{Internal FFT}} \quad (4.21)$$

which can be rewritten as $1\frac{1}{2} + \log_2 N(4 + \tau) + cl + (K + 1)\tau$. The terms identified as "Real packing" are obtained by considering the times to compute the F_{M-m}^* , the packing of two transforms and the communications needed respectively. A similar analysis for the non-concurrent case gives $1\frac{1}{2} + N(4 + 2\tau) + cl + (K + 2)\tau$ for the startup time. The timestep for the concurrent case is $\max(\tau, 4)$ and for the non-concurrent case it is $(4\tau + \tau)$. The calculations of the timestep and the startup time for the Fourier pipe have all been on a per word basis. It is important to note that the communications cost is per word, very little can be done to alleviate this cost. One minor question that needs consideration is the amount of data in the Fourier transform relative to the Legendre transform. After the Legendre transform has completed there will be K transforms each with N words. After the generation of the complex conjugates, each transform will have $2N$ words. When two transforms are packed into one using the symmetry properties of the real transform, then the total number of transform required will be approximately halved³, making the total amount of data that requires transformation the same in the Fourier transform as is in the Legendre transform.

4.3.4 FFT Summary

This section has described a parallel algorithm for the Fast Fourier Transform (FFT). The algorithm is a pipeline which processes several, indepen-

²There are two real words in one complex number.

³If there are an odd number of transforms then $K/2 + 1$ transforms are required.

dent Fourier transforms concurrently. Each stage of the pipeline receives data from a previous stage, carries out part of the Fourier transform and then outputs data to the next stage. At the far end of the pipeline the computed transform is output.

The work, for each stage, is in addition decomposed over several processors. There are four types of process in the pipeline which does the inverse FFT (Fourier space to grid-point space).


Real Packing Each process inputs data and computes packed complex transforms from the real input values. These values are output to the next stage of the Fourier pipeline.

First External Each process inputs data from two *Real Packing* processes and computes the first stage of the complex FFT. This is done using the communications pattern described in Equation 4.17. The results from these computations are output to the next stage in the FFT pipeline, either an *external stage* or an *internal stage*.

External Each process inputs data from two processes in the previous stage of the pipeline and carries out one stage of the FFT using whichever of equations 1.49 or 1.50 is appropriate. It outputs the results of these computations to the next stage in the pipeline.

Internal Each process inputs data then carries out one or more stages of the FFT. No communications are needed to do these computations. The results of these computations are output to either another *Internal* process or to the next stage in the spectral method pipeline.

The forward transform is similar to the inverse transform, though there are slight differences due to the need to avoid having to carry out bit-reversal.

The first stage in the forward pipeline will use *internal* processes which output the results of their computations to either, other *internal* or *external* processes. The *external* processes behave in the same manner as described above. The equivalent of the *first external* process combines two half length transforms into one full length transform as described on page 121. The last stage of the pipeline is to unpack the complex transforms into real transformations using the  communications pattern described in Equation 4.18. An example of the entire inverse FFT pipeline is shown in Figure 4–8.

Having considered one component of the spectral transformation, the next section looks at the other components; the Legendre transformation and the Gaussian integration.

4.4 Legendre Transform

This section will detail how the Legendre transform is implemented, as well as the Gaussian integration (which is similar to the Legendre transform). In order to keep the pipeline balanced, both parts will require $O(N^2)$ processors, where N is the truncation number.

Schematically, the Legendre transformation can be described as,

$$v_G = \sum_{m=\text{lower}}^{\text{upper}} P_{Gm} x_m \quad (4.22)$$

G is the index for Gaussian latitude, P_{Gm} is the value of Legendre polynomial at m and Gaussian latitude G , while x_m is the value of the variable at wavenumber m . For any wavenumber, m it is necessary to compute the Legendre transformation for several different Gaussian latitudes.

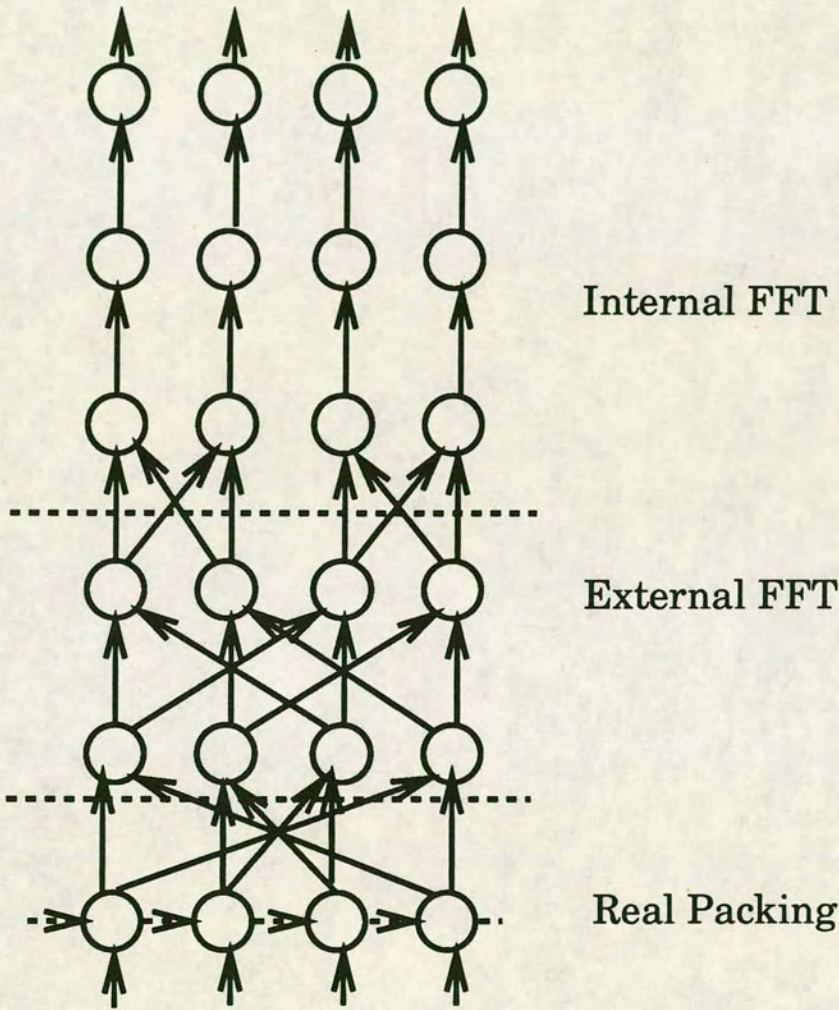


Figure 4-8: Processor topology for entire FFT

This figure shows the three components of the FFT pipeline, the packing stage, the external stage and the internal stage. This pipeline has a width of four and can process 6 FFTs concurrently.

The strategy used is based on the fact that addition is associative⁴. This sum can be decomposed into several partial sums, and the total sum can then be formed by adding the partial sums together. Section 2.5 has dealt with this concept; the reader should refer there for a broader discussion.

As discussed in Section 2.5 the natural topology to compute associative operators is to form a tree as is shown in Figure 4–9. A leaf processor should compute the value of $P_{G_m} x_m$ for all values of m that lie within its subdomain. This operation is purely local. Having done this the algorithm proceeds by each leaf processor computing the partial sums from the values that they have. Having formed the partial sums, these should be passed up to the node processors “above” them in the tree. Each node processor would receive up to $v - 1$ partial sums, v being the valency of the processor which was defined in Chapter 2. These node processors would then sum these partial sums, and pass the computed sums “up” the tree. The final sum will appear at the trunk processor. After the leaf processor has sent out its first partial sum, it then starts work computing the transformation for the next Gaussian latitude and passes the results of this to the next node above it in the tree. This pattern continues with leaf nodes computing partial sums for each Gaussian latitude and then passing the partial sums up to the next node in the tree.

Only the leaf processors and the lowest level of node processors are carrying out operations that would be done on a serial computer. These operations will be termed useful; the remainder of the processors are carrying out work that would not be needed to be done on a serial computer. Using this definition of useful the efficiency of the tree will now be computed.

⁴For the limited precision arithmetic done on a computer this is not strictly true. However a good algorithm should not be sensitive to the order of addition.

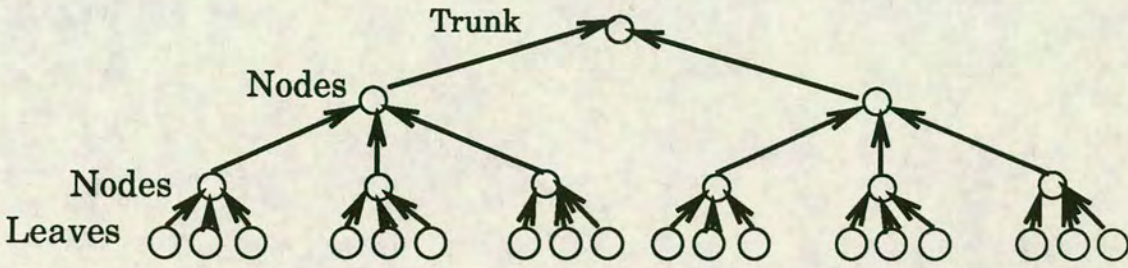


Figure 4-9: Processor topology for Legendre transform

The tree is assumed to have $d_L + 1$ completely filled levels, with a level being completely filled when the number of processors in a level, depth n is $(v - 1)^n$. Level 0 is the trunk processor. The number of processors doing useful work is the sum of the number of leaf processors and the processors in level $d_L - 1$, that is $(v - 1)^{d_L} + (v - 1)^{d_L - 1}$. The total number of processors is $\sum_{n=0}^{d_L} (v - 1)^n$. The efficiency can be shown to be,

$$e(d_L) = \frac{v(v - 2)}{(v - 1)^2 - (v - 1)^{-(d_L - 1)}} \tag{4.23}$$

This expression drops exponentially fast with the number of levels, to $(v(v - 2))/(v - 1)^2$. Assuming that the tree is always completely filled then the total number of processors P is given by:

$$P = \frac{(v - 1)^{(d_L + 1)} - 1}{v - 2}. \tag{4.24}$$

The efficiency, in terms of P , after some easy algebraic manipulation is,

$$e(P) = \frac{v(v - 2)}{(v - 1)^2} + \frac{v}{(v - 1)^2 P}. \tag{4.25}$$

This efficiency will be termed the utilization as it measures the fraction of processors doing useful work. Proofs of the above are given in Appendix C.2.

Next, having constructed the pipeline for the Legendre transform, its balance needs to be considered. Here the constraint is that the time taken by the leaf processors should be the same as that taken by the node processors. The number of complex sites on a processor to be multiplied by the polynomial and added together is n . The Legendre polynomials are real and so the total number of adds and multiplications required is $2n - 1$; for each site one multiplication and then $n - 1$ adds to compute the sum. Each node processor will do $v - 2$ operations, an add on each input.

The effect of communications on the total time is heavily dependent on the processor type. If the processor can carry out communications and calculations concurrently, then the total time for each type of process is given by,

$$\begin{aligned} \mathcal{T}_{\text{leaf}} &= \max(2n - 1, r) \\ \mathcal{T}_{\text{node}} &= \max(v - 2, r). \end{aligned} \quad (4.26)$$

For processors which cannot carry out concurrent communications then the timesteps are,

$$\begin{aligned} \mathcal{T}_{\text{leaf}} &= 2n - 1 + r \\ \mathcal{T}_{\text{node}} &= (v - 2) + vr. \end{aligned} \quad (4.27)$$

Now consider the speedup, the efficiency multiplied by the number of processors. This is

$$S = e \cdot P = \frac{(v - 2)vP}{(v - 1)^2} + \frac{v}{(v - 1)^2}. \quad (4.28)$$

The derivative of the speedup with respect to P gives,

$$\frac{\partial S}{\partial P} = \frac{(v - 2)v}{(v - 1)^2}. \quad (4.29)$$

As this expression is positive⁵ the speedup increases with the number of processors. This expression was derived using the assumption that the only factor affecting the efficiency of the tree was the relative proportions of leaf and useful node processors to the total number of processors. Other factors will limit the efficiency; the most important of these is that decreasing the number of sites on a leaf processor will at some point reduce the time taken doing the work there below that taken by the node processors, at this point no gain in wall clock time will be achieved. If there is a fixed number of points to allocate over the N processors, each processor having n points, then for the concurrent case the constraint is

$$2n - 1 \geq \max(r, v - 2) \quad (4.30)$$

and for the non-concurrent case it is

$$2n - 1 \geq v - 2 + (v - 1)r. \quad (4.31)$$

The speedup, obtained, corresponding to these values is,

$$S_{\max} = \frac{2N}{\max(r, v - 2) + 1} \quad (4.32)$$

$$S_{\max} = \frac{2N}{(r + 1)(v - 1)}. \quad (4.33)$$

When these speedups are computed, the values obtained will be fractional. The number of processors required to achieve these speedups will be the smallest integer greater than S_{\max} . Table 4-2 shows the maximum speedup that can be obtained for various different parameter values for the concurrent case, Table 4-3 shows the same thing for the non-concurrent case. The values in these tables correspond, approximately, to the mean number of meridional waves for truncations of T21, T63 and T213 on a hemisphere.

⁵To build a tree, the valency of the processors must be at least 3.

Table 4-2: Maximum speedup

N	S_{\max}		
	r=8, v=4	r=4, v=4	r=4, v=8
6	1.33	2.4	1.71
16	3.56	6.4	4.58
53	11.78	21.2	15.15

Maximum speedup for various values of v (processor valency) and τ (communications speed) for the case where the processor can do concurrent communications and computations.

Table 4-3: Maximum speedup for the non-concurrent communications

N	S_{\max}		
	r=8, v=4	r=4, v=4	r=4, v=8
5	0.44	0.3	0.34
16	1.19	2.13	0.91
53	3.93	7.07	3.03

Maximum speedup for various values of v (processor valency) and τ (communications speed) where the processor is not capable of concurrent communications and computations.

Both tables show that the maximum speedup, even for large models is, at best, $O(10)$, not the $O(100)$ to $O(1000)$ required to use a massively parallel computer. These limits on speedup were obtained by assuming that the decomposition is only being done over meridional wavenumbers. If the decomposition is also carried out over zonal wavenumbers, using more of the available parallelism of the method larger speedups could be obtained. To do this requires the distributed FFT described in the previous section.

This discussion has referred to the maximum speedup when the pipeline is running at full efficiency. That is, the effect of start up time has been ignored. These calculated speedups can, of course, only be reached asymptotically as the number of tasks tend to infinity. For the case where the processor can carry out several communications and computations concurrently, the startup time for the tree is,

$$S_{\text{tree}} = 2(n - 1) + d_L(r + (v - 2)) \quad (4.34)$$

while for the non-concurrent case it is,

$$S_{\text{tree}} = 2(n - 1) + d_L((v - 1)r + (v - 2)) \quad (4.35)$$

The sending of a message by one processor is done concurrently with the receiving of that message, thus the $(v - 1)r$ term rather than vr . Section 4.2 discussed this in some detail.

The first term in both expressions is the number of operations done by the leaf processors, while the second is sum of communications time and the operations done by the nodes as the data pass up the tree. From Equation 4.24 it can be seen that d_L is approximately $\log_{v-1} P$ for large enough P and thus the startup time increases logarithmically with the number of processors.

Finally in this section the Gaussian integration is considered. This is schematically written as;

$$V_m = \sum_{G=0}^{G_{\max}} P_{Gm} X_G \quad (4.36)$$

It is clear from this expression that the computation of any V_m is independent for different values of m . Therefore, the algorithm proceeds by decomposing the sites labeled m over the processors and computing $V_m = \sum_{G=0}^{G_{\max}} P_{Gm} X_G$ for all the values of m on that processor. As there are no constraints on how this distribution is carried out, it would be sensible to use the same distribution of wavenumbers as was used to compute the Legendre transformation. The problem then is how to efficiently replicate the X_G over the processors. The best way is to build another processor tree just like in Figure 4–9 although the description of processes on the various processors of the tree will be different from the Legendre tree.

The X_G enter at the trunk of the tree and a copy of each is sent to each processor below the trunk of the tree. Each node has this behaviour. Each of the leaf processors then increments its contribution to V_m for all the values of m which lie within its domain using the value of X_G .

Most of the earlier observations on processor efficiency and utilization are still true. Utilization is used to mean the fraction of the processors that are being used and doing useful work. The only difference is that the number of operations that the leaf processors will do has increased from $2n - 1$ to $2n$. Therefore a similar pair of equations to 4.30 and 4.31 is obtained for the constraints on n .

$$2n \geq r \quad 2n \geq v - 2 + (v - 1)r \quad (4.37)$$

The left hand result applies to the concurrent case while the right hand applies to the non-concurrent case.

This section has presented algorithms for the Legendre transform and Gaussian integration parts of the spectral transform. Both algorithms use

a tree topology. The Legendre transformation of a single model variable at a single level and for values on one meridional wavenumber is independent of any other Legendre transformation. The algorithm proceeds, for each transform, by having each leaf process multiply the values in spectral space by the appropriate values of the Legendre polynomials or derivatives thereof. Then the sum of these products is formed. Once these products have been computed for all the independent transforms on a process the values are output to the node processors in the tree. The leaf processes then repeat this work for further Gaussian latitudes.

The nodes then compute, for each transform, the sum of their input data and output the results of this sum to the node above them. At the top of the tree the trunk process outputs the computed Legendre transform to the next stage in the spectral transform.

The Gaussian integration part of the spectral transform also uses a tree process topology. The computations carried out by the node and leaf processes are different from the Legendre transformation. Consider each independent variable arriving at the trunk of the tree. The trunk process replicates this value to all the node processors below it in the tree. They in turn replicate this value to the processes below them in the tree. Eventually all the leaf processes will receive this value. Each leaf process will multiply this variable by the value of appropriately weighted Legendre polynomial and add this contribution to the total Gaussian integration.

4.5 The Full Spectral Transform

This section will show how the remaining parts of the method are done. It will also explain how all the functional units are joined together and then derive some expressions for the speedup of the algorithm (although some

simplifying assumptions will be made). Some details are implementation dependent and will be discussed in the following section.

The two remaining parts of the method, which are both model dependent, are the non-linear/grid-point computations⁶ and the time update. This second part will be considered at the end of this section after the efficiency of the pipeline has been computed.

4.5.1 The Pipeline

Considering first the coupling of the Legendre transform to the Fourier transform, to compute the transformation from spectral space to grid-point space. The processor topologies required were described in sections 4.3 and 4.4. The Legendre transform utilizes a decomposition over zonal wavenumbers, l and requires no communications between meridional wavenumbers. Therefore several Legendre transforms could be carried out in parallel, by constructing several Legendre trees. Each tree would compute Legendre transforms for the values of m mapped to that tree. Of course all leaf processors on a given tree should have the same value of m mapped to them. The outputs from the trees need to have a Fourier transform performed on them in order to compute the spectral transformation. However the Fourier transformation requires communications between sites with different values of m .

The pipeline is constructed by taking the FFT topology or butterfly topology described in Section 4.3, whose width will be determined by the normal demands for the number of grid points in a longitudinal circle (see Subsec-

⁶In a more complicated and realistic model these would include the parameterization schemes.

tion 1.4.1 for details) divided by the number of points per processor. At the base of the butterfly network each processor has attached a Legendre tree. It is not required that all the trees have the same number of processors. Connected to the top of the butterfly will be a number of processors equal to the width of the butterfly. These processors will compute the non-linear terms in grid-point space.

The computation of the non-linear terms may cause considerable imbalance in the pipeline and thus a large loss of performance. The computation per site for the non-linear unit is given by \mathcal{G} . The exact details of \mathcal{G} are dependent on the exact specification of the model and probably the processor hardware. For the Reading model, the imbalance over the other parts of the model is about two. In order to avoid this imbalance for each processor at the top of the butterfly, another tree should be built, its depth given by d_G . The number of leaf processors, $(v-1)^{d_G}$, should be equal or less than \mathcal{G}/T . For future reference this is denoted by \mathcal{G}' . There should be at least one point for each one of the leaf processors, otherwise some processors will have no work to do. As much as is possible all processors should have the same number of points in order to minimise load imbalance

Figure 4–10 illustrates this entire reverse spectral transformation (spectral space to grid-point space).

An alternative approach to building a “fan out” tree for the grid-point computations would be to pipeline the computation of the non-linear terms. However, analysis of the algebraic decomposition required here is difficult, being very model- and very configuration-specific. Thus, balancing the pipeline would also be extremely difficult and very specific. If the grid-point computations were sufficiently time consuming, in relation to other computations in the pipeline, then this approach would have to be used.

The forward transformation (grid-point to spectral) requires a fast Fourier

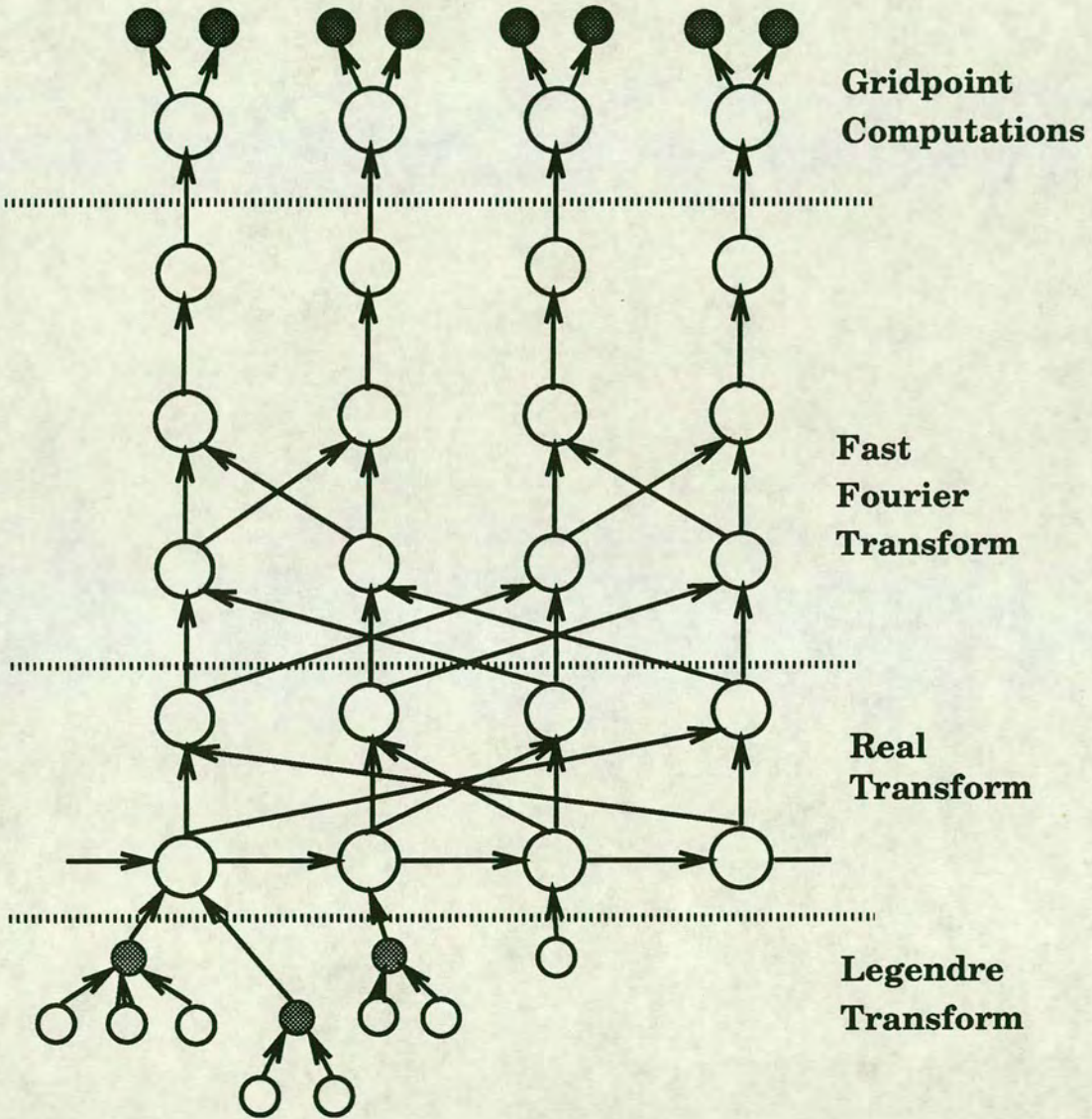


Figure 4-10: Processor topology for the inverse spectral transformation

transformation followed by a Gaussian integration. Previous sections (4.3, 4.4) have described the required topologies for these functional units. The network required is identical to that of Figure 4–10 except for two things;

1. The communications happen in the opposite direction (from top down rather than bottom up)
2. No computation of the non-linear terms is required, though if a tree is built for these terms, then another will be needed to collect the data together.

The reader should be aware that if the underlying hardware supports efficient bi-directional communications and rapid process swapping then some efficiency benefits will be gained if the total topology built is similar to Figure 4–10. The tree processes will, in this case, run both a Legendre process and a Gaussian process. The butterfly processors will run both forward and inverse FFT processes. The node processors should, of course have both the copying and summing processes running. Doing this will balance out the pipeline as different numbers of variables could be transformed in the forward and inverse parts of the pipeline. The startup time will be approximately halved. For models with a small truncation (T21, T42) this may be helpful. However, only approximately half the number of processors can be used in this case.

To complete the sub-section some calculations will be done to compute the efficiency of the pipeline. First the vector efficiency V_e of the pipeline will be computed. Table 4–4 summarises the startup times and output times for the overlapping case. Table 4–5 does the same where communications may not be done concurrently with calculations.

Table 4-4: Properties of the spectral transform components (concurrent case)

Unit	S	T
Legendre	$2n - 1 + d_L(\tau + \nu - 2)$	$\max(\tau, \nu - 2, 2n - 1)$
Gaussian	$2n + d_L(\tau + \nu - 2)$	$\max(\tau, \nu - 2, 2n)$
FFT	$1\frac{1}{2} + (4 + \tau)\log_2 N + K\tau + (q - p)c$	$\max(4, \tau)$
Non-linear Products	$G/(\nu - 1)^{d_G} + \tau d_G$	$\max(G/(\nu - 1)^{d_G}, \tau)$

This table shows the startup times and timesteps for hardware which can carry out concurrent communications and calculations.

Table 4-5: Properties of the spectral transform components (non-concurrent case)

Unit	S	T
Legendre	$2n - 1 + d_L((\nu - 1)\tau + (\nu - 2))$	$\max(2n - 1, (\nu - 2) + \nu\tau) + \nu\tau$
Gaussian	$2n + d_L((\nu - 1)\tau + (\nu - 2))$	$\max(2n, (\nu - 2) + \nu\tau) + \nu\tau$
FFT	$1\frac{1}{2} + \log_2 N(4 + 2\tau) + c_l + (k + 2)\tau$	$4\tau + 4$
Non-linear Products	$G/(\nu - 1)^{d_G} + 2\tau d_G$	$G/(\nu - 1)^{d_G} + 2\tau$

This table shows the startup times and timesteps for hardware which can not carry out concurrent communications and calculations.

The major factor which affects scaled speedup is the vector efficiency and this is a function of the startup time, S . For the concurrent case this is,

$$S = \underbrace{4n - 1 + 2d_L(r + v - 2)}_{\text{Legendre/Gaussian}} + \underbrace{2\left(1\frac{1}{2} + (4 + r)\log_2 N + Kr + (q - p)c\right)}_{\text{Fourier Transform}} + \underbrace{\frac{G}{(v - 1)^{d_G}} + rd_G}_{\text{Non-linear terms}} \quad (4.38)$$

while for the non-concurrent case it is

$$S = \underbrace{4n - 1 + 2d_L((v - 1)r + v - 2)}_{\text{Legendre/Gaussian}} + \underbrace{2\left(1\frac{1}{2} + (4 + 2r)\log_2 N + (K + 2)r + (q - p)c\right)}_{\text{Fourier Transform}} + \underbrace{\frac{G}{(v - 1)^{d_G}} + 2rd_G}_{\text{Non-linear terms}} \quad (4.39)$$

These values have been computed assuming that there are equal numbers of sites to be transformed in both directions. For the Reading model approximately 10% more data needs to be transformed from grid-point space to spectral space than from spectral to grid-point space. From Equation 4.24 d_L is $a\log_2 P_L + b$, P_L being the number of processors involved in computing the Legendre transform, while a and b are constants. The contributions from the internal part of the FFT are also constant, as is the contribution from the non-linear computations. Both of Equations 4.38 and 4.39 can then be rewritten as,

$$S = A\log_2 P_L + B. \quad (4.40)$$

The number of tasks is proportional to the truncation number, while from Table 4-1 P_L is proportional to the square of the truncation. Therefore, the number of tasks is given by $C\sqrt{P_L}$, C being another constant.

The vector efficiency is;

$$V_e = \frac{CT\sqrt{P_L}}{A\log_2 P_L + B + CT\sqrt{P_L}} \quad (4.41)$$

For $\sqrt{P_L}$ sufficiently large the number of tasks will dominate the startup time and V_e will tend to 1. In this limit the utilization will be $\frac{v(v-2)}{(v-1)^2}$ and the asymptotic efficiency will be $\frac{v(v-2)}{(v-1)^2}$. This result is not particularly surprising. Referring to Table 4-1, then the number of processors involved in computing the Legendre Transform/Gaussian integration will dominate for sufficiently large truncations and the efficiency of the entire pipeline will tend to the efficiency of Legendre/Gaussian part.

A cautionary note should be sounded, it may well turn out that “sufficiently large truncation” will be so large that the spectral method is uncompetitive against a grid point method, due to the computational complexity of the Legendre/Gaussian transforms. At this point in time it is not clear at what point the spectral method becomes computationally uncompetitive against the grid-point method. At the European Centre for Medium Range Weather Forecasting a T213 model is being used operationally.

4.5.2 Time-Step Increments

The final part of this section will consider the computation of the time-step increment. These, like the grid-point computations, are highly model dependent.

The field values for a new time-step can be computed in either the Legendre processors or the Gaussian processors. The choice of which is preferred is dependent on the size of the data in the forward or inverse transforms. If the data-set is smaller during the inverse transform than during the forward transform, then these computations should be done on the Gaussian processors, as then communications cost is minimised. The opposite is true if the data-set is larger during the inverse transformation. The data is already partitioned across the processors and so no extra data movement is required. Some models of which Bourke (1974) is an example require communications

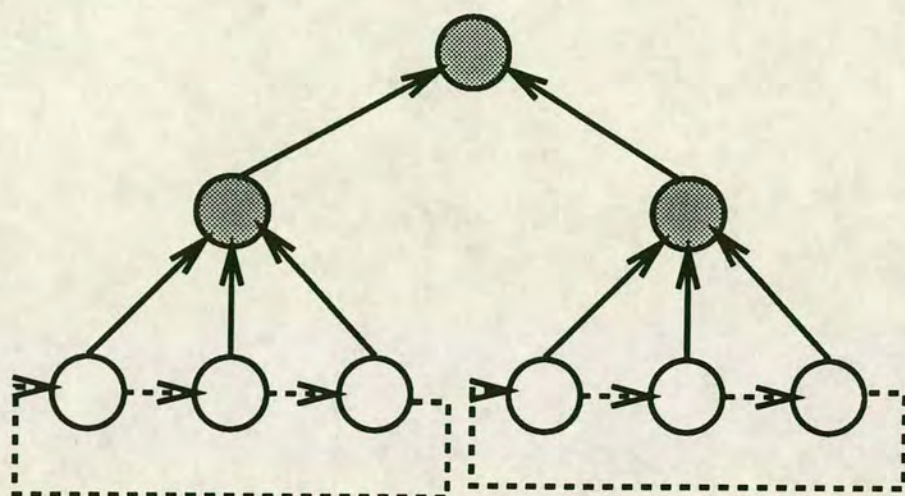


Figure 4-11: Communications within trees

Nodes are shown as grey disks, leaf processors are shown as circles. Communications needed between the leaf processors are shown as dashed lines, while the communications already discussed for the transform are shown as solid lines.

in the latitudinal direction in order to compute the east/west component of velocity. In this case the leaf processors of an individual tree will need to communicate with neighbours to the north and south. From the discussion in Section 2.5 the mapping of indices to the trees should satisfy the property that $P_{\max} + 1 = (P + 1)_{\min}$. If this mapping is used then neighbouring leaf processors will have neighbouring sites. The communications required are shown in Figure 4-11. The Reading model does not require any such connections.

Rather than computing the increment on just the Gaussian/Legendre processors, an extra set of processors could be added in order to speed up the time-step computations. This issue will now be explored. Define the time taken by the increment, per discretised point in spectral space, as \mathcal{I} . Define \mathcal{I}' to be the \mathcal{I}/T . \mathcal{I}' is the time for the increment scaled by the timestep for the pipeline. \mathcal{S}' is the scaled startup time and is defined similarly. The

efficiency of the entire algorithm (pipeline plus time-step increment) will now be computed. Assume that there are P_L processors involved in computing the Legendre or Gaussian transform and that an extra βP_L processors are added to compute the time-step increment ($\beta \in [0, \dots, \infty]$). These processors take no part in computing the Legendre or Gaussian transforms⁷. In Section 2.6 it was shown that the efficiency of an entire algorithm was given by the time weighted efficiencies of all the units. That is,

$$e = \frac{\sum_i e_i \tau_i}{\sum_i \tau_i}. \quad (4.42)$$

In this case there are two units. Subscript t refer to the time-step increment, while p refer to the pipeline. P_p is the total number of processors in the pipeline. Estimates of the efficiency and the pipelines timestep are shown below; the effects of communications times have been neglected in order to simplify the analysis.

$$\begin{aligned} e_t &= \frac{P_L(1 + \beta)}{P_p + \beta P_L} \\ \tau_t &= \frac{\mathcal{I}'}{(1 + \beta)} \\ e_p &= \frac{V_e P_p}{P_p + \beta P_L} \\ \tau_p &= N/V_e \end{aligned}$$

$$\Rightarrow e = \frac{1}{P_p + \beta P_L} \cdot \frac{\mathcal{I}' P_L + N P_p}{\mathcal{I}'/(1 + \beta) + N/V_e}. \quad (4.43)$$

⁷There would be no gain if they did, as the pipe would then be unbalanced.

It can be shown that $\frac{\partial e}{\partial \beta} \leq 0$ and therefore the efficiency will decrease, with increasing β , the maximum efficiency occurring when $\beta = 0$. This efficiency is,

$$e(0) = \frac{1}{P_p} \cdot \frac{I'P_L + NP_p}{I' + N/V_e} \quad (4.44)$$

When $N \gg I'$, $P_L \sim P_p$ then $e(0) \sim V_e$. The efficiency in this case is dominated by that of the pipeline. The speedup is given by the efficiency multiplied by the number of processors, which gives.

$$S = eP = e(P_p + \beta P_L) = \frac{I'P_L + NP_p}{I'/(1 + \beta) + N/V_e} \quad (4.45)$$

It can be shown that $\frac{\partial S}{\partial \beta} > 0$ and therefore the maximum speedup will occur at $\beta = \infty$. This is not particularly realistic and will be interpreted as $\beta \gg I'$.

$$S(0) = \frac{I'P_L + NP_p}{I' + N/V_e} \quad (4.46)$$

Define the relative speedup gain as $\Delta S(\beta)$, given by;

$$\Delta S(\beta) \equiv \frac{S(\beta) - S(0)}{S(0)} \quad (4.47)$$

This relative speedup gain measures the speedup gain by using βP_L extra processors to compute the time-step increment relative to using no extra processors. It is

$$\Delta S(\beta) = \frac{I' + N/V_e}{I'/(1 + \beta) + N/V_e} - 1. \quad (4.48)$$

The maximum gain is ΔS_{\max} and, as was already shown, occurs at $\beta = \infty$.

$$\Delta S(\infty) = \frac{\mathcal{I}' V_e}{N} = \Delta S_{\max} \quad (4.49)$$

In terms of ΔS_{\max} , $\Delta S(\beta)$ can be rewritten as

$$\Delta S(\beta) = \frac{\beta \Delta S_{\max}}{\Delta S_{\max} + (1 + \beta)} \quad (4.50)$$

Figure 4–12 shows a contour plot for a range of values of ΔS_{\max} and β . When ΔS_{\max} is large and therefore when N is small, relative to \mathcal{I}' , then quite large speedups may be obtained. This is because the computation of the time-step increment dominates the time taken by the algorithm; significantly reducing the time taken by the time-step will significantly reduce the time taken by the entire algorithm. In the case when N is large and $V_e \approx 1$, then the relative gain from increasing β is very small, as the time taken for the entire algorithm is dominated by the time taken by the pipeline. For the Reading model, even at low resolutions, the time-step increments take small amounts of time relative to the total time taken. It is therefore concluded that for the Reading model that no great gain in speedup would be obtained by allocating more processors to compute the time-step increments.

Before dealing with the implementation details and presenting some results a summary of the algorithm and its properties will be given.

The algorithm is a pipeline on the work required for the transformation from grid-point to spectral space, the processing in grid-point space and the transformation back to spectral space. The task for the pipeline is this work on one Gaussian latitude. The pipeline consists of three logical components, a Fourier transform butterfly, a Legendre transformation/Gaussian integration tree and processors to carry out the grid-point computations. The tree's efficiency is dominated by the efficiency of the nodes in the level above the leaf nodes and the leaf nodes and can be high. The Fourier transforms form

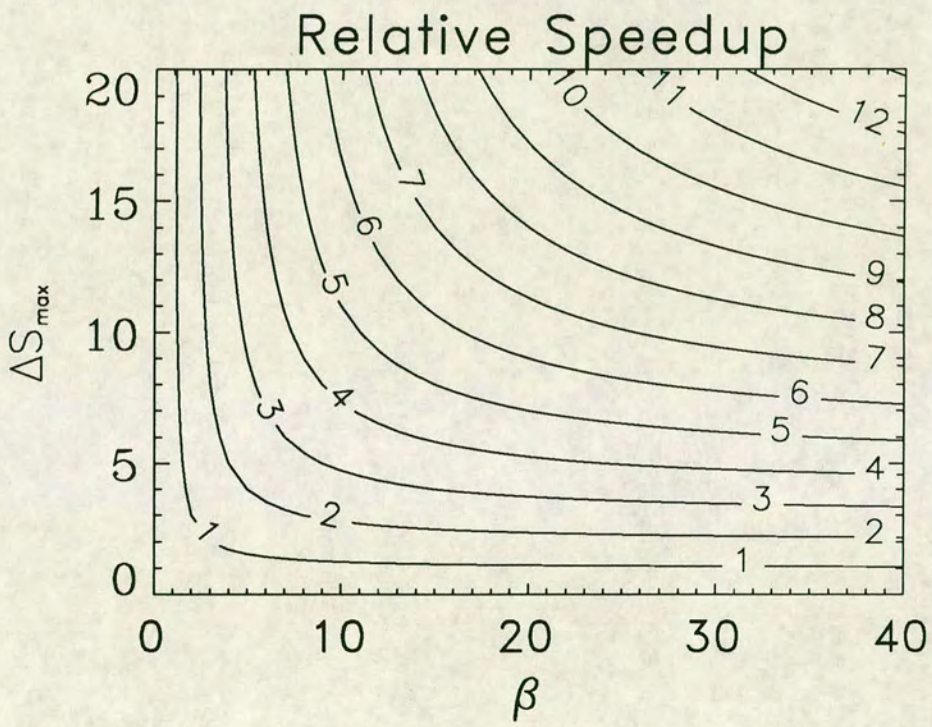


Figure 4-12: Plot of relative speedup vs β and ΔS_{\max}

the bottleneck as the work involved in carrying out a transform, relative to the communications needed, is quite small. Furthermore the communications required for the Fourier transform are a constant factor per site, unlike the Legendre transform where the communications is a constant term for each longitudinal column on a processor. If communications speeds are slow then the distributed Fourier transform may well have an unacceptably low efficiency.

The length of the pipeline is approximately proportional to the logarithm of the number of processors. For small truncations the number of Gaussian latitudes and thus tasks will be small. The pipeline length will be long compared to the number of tasks and the effect of startup time will be large.

This algorithm is unusual in that its *scaled* speedup is super-linear, i.e. $S(P + 1) > \frac{(P+1)}{P}S(P)$. There are three reasons for this:

1. The number of tasks grows linearly with the system size while the startup time grows logarithmically. Therefore V_e will increase with increasing processor number.
2. As the number of processors increases the ratio of efficient tree processors and grid-point processors to inefficient Fourier processors will increase. For small numbers of processors the ratio will be less than one.
3. The relative effect of the computation of the time-step increment on the speedup will fall as the number of processors increase.

This super-linearity is occurring because the efficiency at small processor number is low. However, if the hardware can support efficient multi-tasking then it is possible to improve this inefficiency. The partition into processes has divided the computational load equally among many processes—each

process will require the same computational effort. The efficiency for small numbers of processors could be increased by mapping several processes to the same processor, this will reduce the startup time of the pipe. In addition to this when a process is waiting for a communication the processor could run another process which has some work to do. The communications will be less efficient, although for small numbers of processors this effect will be small.

4.6 Implementation Details

This section will detail the prototype implementation of the Reading Model on the Transputer machine at Edinburgh, described in Chapter 1. Rather than rewriting the model from scratch, it was decided to convert an existing Fortran code. This was done for three reasons:

1. The process of converting this relatively simple model may provide some guidance to the effort in converting a larger and more complex model.
2. It was hoped that this conversion could be done quickly .
3. Verification of each stage was possible by comparing the partially converted model against the serial model.

The communications between processes was provided by a set of Fortran library routines supplied by Meiko, called CS-Tools, Meiko (1991). Meiko also provided a set of routines to build a loader/configurer to allow mapping of processes to processors. Separation of the two allowed development of the separate processes to occur on a workstation. Some of the remarks that follow are specific to the Meiko system and the Reading Model—however,

the author hopes that this will encourage others to convert their spectral models and provide some aid in the process. The author estimates that approximately nine months were taken in converting the Reading Model. This time includes time involved in understanding the model code.

One objective of the implementation was to have the parallel version of the model give exactly the same results as the serial version of the model. Doing this made verification easier. For all but the Legendre transform it was possible to do this. However, because the parallel algorithm uses a tree to compute the sum, the results for the Legendre transform are dependent on the number of processors used. This difference is due to rounding error and for the Reading model, the difference between the parallel and the serial version, after the Legendre transform was found to be approximately one part in 10^7 .

The general approach to convert the model was, for each one of the component subroutines, a “wrapper” module was written which would first carry out data initialisation, then repeatedly carry out the following tasks:

- Receive a task from the previous module in the pipeline
- If necessary process the input data into a format suitable for the subroutine.
- Call the subroutine.
- Again if necessary, do some processing on the data in order to put it in a form suitable for output.
- Finally, transfer the data to the next module in the pipeline.

For the grid-point computation module, there was only one modification required; the maximum length of the longitudinal sub-strip on the processor

was different from the serial model and needed to be computed by the loader program. The tree for this part of the algorithm, described earlier, was not built.

The FFT needed to be re-implemented as the parallel algorithm differed quite extensively from the serial algorithm though the results produced were the same.

The next modules that are considered are the Legendre and Gaussian transforms. In both cases the serial subroutines had a triangular data-structure. This large triangle can be completely covered by a set of smaller triangles, all these triangles are the same size as those used by a serial $T(2^n - 1)$ model. This restriction on the size allowed a radix 2 FFT to be used. Some points on some sub-triangles lay outside the original triangle. At these points the value of the Legendre polynomials are set to zero so that these points make no contribution to the transform. Figure 4-13 shows this mapping for a T21 model partitioned over 9 leaf processors. Some of the triangles are upside down and reversed, in this case some data-manipulation is required to convert the sub-triangles into the required for for the serial subroutines. After the subroutine has processed the data, more data-manipulation is required; the longitudinal rows require reversing for the upside down, reversed triangles.

Only synchronous communications were implemented, therefore concurrent communications and computations could not be carried out. Due to lack of time no conversion of the computations in spectral space were carried out. This conversion is basically a matter of data initialisation and minor restructuring in order to compute the upside down and reversed triangles. It is not expected that this would affect the times taken by this part of the computations.

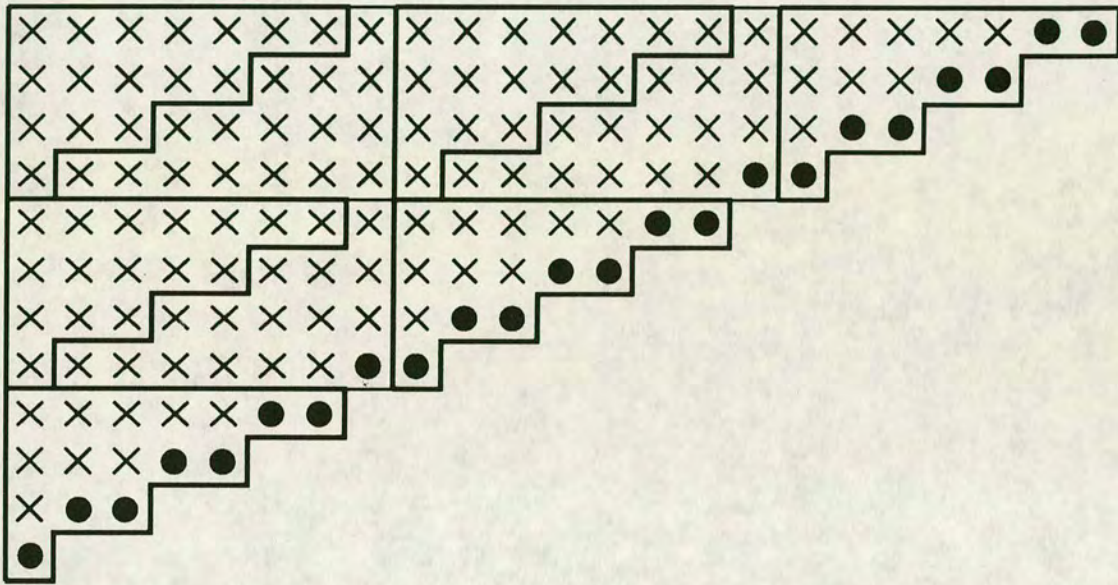


Figure 4-13: T21 model partitioned into 9 T7 modules

This figure shows a T21 spectral model decomposed over 9 processors. Crosses mark original points in the T21 model, the circles show the extra points introduced so that each processor can run the same code as the serial model. Meridional wavenumbers run from left to right while zonal wavenumbers run from top to bottom. The sub-domain assigned to a single processor is enclosed by lines.

4.7 Results

The previous section provided details of the parallel implementation of the Reading model. This section presents times and efficiencies measured from this implementation on the ECS. Efficiencies are measured by taking the times measured for the parallel implementation and dividing them by the product of the number of processors used and the times taken by the serial implementation of the model. The times taken by the serial model are shown in table 4–6. Times taken by truncations of T21, T42, T63 and T84 were measured and extrapolations made to T168 and T336 truncations. All experiments carried out using the serial and parallel implementations used a single hemisphere and five vertical levels.

Table 4–7 shows how the number of points in spectral size, the number of Gaussian latitudes and the size of the Fourier transform varies with these truncations.

For the T21 and T42 truncations benchmarks using 1, 4, and 9 leaf processors were run. The width of the FFT in these cases was 1, 2 and 4 processors with a depth of 3, 4 and 5 processors respectively. For the T63 benchmark, there was not sufficient memory available on all processors to carry out an experiment using only 1 leaf processor. For the T63 4 leaf processor case the FFT had a width of 4 and a length of 5. There were not sufficient processors available to carry out an experiment using 9 leaf processors and a FFT width of 8. The T84 truncation could only be run using 9 leaf processors due to memory limitations. From hereon a configuration will be referred to by T_nL_p where n is the truncation number and p is the number of leaf processors. For each configuration (truncation and number of leaf processors) four individual benchmarks were carried out. In two benchmarks each process was mapped to a single processor. In the other

two benchmarks two processes were mapped to a single processor, this being the doubled mapping described earlier in the chapter. In this doubled mapping equivalent Legendre and Gaussian processes were mapped to the same processor as were equivalent forward and inverse FFT processes. For each pair of benchmarks, one mapping to the processors was carried out such that neighbouring processes were mapped to neighbouring processors while for the other member of the pair no great care was taken to ensure this.

For all the benchmarks no explicit balancing of the pipeline was carried out and neither was the internal FFT split over several processors. The results of these benchmarks are shown in table 4–8. The efficiency ranges from a maximum of 0.330 to a minimum of 0.161 for the optimal mapping. Where processes were not mapped such that neighbouring processes were on neighbouring processors the efficiency of the algorithm was 10–30% lower than when the mapping was optimal. The efficiency loss when using this poor mapping was at a maximum when the efficiency of the well mapped benchmark was at its greatest and many processors were being used.

There are three factors affecting the efficiency for each configuration;

1. Load imbalance between the various components of the pipeline. In particular the size of the internal FFT will increase with increasing truncation. In addition the amount of work required to compute the Legendre transform and the Gaussian integral will depend on the number of sites on a processor.
2. In the 4 and 9 leaf processor cases the total number of sites in spectral space is greater than that required for the serial model. Recall that this was done to reuse the existing code. Table 4–9 shows the ratio of spectral sites in the parallel model to that in the serial model.

Table 4-6: Times for the serial spectral model

	Legendre Transform	Inverse FFT	Grid-point Comps	Forward FFT	Gaussian Integration	Spectral Comps	Total Time
T21	2.72	3.28	1.19	3.85	2.58	0.90	14.65
T42	20.63	15.42	4.82	18.03	19.8	3.44	81.76
T63	69.1	50.26	15.02	58.64	63.8	7.65	264.9
T84	160.6	67.0	20.2	78.1	147.4	12.7	486.9
T168	1280.	302.	81.	351.	1179.	51.	3244.
T336	10240.	1340.0	323.	1562.	9433.	203.	23101.

Times shown are for the total times and the times for the individual components in the program. Times are shown in seconds. Times for T168 and T336 are extrapolated from T84 for later use.

Table 4-7: Sizes for varying spectral truncations

Model Size	Gaussian Latitudes	Points in Spectral size	FFT size
T21	16	121	64
T42	32	462	128
T63	48	1024	256
T84	64	1806	256

Table 4–8: Times for the parallel spectral model

		1 leaf Processor			4 leaf Processors			9 leaf Processors		
Truncation	Mapping	Time	P	e	Time	P	e	Time	P	e
T21	F2	6.67	7	0.314	3.63	18	0.224	1.06	42	0.330
T21	F1	4.95	12	0.247	2.60	33	0.171	0.90	79	0.207
T21	S2	6.7	7	0.314	3.9	18	0.21	1.36	42	0.257
T21	S1	5.05	12	0.242	2.76	33	0.161	1.31	79	0.142
T42	F2	44.2	7	0.264	24.66	18	0.184	6.67	42	0.292
T42	F1	27.26	12	0.250	15.13	33	0.164	4.21	79	0.246
T42	S2	44.1	7	0.265	24.96	18	0.182	7.46	42	0.261
T42	S1	27.14	12	0.251	15.63	33	0.159	5.19	79	0.199
T63	F2				35.87	34	0.217			
T63	F1				20.92	63	0.201			
T63	S2				36.32	34	0.215			
T63	S1				21.88	63	0.192			
T84	F2							47.07	42	0.246
T84	F1							26.83	79	0.230
T84	S2							50.09	42	0.231
T84	S1							30.4	79	0.203

In the above table, timing results for a given truncation, mapping and number of leaf processors are shown. Times are accurate to 1 part in the last digit and are in seconds. P is the total number of processors used and e is the efficiency relative to the serial model. The mapping is coded as F_n if the processes were mapped to the processors such that neighbouring processes were on neighbouring processors or S_n where no great care was taken over the mapping. The number of processes mapped to a single processor is denoted by the number following the mapping code; 2 if the doubled mapping was used, 1 if not.

Table 4–9: Extra computations required by the parallel model

Truncation	FFT Width (processors)				
	2	4	8	16	32
T21	2.12	1.19	1.19	1.00	–
T42	2.22	1.25	1.25	1.05	1.0
T84	2.27	1.28	1.28	1.07	1.07

The above table shows the ratio of extra computations in spectral space required by the parallel implementation, relative to the serial implementation for different FFT widths, in processors, and with different spectral truncations.

3. The vector efficiency depends on the number of Gaussian latitudes which increases with increasing truncation number.

In order to investigate the effect of pipeline startup time and timestep a further set of benchmarks was run. In this case only the optimally placed process case was used and the number of Gaussian latitudes that were used was allowed to vary from 2 to 64. The effect of changing the the number of Gaussian latitudes is to change the number of tasks that the pipeline has to process. If the implementation is behaving as expected then the times taken for each experiment should lie on a straight line. The timestep for the pipeline is the gradient of this line, while the startup time is the point at which the line crosses the time axis. The startup time also includes a contribution from the time it takes to compute the timestep increments. The experiment was carried out for five cases; **T21L1**, **T21L9**, **T42L4**, **T42L9** and **T84L9** and the results are shown in Table 4–10. Figure 4–14 shows plots of these results with the straight line fit used to estimate the startup time and timestep for the pipeline.

Table 4–10: Effects of increasing task numbers

Model	N						<i>S</i>	<i>T</i>
	2	4	8	16	32	64		
T21 F2 L1	2.29	2.84	3.93	6.67	12.26	23.26	1.50	0.32
T21 F1 L1	2.25	2.64	3.41	4.95	8.03	14.19	1.87	0.19
T42 F2 L1	7.35	9.15	14.12	24.09	44.21	84.51	4.295	1.251
T42 F1 L1	7.48	8.78	11.41	16.65	27.26	28.41	6.128	0.660
T42 F2 L4	4.17	5.07	7.88	13.48	24.28	46.96	2.411	0.694
T42 F1 L4	4.20	4.91	6.38	9.31	15.15	26.6	3.497	0.362
T21 F2 L9	0.414	0.489	0.661	1.055	1.98	–	0.268	0.053
T21 F1 L9	0.429	0.495	0.629	0.898	1.434	–	0.361	0.033
T42 F2 L9	1.257	1.473	2.10	3.64	6.66	12.78	0.71	0.19
T42 F1 L9	1.306	1.50	1.89	2.67	4.21	7.38	1.10	0.098
T84 F2 L9	4.261	5.115	7.9	13.5	24.71	47.06	2.484	0.696
T84 F1 L9	4.40	5.116	6.582	9.493	11.315	26.827	3.689	0.362

The table above shows for different configurations the times taken, in seconds, as the number of Gaussian latitudes (*N*) is varied. Also shown is the startup time (*S*) and the timestep (*T*) estimated from these results.

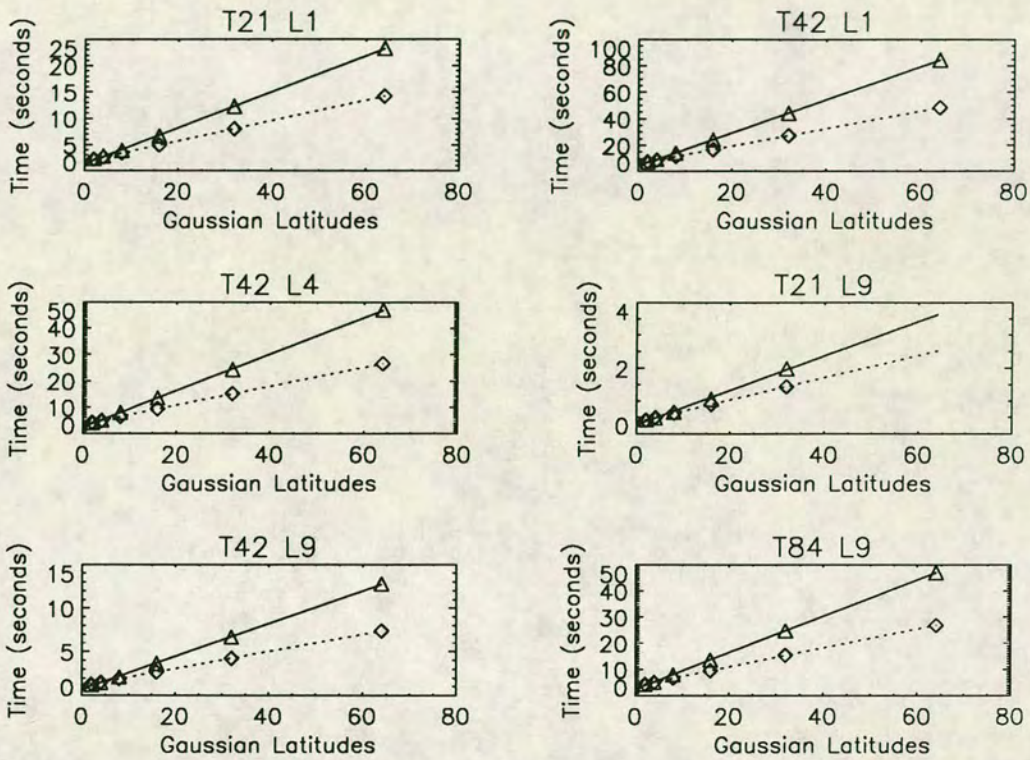


Figure 4-14: Times taken with increasing Gaussian latitudes

The times taken by the model for the doubled case are shown as triangles while for non-doubled case they are shown as diamonds. The straight line fit used to estimate the startup time and timestep for the pipelines are shown as a solid line and a dotted line for the doubled and non-doubled cases respectively.

The straight line fit of these results is highly accurate, showing that the algorithm is indeed behaving like a pipeline. There are two factors contributing to the startup time: (1) the time taken by the computations in spectral space; (2) the time taken by the first task to progress through all the processors. The first factor is the reason why the startup time does not halve when the number of processors used is halved in the doubling experiments. For all but the **T21L9** case, use of the double mapping results in the timestep approximately doubling (In fact it should be approximately 10% less than twice the timestep of the undoubled mapping due to the difference in the number of tasks in the forward and inverse transforms.). In the **T21L9** case the timestep is not doubled when using the double mapping. Here the bottleneck is the time taken to carry out the grid-point computations. Though the time taken to carry out the computations in other parts of the pipeline is reduced, the performance of the pipeline is determined by the slowest component – in this case the grid-point computations. Increasing the numbers of processors doing the grid-point computations by building a tree as described earlier in this chapter would remove this bottleneck and reduce the time taken by the computer.

It would be expected that the timestep for the **T21L1**, **T42L4** and **T84L9** cases should all be the same. The work per processor per task is constant. The pipeline timesteps for the **T42L4** and the **T84L9** cases agree to within 0.2%. However the **T21L1** case has a different, and smaller timestep. Here the bottleneck is the Legendre transform. In the **T21L1** case the one leaf processor does a T21 Legendre transform; in the other two cases a T31 transform is carried out by each leaf processor.

These results can then be used to estimate the time taken by the algorithm on larger numbers of processors and on larger truncations. Considering the scaled problem where the number of points on a processor is the same as that in the T21 truncation on 9 processors, then the timestep for the pipeline

will be the same but the startup time will increase. It is assumed that each doubling of the FFT width will cause the startup time, S , to increase by twice the timestep for the doubled case and four times the timestep for the non-doubled case. In fact this is somewhat of an over-estimate, and is an upper limit on the startup time. This assumes that doubling the width of the FFT will require an extra row of processors for the FFT and also increase the depth of the Legendre tree by one. Figure 4–15 shows a plot of the estimated efficiency, total number of processors required and the estimated speedup. If the extrapolation is correct then speedups of over 2000 are possible for the T336 case. The number of node processors in the tree is estimated at one half the number of leaf processors, by using Equation 4.25.

4.8 Conclusion and Possible Extensions

The algorithm presented in this chapter is a pipeline with several functional components. The tasks for the pipeline is the transformation from spectral space to grid-point space, computations in grid-point space, and the transformation from grid-point space back to spectral space. The length of the pipeline grows logarithmically with the number of processors in it i.e. doubling the total number of processors will cause the pipeline length to grow by a fixed amount. The efficiency of the algorithm can be increased by mapping similar forward and inverse Fourier transform processes and mapping Legendre transformation and Gaussian integration processes to processors such that there are two processes per processor for this part of the algorithm. The algorithm also has the useful property that no replication of the Legendre polynomials is required over the processors. Therefore the memory requirement of the algorithm per processor is constant. Based

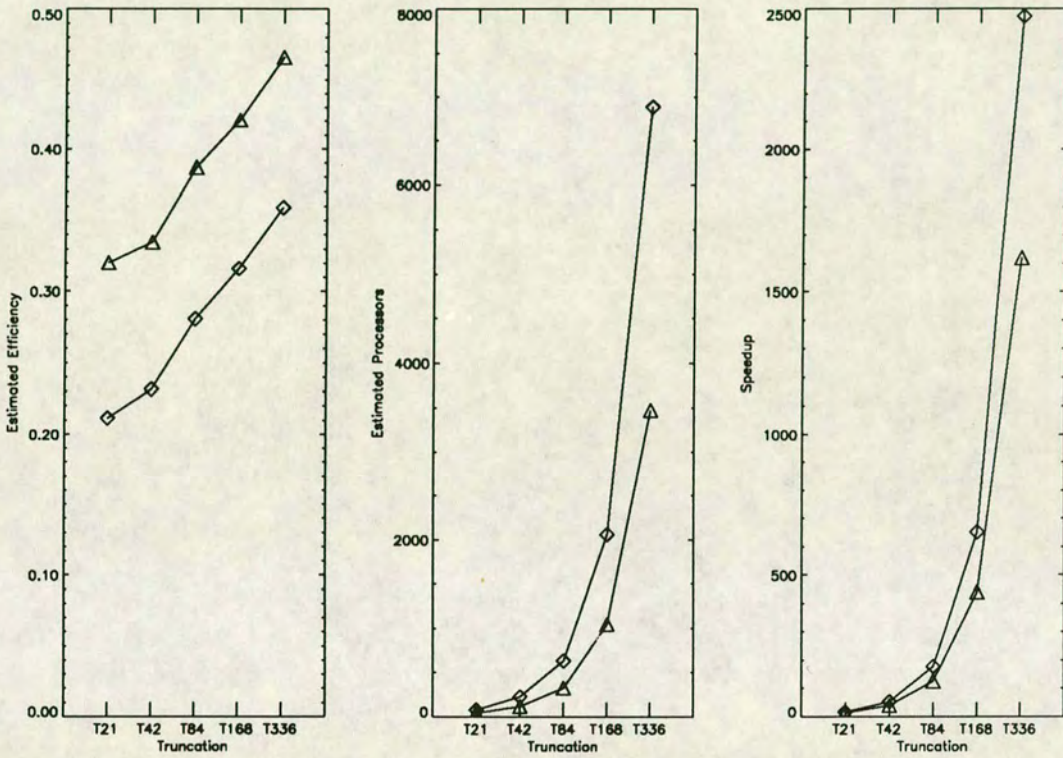


Figure 4-15: Estimate of efficiency, processors and speedup against truncation number

The left hand plot shows the estimated efficiency, the middle plot shows the estimated number of processors required and the right plot the estimated speedup. The doubled case is shown with triangles while the non-doubled case is shown as diamonds. The serial times for T168 and T336 were extrapolated from the times measured for T84.

on the authors experience with the Reading model, conversion of an existing serial spectral model seems straightforward.

The previous section presented results of a prototype implementation of this algorithm on the Edinburgh Concurrent Supercomputer (ECS). Efficiencies varying from 0.16 to 0.33 were measured for differing numbers of processors and truncations. It was concluded that the greatest contribution to this variability was from load imbalance between the various components of the pipeline. Those results also showed the importance of mapping the process topology onto the processor topology optimally with the loss in efficiency being up to 30% of the best efficiency due to a poor mapping.

The prototype implementation used non-concurrent communications and computations (synchronous communications), an implementation using asynchronous communications would be expected to have greater efficiencies. The performance of the algorithm was estimated for large truncations (T168 and T336) and very large numbers of processors. The efficiencies for synchronous communications are expected to reach 0.40 to 0.45 and speedups of approximately 600 for the T168 and over 2000 for the T336 cases were estimated.

The transform part of the algorithm, that is the Legendre Transform, Fourier transforms and Gaussian integrations require communications horizontally but require no vertical communications. Therefore several spectral transformations could be performed independently by different pipelines. The grid-point computations require vertical communications. In the Reading model, which uses an implicit scheme, the *spectral-space* computations require a matrix multiplication which can easily be decomposed over several processors. However, the Legendre polynomials would then be replicated over several processors. There is more parallelism available in the spectral transformation; the transform and grid-point computations carried out on each Gaussian latitude is independent. Therefore if there are sufficient

Gaussian latitudes that the vector efficiency of the pipeline is high enough more pipelines could be built doing several transformations concurrently. These would all be gathered together in the Gaussian integration stage. Some research however would be required to put these into practice.

However there are several points that require consideration before implementation should be carried out on several hundreds or thousands of processors.

Perhaps the most important is that an efficient mapping of the algorithm's topology, to the processor topology must be possible or alternatively that the computer has sufficient communications resources that the exact details of process placement are unimportant. The algorithm is a very fine grain algorithm and many communications are required, therefore the communications speed of the machine must be high. Furthermore the actual size of the tasks on each processor needs tuning to achieve optimal performance. As described the algorithm is purely for the dynamics part of a spectral method. For operational use it would need to be extended to include parameterization schemes.

The problem with the parameterization schemes is that they threaten to form the bottleneck in the pipeline unless preventative measures are taken. Speculating now and using some of the ideas discussed in chapter 3 on task movement: the computation of all the parameterization schemes could be distributed over the processors of the pipeline, in addition to the spectral transformation. A task-mover would then move tasks in order to use as many processor as possible to compute these schemes in order that this part of the algorithm is not a bottleneck.

If all these problems could be solved, and if a single processor delivering a sustained speed of 100 Mflops, for the serial model, and with a sustained communications speed of 100 Mbytes/sec could be made then the author

believes that a T168L40 version of the Reading model could deliver a computations speed of 400 Giga-flops using 10000 processors. There are 40 levels which are assumed to be divided up over 10 transformation pipelines, each with 1000 processors. The doubled mapping is assumed to be used in each pipeline. If the undoubled mapping was used with the same number of processors, then 5 transformation pipelines would be used, each with 2000 processors. In this case a computations speed of only 300 Giga-flops could be achieved. If only 1000 processors were available then a single transformation pipeline should be built which could provide a computations rate of 40Giga-flops when using the doubled mapping.

Having examined the spectral method the next chapter looks at grid-point methods.

Chapter 5

Grid-Point Methods

5.1 Introduction

This chapter examines explicit grid-point models. The problems involved in implementing an explicit grid point model on a massively parallel computer, such as the ECS are examined. In addition results of the implementation of a three-dimensional primitive equation are presented. This chapter only considers the dynamics part of a model, α_s parameterization schemes were discussed in Chapter 3.

In order to highlight the problems in using geometrical decomposition, a limited area model is considered first. The idea of interaction range is introduced. This enables quantitative analysis to be made of the communication requirements of different schemes. Fox *et al.* (1988) consider the size of stencil for various difference schemes but they do not use it to compute efficiencies. Using this quantitative approach various minimum constraints on the domain size per processor can be developed and predictions of the efficiency can be made.

Next, the problems involved in implementing a global grid point model will be discussed. This involves the fast Fourier transform (FFT), and also involves a load-balancing problem.

In order that realistic timings for an atmospheric model can be obtained a three-dimensional model was implemented on the ECS using the language Occam. This model uses an Arakawa 'B' grid on a regular latitude-longitude grid and it uses the horizontal differencing scheme described in Bell and Dickinson (1987). The vertical coordinates use the hybrid scheme described by Simmons and Burridge (1981) as well as the changes needed to conserve energy and angular-momentum. Appendix B gives a description of the difference equations used.

Most of the comments made will hold for all finite difference schemes. Many meteorological grid-point schemes use staggered grids and as has been pointed out by Cats *et al.* (1990) this reduces the amount of data that need to be communicated by a factor of two. However, for an architecture which can carry out several communications concurrently, this should not reduce the communications time significantly.

5.2 Limited Area Models

This section considers the problems involved in implementing a limited^{area} grid-point model on a MIMD architecture. Only a grid topology is required and the richer topology provided by, for example, a hypercube topology is not necessary. An analysis for a two dimensional decomposition is first carried out, followed by the results of the implementation using this decomposition. After this, some theoretical analysis of a vertical decomposition is presented.

5.2.1 Geometrical Decomposition

In geometrical decomposition each processor has the same code but different data. For efficient finite difference schemes, data should be divided up such that adjacent processors, have neighbouring data points as in Figure 5-1.

It is useful to introduce the concept of *temporal decomposition*, in which first some variables are communicated, then all the calculations that are possible upon these variables are carried out. Algorithms naturally decompose into *blocks* where each block consists of communications of some variables followed by some calculations using these variables. In general, the blocks are ordered in time. For example, the finite difference form of the surface pressure evolution equation is, with NLEV vertical levels,

$$p_*^{n+1} = p_*^n - \frac{\delta t}{a \cos \theta} \sum_{m=1}^{\text{NLEV}} \delta_\lambda (u_n \overline{\Delta p_n^{-\lambda \theta}})^\theta + (\delta_\theta \cos \theta v_n \overline{\Delta p_n^{-\lambda \theta}})^\lambda \quad (5.1)$$

where

$$\overline{f(i, j)}^\lambda = \frac{f(i + \frac{1}{2}, j) + f(i - \frac{1}{2}, j)}{2}$$

and

$$\delta_\lambda f(i, j) = \frac{f(i + \frac{1}{2}, j) - f(i - \frac{1}{2}, j)}{\delta \lambda}.$$

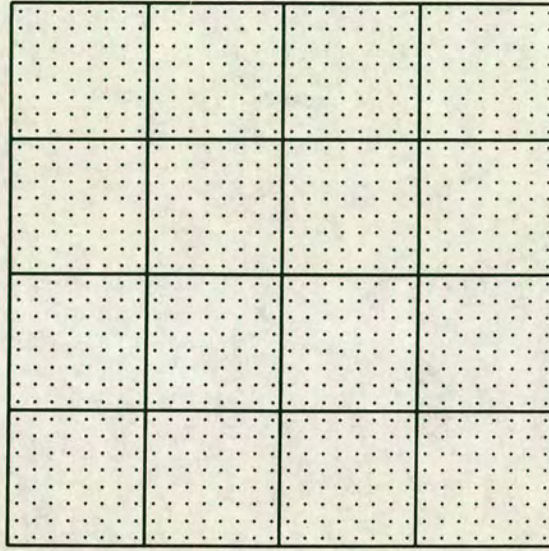


Figure 5-1: Processors with grid over-laid.

Processors are shown as thick boxes and grid points by dots. Each of the 16 processors has 8×8 grid points.

See Appendix B for a description of the variables in the above equations. To compute the value of $\overline{\Delta p}^{\lambda_0}$, first edge values of Δp will have to be exchanged with neighbouring processors, then the value of $t(i + \frac{1}{2}, j) = \Delta p(i + 1, j) + \Delta p(i, j)$, for all values of $t(i + \frac{1}{2}, j)$ on the processor, can be computed. Having computed this then these values of t can in turn be exchanged allowing the computation of $\overline{\Delta p}^{\lambda_0}$. This whole computation therefore consists of two blocks.

Another useful concept is *interaction range*. This characterises the neighbourhood size of the algorithm being a measure of how far data must be moved in grid-point space. Consider a set of variables defined on a grid of points. For example in the model being considered, some of these variables would be p , u , and v (where the usual meaning is assumed). Define a vector $\mathbf{X}^0(i, j)$ at each grid point where $X_0^0 = p(i, j)$, $X_1^0 = u(i + \frac{1}{2}, j + \frac{1}{2})$, $X_2^0 = v(i + \frac{1}{2}, j + \frac{1}{2})$, and so on for any other variables. Note that this is for a

particular way of storing the variables, which use a staggered grid, on the processor. It would be possible to define the vector \mathbf{X} in many ways. Then, given some new vector $\mathbf{X}^1(i, j) = f(\{\mathbf{X}^0(n, m)\})$ which is a function of a subset of the old values, labeled by $m, n \in$ a subset of all the grid points, define I_R (the interaction range) as the smallest integer such that

$$\forall n, m \quad |n - i| \leq I_R(i, j), |m - j| \leq I_R(i, j). \quad (5.2)$$

Note that the subset of all grid points $\{(n, m)\}$ for any particular (i, j) define the neighbourhood of the point (i, j) for the algorithm and I_R can take on a different values at different grid points and at different times. For an algorithm with a regular communications structure, which Fox *et al.* would call "crystalline", the interaction range is best defined for a block of the algorithm and so different parts of the algorithm may have different interaction ranges.

Consider the algorithmic block $f(\phi) = \delta_{\lambda p}$. From this definition of interaction range and the way that variables are stored on the grid then for this case the interaction range is one for all grid points. The concept of interaction range is particularly useful for finite difference grid point methods. For semi-Lagrangian methods the interaction range may well be different at each grid point and vary with time and thus may not be such a useful concept. For an example of a grid-point semi-Lagrangian scheme see Robert (1982).

Using the interaction range it is possible to split the subgrid on each processor into two different regions: an inner region where data to carry out the calculations is already on the processor and an outer region where some data needs to be exchanged with neighbouring processors. The size of the outer region depends only on the interaction range. Figure 5-2 shows this for an idealized communications pattern. Figure 5-3 shows what would be required for part of a fourth order finite difference scheme.

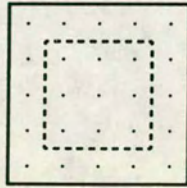


Figure 5-2: Idealized separation of sub-domain into inner and outer regions

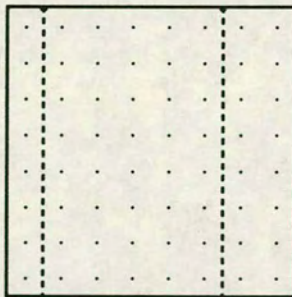


Figure 5-3: A more typical example of how the grid-points split up into two regions

On some architectures it possible to do communications concurrently with computation. Exploiting this feature is useful in obtaining the maximum performance from such an architecture. If each processor sends data from its outer region to neighbouring processors while simultaneously calculating in the inner region, a substantial overlap of calculations and communications is possible. When both of these processes have been completed, every processor calculates in the outer region. There are two effects of this; first communications are at a very low level in the code and second a minimum size for the subgrid on each processor is imposed, if the maximum speed from each processor is desired. The corresponding segment of code will look like this:

```
BEGIN SEQUENTIAL
  BEGIN PARALLEL
    ... Do communications for outer region
    ... Do calculations for inner region
  END PARALLEL
  ... Do calculations for outer region
END SEQUENTIAL
```

The original sequential code would be similar to:

```
DO i=1,limit
  B(i)=A(i)+A(i+1)
ENDDO
```

The calculations for the two regions on a processor would be very much like the original sequential code, though the loop limits would be different. As can be seen there is quite an increase in code complexity. In order to minimise the ratio of communications time to computations time a square

subdomain should be chosen. In the analysis that follows it is assumed that each processor has a subdomain size of n^2 .

On a staggered grid, the region which does not need data from neighbouring processors has $n^2 - (2I_R - 1)n$ sites. The time taken to carry out the communications is given by $r\tau I_R n$ and the time to do the computations in the interior region is $(n - (2I_R - 1))n\tau c$. With c being the number of operations per grid-point that need to be done, τ the time it takes to do 1 floating point¹ instruction and r being the number of instructions that can be done in the time it takes one word to be transferred between neighbouring processors. The total time taken by the processor in this case is the sum of the time spent computing in the interior region and the time spent in the outer region. This time is given by;

$$\tau_{\text{tot}} = [(2I_R - 1)nc + \max(n^2c - (2I_R - 1)nc, I_R r n)]\tau. \quad (5.3)$$

If the maximum efficiency is required, then a total overlap of the edge communications with the interior computations is required. This gives the following constraint on the size of the sub-domain.

$$rI_R \leq c(n - (2I_R - 1)) \quad (5.4)$$

which has a solution:

$$n \geq I_R \left(\frac{r}{c} + 2 \right) - 1. \quad (5.5)$$

If the simplest operation is considered, to give the maximum array size, then c is 1. For the T800, the bandwidth is 1×10^6 bytes per second per link, and

¹In this model of computation one floating point operation is assumed to take the same time for all operations.

the calculation speed is about 0.5×10^6 floating point operations per second for 32 bit arithmetic giving an τ value of 2 which leads to the condition

$$n \geq 4I_R - 1. \quad (5.6)$$

For $I_R = 1$ and $I_R = 2$ this gives $n \geq 3$ and 7 respectively. Considering the FPS-T series where $\tau \approx 12$ (Snelling and Tanqueray, 1988), the corresponding constraint is that $n \geq 13$ and $n \geq 27$ for $I_R = 1, 2$ respectively. These minima of n depend on the difference scheme used and in particular on the least computationally intensive block.

If the array size is less than optimal then the efficiency is

$$e = \frac{\tau_{\text{calc}}}{(\tau_{\text{comms}} + \tau_{\text{calc.edge}})} \quad (5.7)$$

In this case the time to communicate edge values is greater than the time to calculate interior values. From Equation 5.3 this reduces to,

$$e = \frac{cn}{I_R(\tau + c)} \quad (5.8)$$

Having computed the efficiency for a given block, the next problem is to do this for an entire program. For this analysis the temporal ordering of the individual blocks is irrelevant. The essential characteristics of a block are c , the computational complexity and I_R , the interaction range. A relative weight for each *type* of block is defined; this just measures its relative importance in the entire program. Blocks are divided into types depending on their computational complexity and interaction range.

For a given *type* the relative weight is defined as the total time neglecting communications spent in all blocks of this type normalised by the total time, again neglecting communications, taken by the entire program. This is equivalent to the following definition, with $N(c, I_R)$ being the number of blocks of a type.

DEFINITION 5.1

$$k(c, I_R) = \frac{N(c, I_R)c}{\sum_{\forall c, I_R} (N(c, I_R)c)}$$

The sum in the denominator of the above expression is over all values of c and I_R that occur in the program. From this definition, it is clear that the sum of all k is 1. When the time taken by a single block is $\tau(c, I_R)$, the total efficiency is given by;

$$e = \frac{\sum_{\forall c, \forall I_R} N(c, I_R)n^2c\tau}{\sum_{\forall c, \forall I_R} N(c, I_R)\tau(c, I_R)} \tag{5.9}$$

which can be rewritten as,

$$e = \frac{1}{\sum_{\forall c, I_R} k(c, I_R)\tau(c, I_R)/(\tau n^2c)} \tag{5.10}$$

Communications are only necessary when values of $I_R > 0$. For blocks where communications are required, then using Equation 5.3, the time, $\tau(c, I_R)$, spent in single block is $[(2I_R - 1)nc + \max(n^2c - (2I_R - 1)nc, I_R\tau n)]\tau$. When there are no communications between neighbouring processors, $\tau(c, 0)$ is always $n^2c\tau$. If k_0 is defined to be $\sum_{\forall c} k(c, 0)$ then expression 5.10 can be restated as,

$$e = \frac{1}{k_0 + \sum_{\forall c, I_R > 0} k(c, I_R)\tau(c, I_R)/(\tau n^2c)} \tag{5.11}$$

For the model being considered, analysis of the equations in Appendix B leads to the values of $N(c, I_R)$ shown in Table 5-1. The values of k this leads to are shown in Table 5-2.

The efficiency after substituting for τ is then given by the following expression;

$$e = \frac{1}{k_0 + \sum_{c=1, I_R=1}^{2,2} k_{c, I_R} \{ (2I_R - 1)/n + \max(1 - (2I_R - 1)/n, I_R\tau/cn) \}} \tag{5.12}$$

Figure 5-4 shows a plot of efficiency against n using the computed values of k for different values of τ for both the concurrent and non-concurrent case.

Table 5-1: Values of $N(c, I_R)$ for the grid-point model

		Values of $N(c, I_R)$		
		Advection stage		Adjustment stage
		I_R		
c	1	1	2	1
	2	30	7	9
	2	22	17	23
No communications		95		94

Values are for a single column of the atmosphere with five vertical levels and no vertical communications.

Table 5-2: Values of $k(c, I_R)$ for the grid-point model

		I_R	
		1	2
c	1	0.0868	0.0107
	2	0.277	0.0517
k_0		0.574	

These values were derived from Table 5-1.

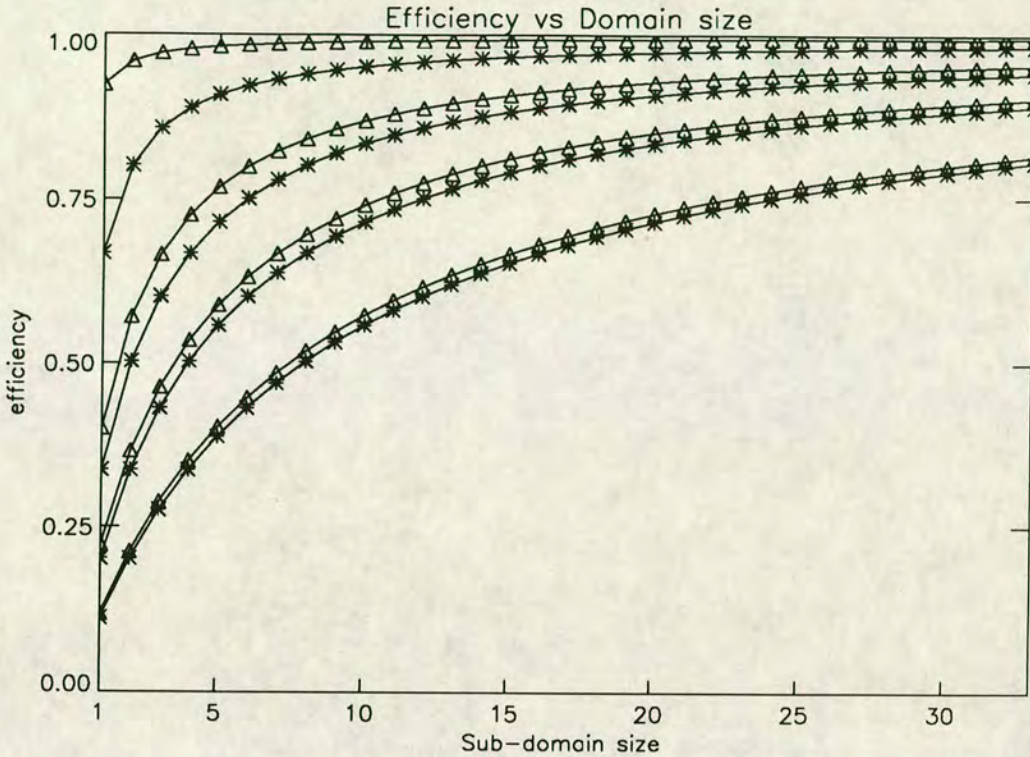


Figure 5-4: Calculated efficiencies for different communications.

Lines with values plotted as triangles are for the concurrent case while lines with stars are for the non-concurrent case. The values of r are from the top to the bottom 2.0, 8.0, 16.0 and 32.0 respectively.

Note that the benefit from concurrent communications is greatest where the sub-domains on individual processors are small.

Figure 5-5 show the times measured for both the concurrent and non-concurrent communications case using the implemented model described in Appendix B. In addition to these results the times taken relative to a domain size of 32×32 for the concurrent case are also shown, this being an approximate measure of efficiency. Note how the efficiency of the adjustment step with several $I_R = 2$ communications is below that for the advection. The times were obtained using 64 processors arranged in an 8×8 grid and

Table 5-3: Timing results for grid-point model with a fixed number of processors

	Domain Size					
	4×4	8×8	12×12	16×16	24×24	32×32
Concurrent	347	856	1598	2538	5102	8600
Non-concurrent	377	922	1652	2717	5259	9112

Times are the total time taken in milliseconds for horizontal domains shown on each processor and with 5 vertical levels. These results are on a 8×8 processor grid with 5 levels and using cyclic boundary conditions.

are tabulated in Table 5-3. The efficiencies measured are different from the calculated values particularly for small values of N . The difference is probably due to the costs of starting communications.

The effects on the time taken, by varying the total number of processors while keeping the sub-domain size fixed, are tabulated in Table 5-4 and plotted in Figure 5-6. In addition to the total times measured, the times measured for the adjustment and advection steps are also shown. As expected, there is no variation in the time taken when using concurrent communications as the number of processors increase. When using sequential communications there is small increase in the time taken as the number of processors in the domain decreases. From the small amount of data available the increase seems to be logarithmic with increasing processor number. This increase is most likely a function of the ECS hardware. If this increase is indeed logarithmic then when using approximately 8000 processors the time taken would be increased by 10%.

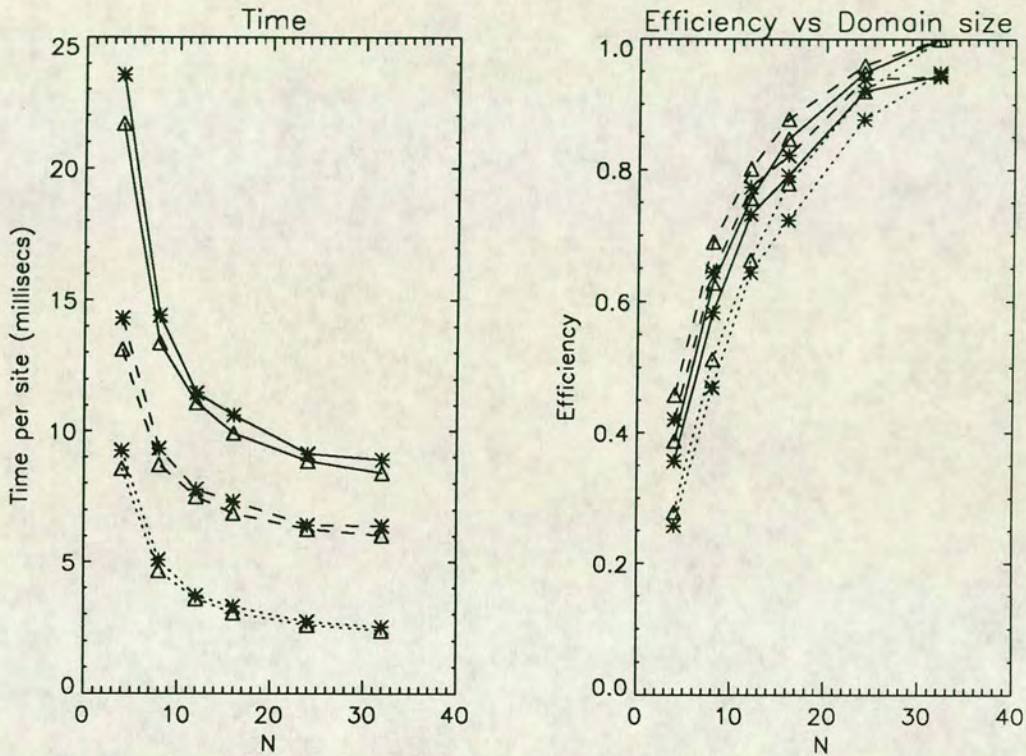


Figure 5-5: Times and efficiencies for the grid-point model with changing sub-domain size

The left hand graph shows the times taken while the right hand graph shows an estimate of efficiency. Times and efficiencies for the concurrent and non-concurrent are show as triangles and stars respectively. In both graphes the values for the advection part of the model are shown with a dotted line, the adjustment stage with a dashed line and the whole timestep with a solid line.

Table 5-4: Timing results for grid-point model

	Number of processors				
	2 × 2	4 × 2	4 × 4	8 × 4	8 × 8
Concurrent	2538	2538	2538	2538	2538
Non-concurrent	2689	2683	2692	2705	2718

Times are the total time taken in milliseconds with a fixed sub-domain size per processor and the total number of processors changing. Each sub-domain on a processor has 5 vertical levels and a horizontal size of 16×16 .

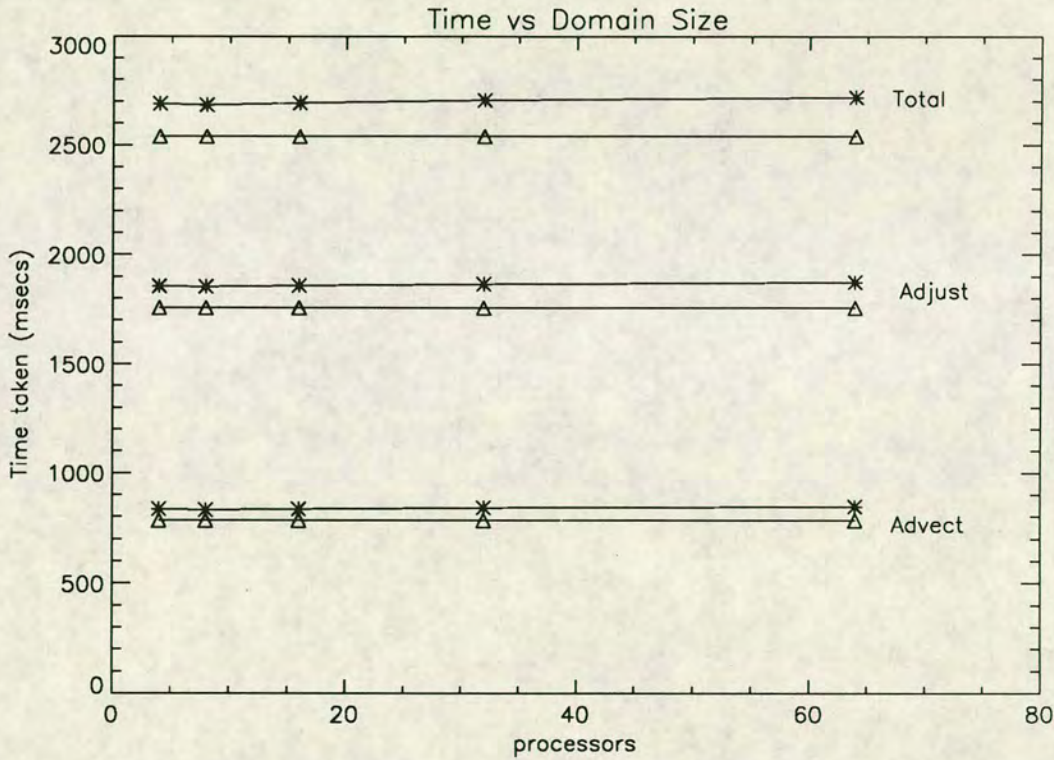


Figure 5-6: Times taken with varying processor number

This plot shows the effects of varying the processor number on the times taken for one timestep on fixed sub-domain size of 16×16 . Times taken by concurrent communications are shown as triangles while times taken by non-concurrent communications are shown by stars.

5.2.2 Vertical Decompositions

If a fixed size of problem is considered, then, as the number of processors increases, the subdomain size will decrease. In the previous subsection it was shown that the efficiency falls with decreasing sub-domain size.

In order to increase the number of processors that can be used on a given size of problem, without decreasing efficiency, a vertical decomposition over processors could be used in addition to horizontal decomposition described earlier. In the dynamics schemes considered in this chapter and the previous chapter, on the spectral method, there were many communications horizontally but only a limited number vertically—there is a strong horizontal coupling but a weak vertical coupling. However for the parameterization schemes the opposite is true, they have no horizontal communications at all but they do have very strong vertical communications. As was stated in Chapter 3 it is necessary that single columns of the atmosphere be on the same processor for the parameterization scheme. Therefore if a vertical decomposition is used, there will need to be a movement of data such that the data for any single atmospheric column is on a single processor. Figure 5-7 illustrates this.

The advection part of the timestep has no vertical coupling at all, but there is some vertical coupling in the adjustment steps. For the adjustment step with the exception of Equations B.9, B.11, B.13 and B.28 any vertical communications that occur are local with an I_R of 1. These four equations require the computation of terms which are similar $F_k = \sum_{j=1}^k G_j$ and are normally calculated iteratively from $F_k = F_{k-1} + G_{k-1}$.

The rest of the vertical interactions are similar to the horizontal interactions discussed in Subsection 5.2.1. There is one difference; the grid is not staggered in the vertical which changes the size of the interior region to $(u - 2I_R)n^2$, where u is the vertical extent of the sub-domain, n^2 is the

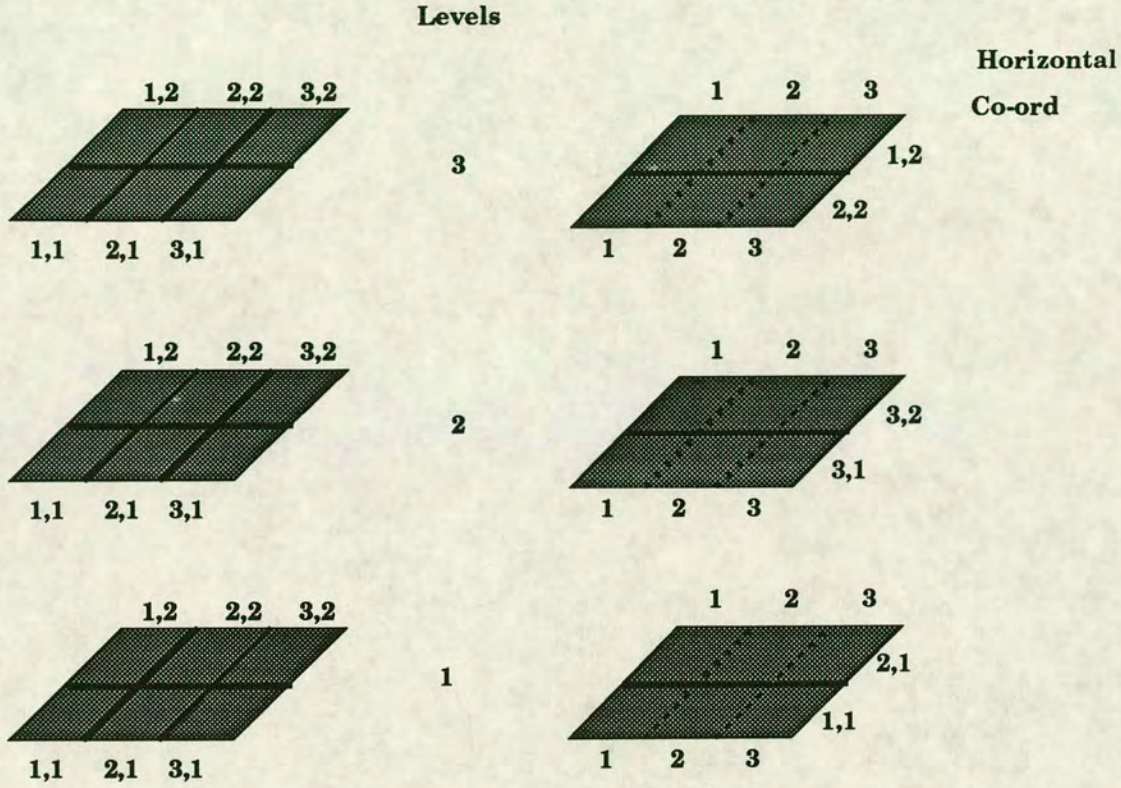


Figure 5-7: Movement of data so that every atmospheric column is on a single processor

The shaded squares each represent a processor, calculating 6 grid points with 3 levels of vertical decomposition, thus each column labeled i,j has data on three different processors. The data needs to be moved, as in the right hand diagram, so that each atmospheric column has its data on the same processor.

horizontal size of the sub-domain. The equivalent of Equation 5.3 for this case is,

$$\tau^V(c, I_R) = [2I_R c + \max((u - 2I_R)c, I_R r)]n^2. \quad (5.13)$$

Recognizing that each processor will have some number, K , of partial atmospheric columns within its sub-domain then the iterative method could be pipelined. See Section 4.2 for details of pipelines. How this could be done will now be described. The total number of points in the entire column is V and it is assumed that each column is divided up amongst P processors giving each processor a sub-column with $V/P = u$ points. The algorithm that will now be described is a specific case of the general algorithm shown in Section 2.5

First introduce the following definition,

DEFINITION 5.2

$$F_i^j \equiv \sum_{k=i}^j F_k$$

then F_1^j is the same as the F_j defined earlier. It is also useful to add the labels $\min(p)$ and $\max(p)$ for the minimum and maximum values of the vertical label on the processor p . To actually compute this, the procedure is, for all processors except the first² one: wait until the value $F_1^{\max(p-1)}$ arrives from processor $p - 1$; use this value to compute F_1^j for all the values of j on the processor. ($j \in \{\min(p), \dots, \max(p)\}$); now pass $F_1^{\max(p)}$ onto the next processor. This is repeated until the processor has done the computation for all the vertical columns in its horizontal domain. For the bottom processor

²First here means the processor with a sub-domain at the bottom of the atmospheric column.

there is only one slight modification required; it does not need to receive a value from any other processor, and can immediately compute F_1^j .

For the pipeline, the startup time is $[V + (P - 1)r]\tau$, this comes from the total number of additions plus the total time spent communicating; each communication is a transmission of one word to the next processor. After the first column has been computed, successive columns will have been computed a time later given by the maximum of the time to do the summation on the partial column or the time to transfer one word between two neighbouring processors. The efficiency of the entire pipeline is,

$$e = \frac{Ku}{V + (P - 1)r + (K - 1)\max(u, r)}. \quad (5.14)$$

If the vertical granularity, u , is kept fixed as the number of processors, V , in the vertical domain increases then the startup time of the pipeline will become important. It is assumed that u is r , in order to give the optimum efficiency³. The point at which the pipeline is 50% efficient is,

$$Kr = 2(V + (P - 1)r + (K - 1)r) \quad (5.15)$$

which, after some rearrangement and using $P = V/r$, shows that the number of processors at which half efficiency occurs is,

$$P = \frac{1}{2}(K + 1). \quad (5.16)$$

Inefficiencies in this stage of much lower than a half could be accepted as the computation of the iterative scheme accounts for approximately one sixtieth of the time spent in the dynamics part of the model. For some models, in particular models with a large number of vertical levels, in order to

³This assumption is not unreasonable as the time to initiate communications will make the communications time larger than might be expected, see Figure 1-2.

utilize the available parallelism it may be necessary to have more processors in the vertical. An example of a stratosphere-mesosphere model with 32 vertical levels on $5^\circ \times 5^\circ$ grid is described in Fisher (1987). For this case some benefit might be gained by reducing the startup time. This can be done by modifying slightly the algorithm discussed above.

1. Each processor computes the partial sum $F_{\min(p)}^{\max(p)}$. As a side effect of this all processors will have $F_{\min(p)}^i \forall i \in [\min(p), \max(p)]$. All processors can proceed in parallel for this stage.
2. Having done this, the first processor will send $F_1^{\max(1)}$ to the next processor. All other processors, with values of $p > 1$ will input $F_1^{\max(p-1)}$ and use this to compute $F_1^{\max(p)}$ from $F_{\min(p)}^{\max(p)} + F_1^{\max(p-1)}$.
3. In parallel with sending $F_1^{\max(p)}$ to the next processor in the column, each processor computes $F_1^i \forall i \in [\min(p), \max(p) - 1]$ giving the desired sums. After each processor has completed both the send and the computation it can proceed with the normal pipeline.

This reduces the startup time to the following

$$S = [u + (P - 1)(\tau + 1) + u]\tau. \quad (5.17)$$

The first term in Equation 5.17 comes from stage 1 where all processors compute in parallel the partial sums. The second term is the time that $F_1^{\max(V-1)}$ takes to arrive on the top processor. The last term is the time that the top processor spends computing the term F_1^N . The efficiency of the pipeline is now,

$$e = \frac{Ku}{2u + (P - 1)(\tau + 1) + (K - 1)\max(u, \tau)}. \quad (5.18)$$

In this case the number of processors at which 50% efficiency occurs is,

$$P = \frac{\tau}{\tau + 1}(K - 1) + 1, \quad (5.19)$$

which is approximately K for large values of r —about twice that of the previous case. Therefore, in the case of low communications speeds, relative to computation rates, the second method would be better.

One further problem is that each communication between processors only involves the transfer of one word. The effect of the communications startup time, τ_0 , as distinguished from the pipeline startup time, is quite significant. See Figure 1–2 and note how the bandwidth rises from low values for small message length. This effect can be reduced by grouping together tasks into one packet. The grouping factor is termed G and S_0 is defined to be either $2u$ or V depending on how the pipeline startup is done. Expression 5.17 and 5.14, including the communications start up time, τ_0 , can then be rewritten as

$$e = \frac{Ku}{GS_0 + (V - 1)[Gr + \tau_0] \frac{K-G}{G} \max(Gu, Gr + \tau_0)}. \tag{5.20}$$

Increasing G will increase the pipeline startup time but decrease the effect of the communications startup. The problem is identifying the optimum value of G .

There are three cases.

1. $u < r$, then the optimum value is $G = 1$
2. $u > r + \tau_0$, then the optimum value is $G = \max(\sqrt{\frac{Kr_0}{S_0 + (V-2)r}}, 1)$
3. $r \leq u \leq r + \tau_0$ then $G = \frac{r_0}{u-r}$

See Appendix C.3 for a proof of the above. If it is not possible to overlap communications then the efficiency is

$$e = \frac{Ku}{GS_0 + (V - 1)[Gr + \tau_0] + (K - G)[u + r + \tau_0/G]}. \tag{5.21}$$

The optimum value of G is also computed in Appendix C.3 and is

$$G = \max(\sqrt{\frac{Kr_0}{S_0 + (V-2)r - u}}, 1)$$

5.3 Global Models

This section will discuss the differences in implementing a global model compared to a limited area model. These differences are :-

1. A modified topology is required to take account of “cross-pole” communications.
2. Computing polar values of some variables is required by taking the mean value of the longitudinal row adjacent to the pole.
3. The need to work out the maximum velocity, for each row, between the equator and a row. This is done to compute the damping coefficients required for each timestep.
4. The need to Fourier damp variables on near-polar longitudes in order to give a reasonable timestep.

The cross-pole communications required by a global model on a sphere are between points directly opposite from one another over the poles. Such points are separated by 180° of longitude and are at the same latitude. This requires that processors should be connected to processors in the same row but $N/2$ processors away, where there are N processors in a row. Figure 5–8 shows this topology for a sphere with 16 processors.

5.3.1 Computing the Maximum Velocity

In order to correctly compute the damping coefficients for the Fourier damping it is necessary to compute the maximum velocity for the row and all rows between the equator and it. The ideas discussed in Section 2.5 are useful

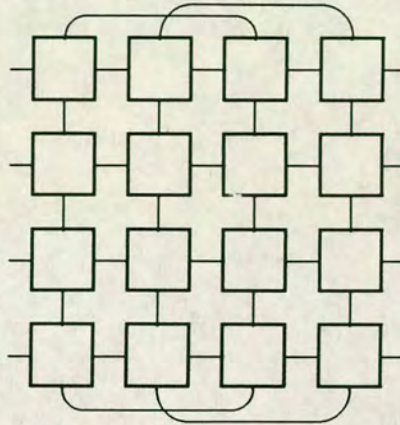


Figure 5-8: Processor topology used to map a sphere

This figure shows 16 processors configured into a sphere. The links shown at the east and west sides wrap around. The links shown at the north and south edges are connected to processors 2 away, to provide the 'over the pole' communications.

here as maximum is an associative operator. This operation can be divided into two stages.

1. Computing the row maximum velocity.
2. Computing the maximum speed of this row and all rows between the equator and it.

The computation of the first depends on the architecture being used and in particular the number of links that the processor has left after connecting it to its neighbours to form a grid. In the hardware being used at Edinburgh all links were connected to their four neighbours and so each row is a ring of processors. The row maximum is computed by the following algorithm which has also been described by Horiguchi and Miranker (1989).

1. The processor computes the row maximum for each row in its sub-domain using the points on each row that lie within its sub-domain. All processors can do this in parallel.
2. Each processor then sends its values of maximum out to its neighbour, (either east or west). It will then receive from its other neighbour some values.
3. These values will then be compared with the original values and the maximum stored.

Steps 2 and 3 will be repeated until each processor has processed $N - 1$ messages, there being N processors in the ring. Each processor will now have the global maximum stored for each row.

To compute the maximum speed for a row and all rows closer to the equator than it, requires similar algorithms to those described in Subsection 5.2.2, which were used to compute the vertical iteration schemes. The problem in this case is to compute, for a row k rows away from the equator, $U_k = \max(u_k, U_{k-1})$. This is an iterative scheme except that max is used rather than addition. The algorithms necessary are the same as those described in Subsection 5.2.2 except that rather than summing values the maximum should be taken.

The value of variables at points at the poles is the mean of the variable on the surrounding near-polar row. The algorithm required is the same as that described for the computation of the row maximum with sums being performed on incoming data rather than maximum.

It is worth noting here how a parallel algorithm for any associative operator can easily be extended to any other.

5.3.2 Fourier Transforms in Grid Point Models

At present, global grid point models use regular longitude-latitude grids. Near the poles the *physical* separation between grid points decreases and because of the CFL condition a shorter timestep will be required in order for the model to be numerically stable. This is unacceptable as it would require an extremely short timestep which would make integrations prohibitively expensive. In the early 1970's research was done into using skipped grids where, as the poles were approached, rows would have progressively fewer grid points in them (Williamson and Browning, 1973). These were found to have other problems, in particular there were large inaccuracies in large-scale cross-polar flow (Holloway *et al.*, 1973). Also see Williamson (1979) for a general overview of global grid-point methods.

The approach that is taken in the model implemented on the ECS is to linearise the discretised model equations and compute by how much a wavenumber will grow in one timestep using the von Neumann procedure described in Subsection 1.4.2. For stable wavenumbers there will be no amplification. If any wavenumbers in a row are unstable then the points in that row are transformed to Fourier space and unstable modes either damped or chopped. In damping the amplitude of a wavenumber is divided by its amplification factor times a constant factor, while in chopping a wavenumber is discarded if it is unstable. The constant factor is there to take some account of the non-linear interactions in the problem. After this has been done the data is transformed back from Fourier space to grid point space.

Fourier damping for the adjustment step, is only required on rows north and south of 60° N and 60° S respectively. For the advection, the decision of whether or not to do Fourier damping depends on the maximum velocity found in that row or in a row closer to the equator.

In the context of the two dimensional decomposition described earlier

there does not seem to exist an efficient implementation of the fast Fourier transform. The problem is that all four of the processors links are committed to connections with their neighbours and none are free to provide extra links for the FFT. The topology that is left is a ring—the problem is then is how to implement the FFT on this ring topology.

In Section 4.3 it was shown that an efficient implementation of the FFT was possible, by building a butterfly topology. Why not do that here? There should be enough tasks to make the FFT pipeline efficient, the pipeline length is $O(\log_2 N)$ while the number of vertical levels should increase in proportion to the total horizontal domain size. Each processor row would have a butterfly added to it. See Figure 5–9 for the modifications to a single row required to add the FFT butterfly in. The efficiency of the FFT is now the figure derived in Chapter 4 and will be approximately 50% if the communications speed, r , is less than 4. The problem is then what is to be done with these extra processors for the rest of the dynamics step? For the remainder of the dynamics step these processors will be idle, thus the utilization factor of the entire computer will be $O(1/\log_2 N)$ and this does not constitute an efficient implementation.

One alternative may be to increase the valency of each processor in order to provide the required connectivity. A hypercube topology could be built. If each processor has a valency of v , then $\frac{\log_2 N}{v}$ processors will be required to make each node⁴ have the required connectivity. Using homogeneous processors this has, in essence, recreated a hypercube but with $N \log_2 N/v$ processors. As in the previous example, these processors would be computationally idle for the remainder of the dynamics step. In contrast to the previous example, these extra processors would be idle during the FFT as

⁴There would be N nodes in total.

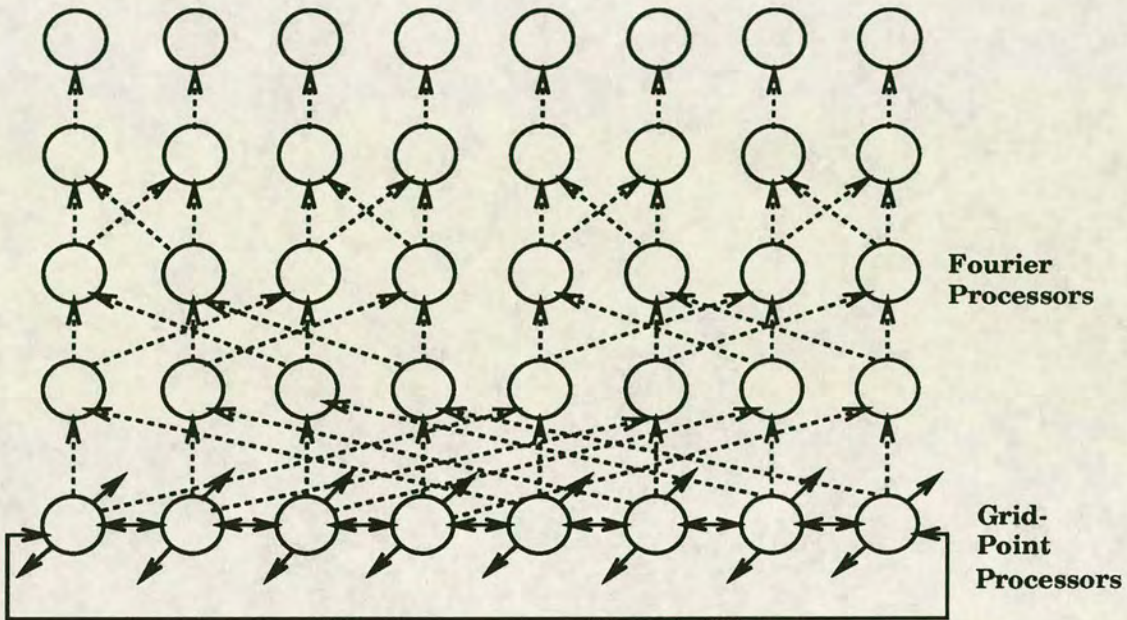


Figure 5-9: A grid point processor row with a FFT butterfly added
This figure shows a single row, with 8 processors, of the computer. Communications for the grid part of the problem are shown with lines, while the communications needed for the FFT are shown by dashed lines. Notice how the number of processors involved in computing the FFT is much greater than those used in computing the grid-point part of the calculations.

Table 5-5: Times for grid-point model with FFTs

Processors in Row	Time
16	14.0
8	6.64
4	1.52
2	0.21

Measured Times, in seconds, using four rows of processors and differing numbers of processors in a row are shown in the above table.

well. In a heterogeneous environment where some processors are designed principally as computational engines and the rest optimised for communications (small amounts of memory and no floating point unit) then this approach might be of some benefit. Use of wormhole routing techniques would be useful in reducing the latency of the communications.

Having rejected the various alternatives above, consider now implementing the Fourier transform on the two dimensional grid. The FFT will take place on longitudinal rows of the grid. For the FFT distributed over N processors with n points on each processor there are $\log_2 N - 1$ communications required where each processor will need to communicate with one other processor. Messages will need to go along $N/2$ links for the first stage and then $N/4$ for the second and so on. The total number of communications events required per processor is $N - 1$. Therefore the total time spent communicating will be $(N - 1)\tau n$, while the total time spent computing is $cn \log_2(nN)$, using the notation from earlier in this chapter. The total time spent computing could be reduced by overlapping though this would make very little difference to the total time taken. The time taken by the Fourier transform will grow linearly with the number of processors in the east/west row. Table 5-5 shows results for the grid point model on a sphere.

If a three-dimensional decomposition is used, then the butterfly pipeline could be used. In this case it is possible to build a butterfly topology without using any more processors, or nodes, though the valency of a single node would have to be at least eight. Consider a vertical east/west cross-section of the processor cuboid described in Subsection 5.2.2. This can be converted into a butterfly network by adding extra links between processors. Level 1 processors will be connected to level 2 processors $N/2$ away, in general a level V processor will be connected to a level $V + 1$ processor $N/2^V$ away. Figure 5–10 shows this.

In Section 4.2 it was shown that the length of the butterfly is $\log_2 N + K$, with K being a fixed constant proportional to $\log_2 n$. As was already stated earlier in this chapter, the number of processors over which a vertical decomposition should be carried out is limited. This limit is a function of the time spent computing the parameterization schemes, the communications speed and the number of atmospheric columns on a single processor. The larger these are the greater the number of processors over which the vertical decomposition could be carried out. Is this limited vertical decomposition, likely to form an unreasonable limit on the total number of processors that can be used? Assume that the number of vertical processors is 10, and further assuming that K is 2 then this would allow 256 processors in any row, very modest increases in the number of vertical processors would allow many more processors in any row. Note that 256 processors in a row, 10 processors vertically and 256 rows of processors would give a computer composed of approximately half a million processors, a not unreasonable limit.

5.3.3 Load-balancing the Fourier Transforms

When carrying out the FFTs as described, there is load imbalance. A deterministic calculation of the imbalance is possible for the adjustment step.

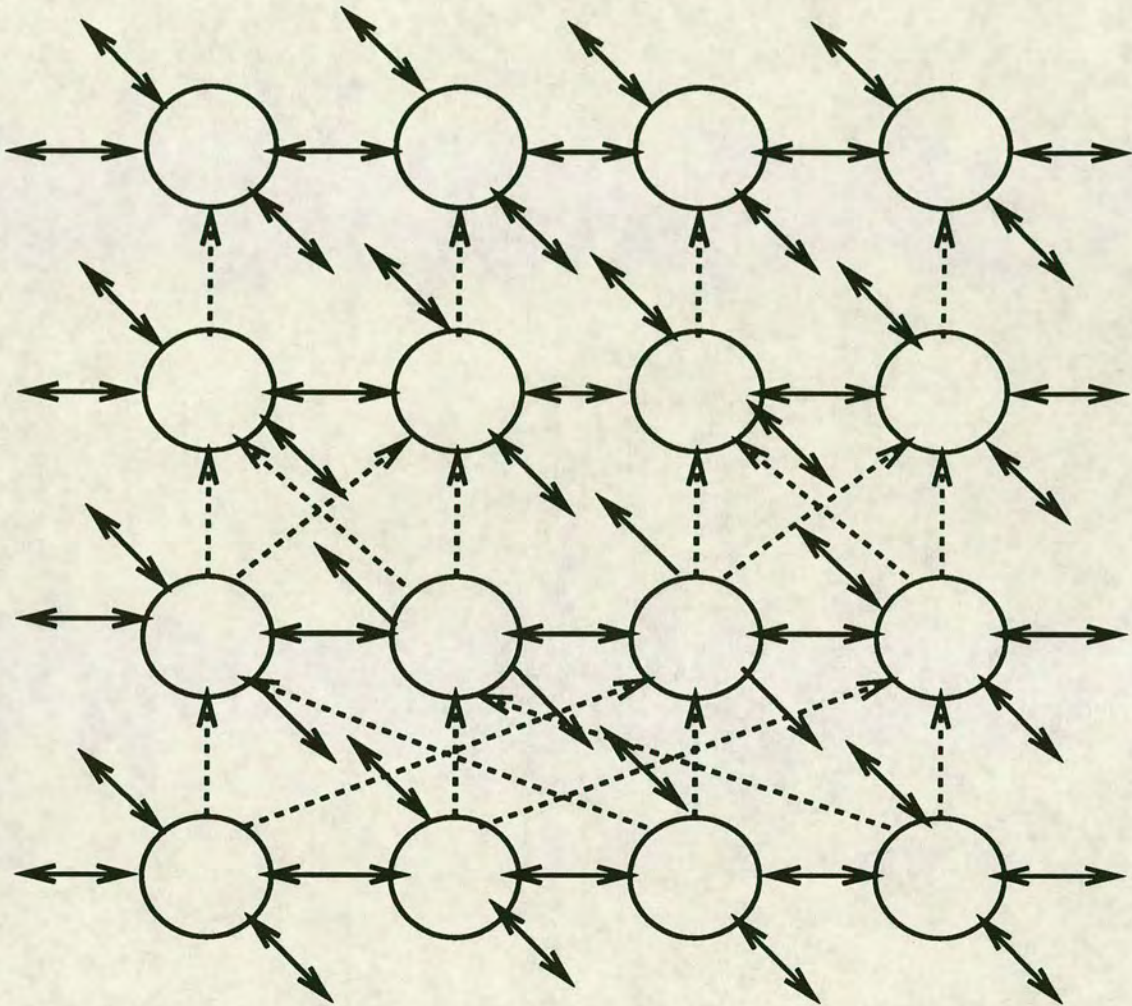


Figure 5-10: Vertical Butterfly

This figure shows part of a three-dimensional decomposition with four processors in the vertical. Solid arrows show the communications required for the horizontal finite difference computations, while dashed lines show the communications needed for the Fourier damping.

Here, Fourier damping will only take place polewards of 60° N/S. For the advection step, Fourier damping is dependent on the maximum speed on the row or rows closer to the equator than that row. For the adjustment step only one third of the processors will be taking part in the transforms, the remaining two thirds of the processors will be idle. In Chapter 3 a load-balancer was described. A similar balancer is needed here in order to move tasks from polar rows to near-equatorial rows. There is however one significant difference; tasks are divided over the processors in a row. When a processor makes a decision about how many tasks to move to a neighbour, *each* processor in the row must make the same decision about how many tasks to distribute to its neighbours. This necessitates that the random selection of neighbours described in Chapter 3 should not be used. Instead each processor will exchange information with both its north and south neighbours on how many tasks it has.

Each processor first computes the number of available tasks, which is the number of tasks it has minus the number of tasks it must keep to process (usually one). Each processor then computes how many tasks it would like to exchange with both of its neighbours. This is given by the difference in available tasks between them times a fractional value. This fractional value is termed the diffusion constant. If the total number of tasks to be moved is greater than the available tasks then the tasks moved to the north and south neighbours are reduced proportionally. All processors at this stage now know whether they will be receiving tasks, transmitting tasks or not taking part in any exchange. All processors then carry out any task exchange required. Processors with no tasks will “sleep” in the manner described in Section 3.3 in order to avoid live-lock. All other processors will then wait for the Fourier transform to be completed. If the architecture permits it, the Fourier transform runs in parallel with the task exchanger described earlier.

One important parameter for this load-balancer is the diffusion constant. In order to investigate how the load-balancer would behave, a simple model of it was constructed. The aims of this simple model, of a load-balancer, were to give a qualitative understanding of the importance of the diffusion parameter and to give some guidance on the behaviour of the load-balancer when using more processors that were available on the ECS.

In this idealized model each row of processors is represented by a single site with a number of tasks. The FFT takes a unit length of time. The number of tasks on the site is reduced by one to represent the FFT taking place. The site then exchanges tasks with its neighbouring sites in the manner described above. Two different simulations were done. In the first the maximum number of tasks that could be moved to any neighbour was set to N . This represents a grid topology where the FFT takes a time $O(N)$. In the second experiment the maximum number of tasks that could be exchanged was set to $\log_2 N$, this represents a butterfly topology where a FFT would take a time $O(\log_2 N)$. The results of the simulations are shown in Figure 5-11. These simulations are intended to give qualitative guidance to a choice of diffusion parameters. They suggest that only in the case where there are a large number of tasks per processor, will the time taken be significantly decreased compared to the non-balancing case. For the grid topology there may^{be} some sensitivity to the value of the diffusion coefficient. In addition this simple model predicts, for small numbers of processors rows, that speedups of almost 3 will be achieved.

This model of parallel computing is deficient in that the cost of a transfer is proportional to the amount of data that needs to be moved. In reality this may not be true, there is a high cost involved in the "dialogue" between processors when information is transferred in order to compute how many tasks should be moved.

In order to construct the load-balancer described above, it was necessary

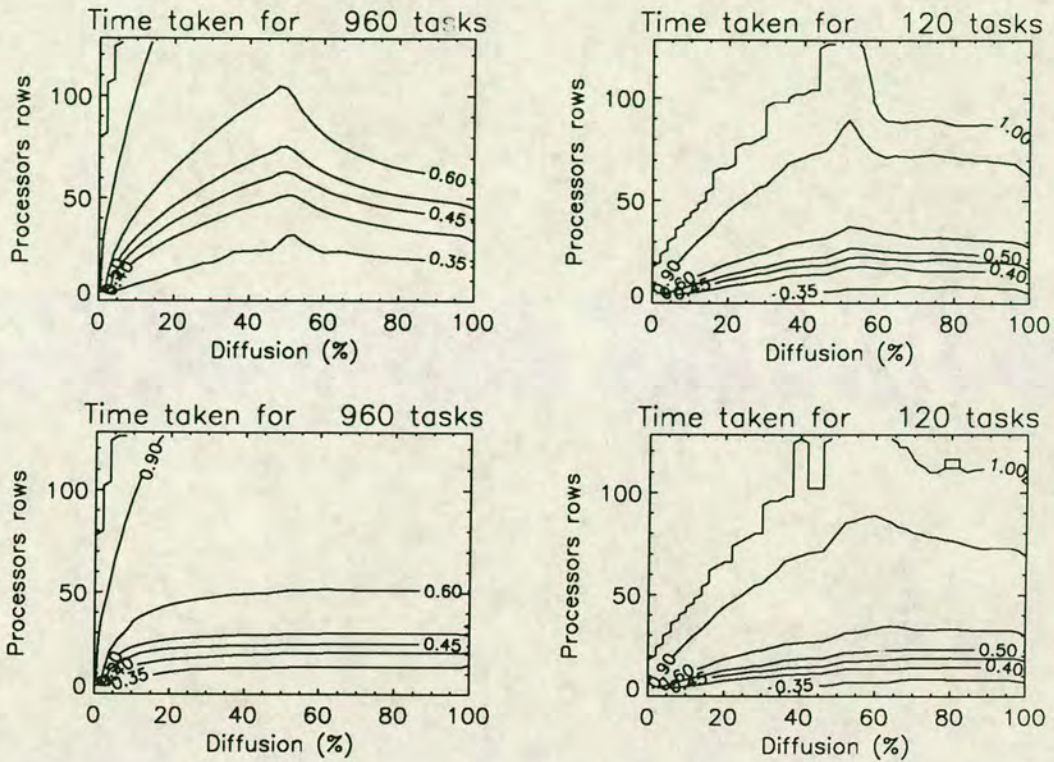


Figure 5-11: Simulated effects of FFT load-balancing

The four diagrams all show the time taken relative to the non-balanced case with varying numbers of processor rows (N) and diffusion constant. The top two plots are for a grid topology with different numbers of tasks per processor. The bottom two are for a butterfly topology. The difference between the two is in the maximum number of tasks that are allowed to be transferred. For the butterfly topology, a maximum of $\log_2 N$ tasks can be transferred, while for the grid topology a maximum of N tasks can be transferred. No results were computed for less than four processor rows. Contours are drawn at values of 0.35, 0.4, 0.45, 0.5, 0.6, 0.7, 0.8, 0.9 and 1.0. The number of tasks in the plots is given by the number of variables on full three-dimensional fields (four for the model in Appendix B) times the number of levels on a processor plus the number of surface only variables (one, surface pressure) all times the number of rows. For 8 rows and 5 vertical levels this would give 168 tasks.

Table 5-6: Times for grid-point model with software multiplexing

	Domains Size			
	4 × 4	8 × 8	16 × 16	32 × 32
Concurrent	2392	2910	4621	10700
Sequential	2364	2882	4622	10784

to have several processes running in parallel on the same processor. These are: the deterministic load-balancer; the router process; and the master process. All these processes need access to the N/S links while the load-balancer and the master process need access to the E/W links of the processor. In order to allow this, a software multiplexer/demultiplexer was designed and implemented in software. This multiplexer system caused the peak communications rate to drop by about a factor of a half and increased the startup time. Figure 1-2 shows the communications speeds for various different configurations; the two to compare are the “through routing” curve and the “multiplexer” curve. Table 5-6 shows the effect on the speed of execution of the grid-point model described in Section 5.2. This table should be compared to 5-3 to show what the effects of slow startup time are on the time taken for small sub-domains. Note that concurrent communications in these circumstances produces very little benefit and can even increase the time taken.

Figure 5-12 shows a plot of the times taken per site for the concurrent and the sequential cases. Compare Figure 5-12 to 5-5 and note that the effect of slow communications is most significant for small domains or low granularity.

The next set of results presented will show that the load-balancing of the near-polar Fourier transforms leads to an improvement. The experiment

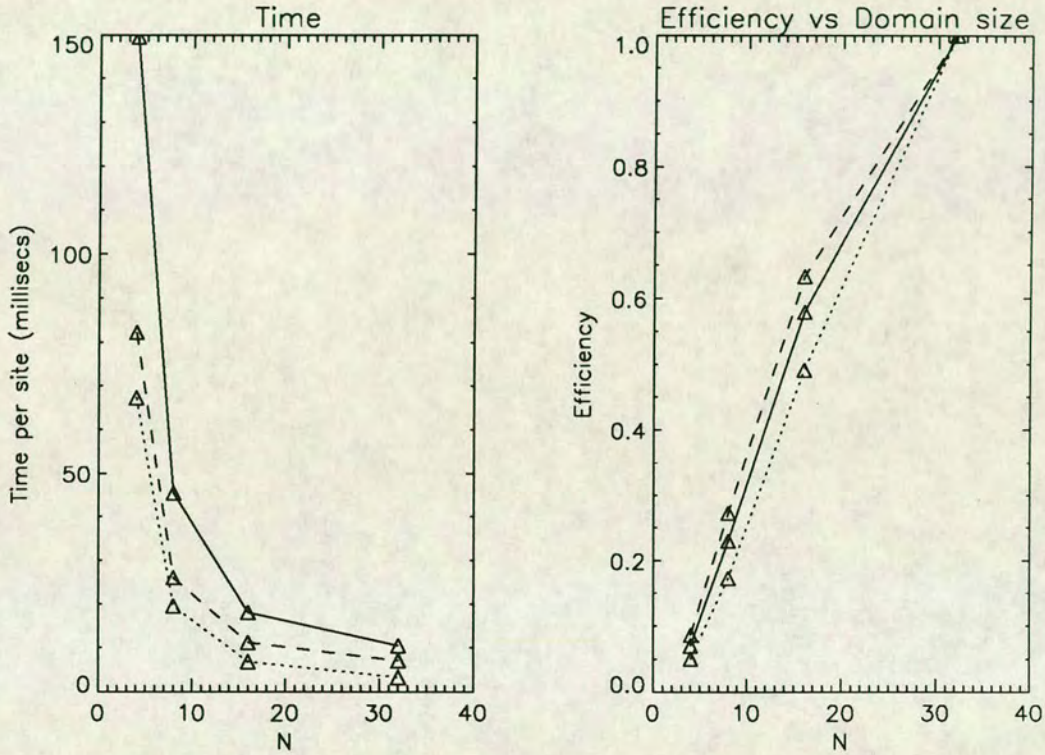


Figure 5-12: Times per site for multiplexer

The left hand graph shows the times for the concurrent communications while the right hand side shows the efficiency relative to the 32×32 case. The times for the non-concurrent communications case are not shown as they are very similar to the concurrent case. Values for the advection step are shown with dotted lines, the adjustment with dashed lines and the total timestep with a solid line.

Table 5–7: Times for FFT load-balancing

Processors	No load-balancing	Diffusion Coefft (%)					
	–	0	20	40	60	80	100
4 × 2	2057	2334	1912	1828	1809	1809	1934
4 × 4	2534	2887	1537	1463	1441	1418	1667
4 × 6	2537	2883	1255	1187	1103	1170	1244
4 × 8	2541	2888	1229	1102	1030	1022	1041

Times are in milli-secs and are accurate to 10 m secs.

that was carried out was to fix the size of the FFT but change the number of rows in the processor domain with the sub-domain size fixed at 8×8 horizontally and with 5 vertical levels. The only factor in the change in time taken by the FFT would be due to the load-balancer. Four different processor configurations were examined, ranging from 4×2 to 4×8 ; this notation means that the processors were arranged in a spherical topology with 8 rows and 4 columns. The transfer rate was allowed to vary from 0% to 100% in steps of 20%. Table 5–7 shows the times measured for the FFTs while Figure 5–13 shows a plot of the speedup relative to the non load-balanced case. Also shown in this figure are speedups from the simple model of the load-balancer. The times for the FFTs were obtained by measuring the total time taken and subtracting the grid-point times from these. These are obtained from the results in 5–7 and 5–5. Speedups of up to 2.5 are obtained. The measured results are not acting in the same way as the simple model would predict. The simple model predicts that speedup should decrease with increasing domain size rather than increase as measured. What is happening is that the fixed costs of the load-balancer become relatively less

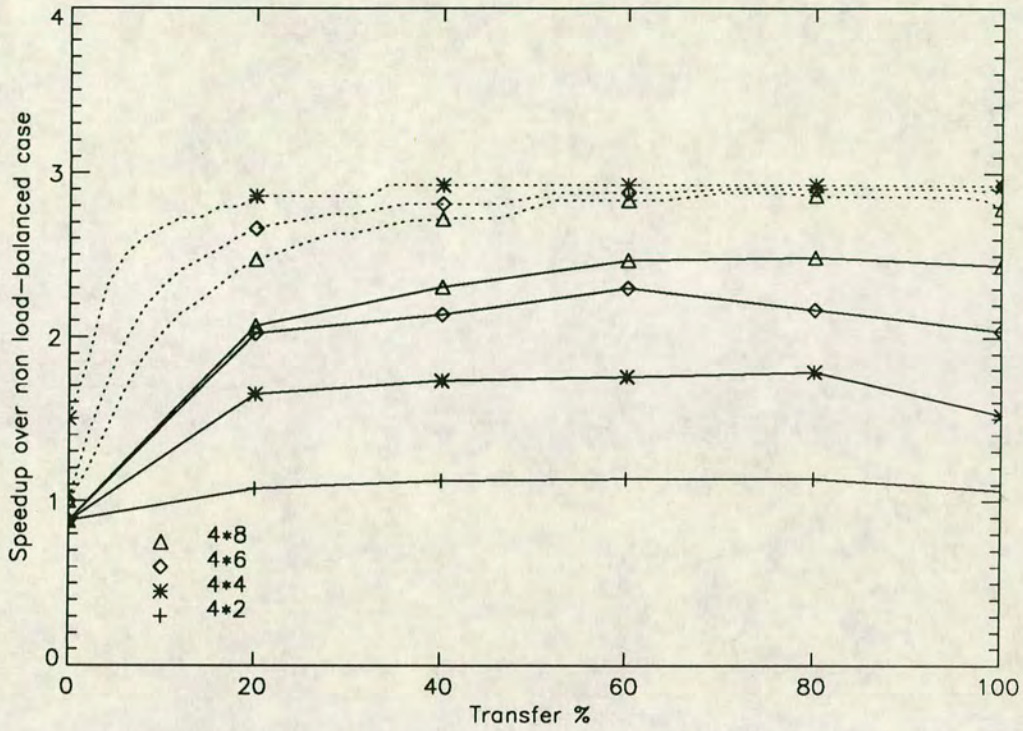


Figure 5-13: Speedups for the FFT with different amounts of load-balancing

Solid lines show the measured times for different processor domain sizes and changing values of the diffusion coefficient. Dotted lines are from the simple model of a load-balancer's behaviour. Symbols correspond to the different processor domain sizes.

important as the number of rows increase. The shape of the measured curves becomes more like the model curves as the number of rows becomes larger.

5.4 Conclusion

In this chapter, a three-dimensional limited area model was first examined. This model was implemented in the language Occam on the ECS. Measurements of the time taken by this model showed that it had a linear scaled speedup with increasing processor number. Times were measured when using concurrent and non-concurrent communications. The times taken when using non-concurrent communications were approximately 10% greater than the times taken when using concurrent communications. For the amount of work required to implement concurrent communications, there seems to be little gain from using concurrent communications for the bulk of a grid-point model code. Many vector computers are limited not by the processing speed of their CPU, but by the time it takes to transfer data from memory to the CPU and back again. If this were true for parallel computers, then concurrent I/O may not be possible as the memory bandwidth required for the transfer would not exist.

The concept of interaction range was introduced to measure the communications volume for any particular stage of the program and was then used to make predictions about the efficiency with variations in sub-domain size. The efficiency of an individual processor is expected to increase with increasing processor sub-domain size. This pattern was confirmed by the implementations. The effects of communications startup time were not considered in these calculations of efficiency, but the experience of the software multiplexer showed these could be quite considerable. The importance of the communications speed or alternatively the total communications bandwidth in the computer should be stressed. The grid-point model examined and implemented had a rather low ratio of computations work to communications. This model is typical of many meteorological grid-point models

and suggests that the performance of parallel computers for realistic meteorological applications will be less than some other applications, because of their higher ratio of computations to communications.

For models with only relatively small numbers of grid-points, such as climate models then relatively small numbers of processors could be used efficiently. How small that number is depends on the communications speed of the computer.

In order to increase the horizontal sub-domain size, a limited vertical decomposition should be used in addition to the horizontal decomposition. Two parallel algorithms were presented for iterative computation of the geopotential term though no implementations were carried out. The first algorithm is simpler, but for slow communication speeds the second one can use twice as many processors as the first before reaching half efficiency. However neither iterative algorithm would show linear *scaled* speedup while the horizontal decompositions would. There will be a limit on the amount of vertical decomposition that can be done, the time to compute the iterative scheme will eventually dominate the time taken by the rest of the problem. However it is likely that the need for the parameterization schemes to rearrange atmospheric columns so that a column was contained entirely on a single processor, would set the limit on the number of processors over which vertical columns can be decomposed.

For the global model, the major complication was the need to carry out Fourier transforms to control numerical instability. On the ECS these are extremely inefficient. It is believed by the author that the Fourier transforms could be done efficiently but only in the context of efficient long range communications. An efficient Fourier transform could be done by using a three-dimensional decomposition, though this would require a processor valency of eight (see page 194). A very simple model of a load-balancing strategy for the Fourier transforms was made. This model suggested that there

would some benefit in implementing load-balancing, this benefit falling with increasing processor number and decreasing number of tasks per processor. The experiments carried out using the model implemented in Occam on the ECS, with only up to eight processor rows, showed that the load-balancer could decrease the time taken for the Fourier transforms by a factor of two and a half.

Chapter 6

Conclusion

This, the final chapter of the thesis, will present some conclusions and then use these to provide some speculative thoughts on what a massively parallel supercomputer might look like in the meteorological centre of the future. Based on the work presented in this thesis the author believes that, before the end of the decade, massively parallel computers will replace the present types of super-computers. For meteorologists to benefit from these computers work would need to start in the near future to convert existing serial models.

Prior to presenting the conclusions proper, a summary of the results of Chapters 3, 4 and 5 will be given. In Chapter 3 the problems of implementing parameterization schemes on parallel computers were examined. For the data-parallel class of computers, it was concluded that implementation would be difficult and efficiencies low. Implementation of these schemes on MIMD computers should be straightforward based on the experiences of implementing two schemes on the Edinburgh Concurrent Supercomputer. Results were presented showing the variation in work for different atmospheric columns. A load-balancer was implemented but produced only a marginal increase in speed for small numbers of processors and a decrease

at larger numbers of processors. It was concluded that load-balancing was likely to be of little benefit for an operational model.

In Chapter 4, a parallel algorithm for the spectral method was presented. This algorithm was a pipeline built around the spectral transformation of that method. The analysis presented showed that the algorithm had good scaling properties but a major question remaining for this algorithm is what the effects of the parameterization schemes would be on the balance of the pipeline. Implementation of the algorithm by converting an existing three dimensional primitive equation code was relatively straightforward as the number of places where communications are required in the program is small. Measurements on different configurations of the parallel version of the model showed a high variability in efficiency. It was concluded that this was due to load-imbalance in the pipelines. By changing the number of Gaussian latitudes for a configuration and measuring the changing times taken, it was possible to extract the startup time and time-step for that configuration of the pipeline. Extrapolations were made of the algorithm's performance and it was concluded that a T168 forty level version of the model could achieve speedups of approximately 4000 when using 10000 processors.

The grid-point model of Chapter 5 has many places in the program where communications are required, in particular the finite difference computation. Coding of the three-dimensional model described in Appendix B was tedious due to the large number of places where communications were required. Hand conversion of similar models would be expensive and error prone. However their relatively simple structure should make partial automation of the process possible. See Gerndt (1989) for a description of automation for the finite difference part of such programs. The local-area part of the model has good scaled speedup properties. The FFT implemented in Chapter 5 was inefficient; successful implementation of the FFT in the context of grid-point models would require much work and a different hard-

ware from the ECS. This illustrates one problem in parallel computing; there may exist good parallel algorithms for parts of any problem but combining the algorithms may require a new processor topology or a movement of large amounts of data from processor to processor. Building a *deterministic* load-balancer to transfer FFT tasks from near polar regions of the processor domain to equatorial regions would be of some benefit.

Having summarised the conclusions from earlier chapters, some general remarks will be made about massively parallel computers.

When the dynamics schemes were examined, it was found that efficiency depended on the sub-domain size on each processor. The theoretical analysis showed that the ratio of communications speed to computational speed would determine the sub-domain size for optimum efficiency. The results of a practical implementation also confirmed this. The conclusion is therefore drawn that to use massively parallel computers it is necessary that the problems tackled have a large number of spatial degrees of freedom. In grid-point models there must be a large number of grid-points in a horizontal plane or, for a spectral model, a large value of the truncation number will be required. For meteorological models, the decomposition over the processors will be in the horizontal plane. Only a limited decomposition over the vertical dimension will be possible due to the tight vertical coupling intrinsic to the parameterization schemes. It is also likely that the number of points in a horizontal plane on a processor will be quite small (of order 100). Therefore, if the individual processors have vector units for the floating point computations, these should have high speeds for short vector lengths required on massively parallel computers.

However, for small problems with long time integration, such as climate models, the efficiencies achieved will not be so high. An extreme example of such a problem is described by James and James (1989) in which a T21 model was integrated for a thousand years.

For MIMD computers, the ability to run several processes on a single processor is advantageous as "automatic" load-balancing could be achieved. If, for example, a process was idle waiting for a communication to occur then the processor could run another process keeping the processor busy. To do this the processor needs to be able to rapidly swap between processes. The use of concurrent communications is likely to be of most use when several processes are run in parallel on an individual processor. Its use in speeding up the time taken was also examined. For the grid-point model a reduction in the time taken of approximately 10% over the non-concurrent case was obtained but with a considerable increase in program complexity. This small gain is probably not worth this extra complexity unless the addition of communications subroutine calls to the code and the restructuring of the code required could be done automatically by a compiler.

The programming model used throughout this thesis was that of Communicating Sequential Processes (CSP) where serial programs communicated with each other by message-passing. There is often a need to have several processes on a single processor. These processes are tightly coupled and normally consist of several processes surrounding a process doing the computations required. These additional processes all carry supporting functions, for example routing data to the correct processor. An example of this is the load-balancer described in Chapter 3. In this case the CSP model is deficient, in that it forces unnecessary copying of data between processes. Efficiency could be considerably improved if a shared memory mechanism was available for these intra-processor communications.

One topic that needs consideration is a choice between different parallel architectures, SIMD or MIMD. The work in this thesis has been with MIMD computers.

The most effective way to use a parallel computer is by geometrical decomposition, in which the computational domain is divided into several

patches. Each patch is then assigned to a single processor. This is true even for the pipelined spectral algorithm described in Chapter 4. For both schemes investigated, the spectral and the grid-point, algorithms for SIMD machines exist.

The algorithm for a regional or limited area model described in Section 5.2 is a data-parallel algorithm in that all processors follow the same sequence of instructions but on different data. The grid-point model implemented used "IF" parallelism in that, for most of the time, all processors do the same thing but can follow different paths for part of the time depending on the state of certain variables. This makes the programmer's or designer's task easier as the same program can then be written for all processors. The smaller the number of different paths that different processors can follow the easier will be the designers task. Following this approach means that a large MIMD computer will have many copies of this program running—one copy per processor.

In order to utilize a large machine, the sub-domain size per processor will be small. Existing meteorological models use large complex programs and it is conceivable that the amount of memory required to hold the compiled program on a single processor will be much greater than that required to store the variables used by the program on that processor. The total cost of the computer may therefore be dominated by the cost of the memory, on each of the many processors used to store the replicated program.

The transfer of existing parameterization codes to MIMD computers should be trouble free as long as they are written in a standard programming language, such as Fortran 77. Load-balancing is unlikely to increase the efficiency of the parameterization part of a model. The conversion of the Reading spectral model to run on a parallel computer though not trouble free did allow large amounts of the existing software to be reused.

MIMD machines are flexible, but this flexibility carries a cost. The danger is deadlock. In a large program with many people making modifications it is possible for one person to introduce the possibility of deadlock into the program¹ but it may not actually occur until much later when another modification is made to the program. This second modification causes a slight change in the detailed timing of the program—the potential deadlock then becomes realised.

Examining now SIMD machines, for dynamics schemes their efficiencies would be high and there would be no need for the massive replication of a single program required for MIMD machines. It seems likely to the author that the parameterization stage of a model would run at low efficiencies on data-parallel machines. An acceptable implementation would probably require a total rewrite of the existing code into a Fortran 90 array format and code restructuring. The author believes that, mainly due to the problems with using existing parameterization codes on SIMD machines, MIMD machines are preferable for atmospheric models. Where possible the model should be coded in a data-parallel style possibly using Fortran 90.

When designing algorithms for parallel computers there are three crucial issues which must be considered:

1. Work decomposition—the algorithm should be decomposed into pieces, such that processors have equal work to do between synchronisations.
2. Scaling—The algorithm should scale with increasing numbers of processors, though the subdomain size may remain fixed to achieve this.
3. Communications—communications are important. Placement of the processes on the hardware should be such as to minimise communi-

¹The process topology will have cycles in it.

cations costs. The importance of process placement will decrease as the total bandwidth of the machine, relative to the total computational speed of the machine, increases. For sufficiently high bandwidth, a computer which appears to have only one global memory could be built.

Many members of the meteorological community have experience with existing numerical methods. In addition they use large and complex computer codes. Based on the algorithms presented in this thesis the author feels that both grid-point and spectral methods could be used efficiently on massively parallel computers. There is no great restriction due to the nature of these computers to prefer grid-point methods to spectral methods or *vice versa*.

The author believes that automatic conversion of existing Fortran 77 codes will be a difficult task. An automatic system would find it difficult to have the necessary information about the entire code to find good distribution and mapping strategies.

The author believes the best method to use parallel machines is to begin with working serial code and convert it. This may not achieve optimum efficiency but it does allow some of the existing software to be reused. In addition, values of variables within the code can be compared between the serial version (running on a single processor) and the parallel version thus making it easier to verify that the parallel version of the model is producing the same results as the serial model. Exact agreement for the values of the variables of two models should not be expected if the parallel algorithm requires rearrangement of the order of computations as in the spectral model described in chapter 4. The necessary changes to the program code, for it to run on a parallel computer, will involve addition of communications; most likely through subroutine calls. After the model is running on the parallel computer it can then be further optimised.

Spectral method models will be easier to convert to parallel computers than existing grid-point models. This is because the places in the spectral method models where communications are required are few and can be easily identified, so the remainder of the code can be used with no changes. In grid-point models there are many places where communications are required. The use of the Fortran 90 language with its array syntax makes conversion of grid-point methods significantly easier. However to exploit this would require rewriting of most existing grid-point models. For both spectral and grid-point methods, existing parameterization scheme codes could be used on MIMD computers without any difficulties greater than transferring these codes from one serial computer to another.

To complete this chapter, and the main body of the thesis, a few speculative thoughts on what a massively parallel supercomputer should have for meteorological users will be presented. As stated earlier in this thesis, the need for increasing computing power is likely to make the use of massively parallel computers mandatory for atmospheric modelers. The "holy grail" that many organizations are searching for is a Teraflops (10^{12} floating point operations per second) computer. Such a computer would make tractable many problems which are at present computationally intractable.

The author imagines that such a computer would have 10,000 processors, with each processor being capable of sustained rate of 100M flops. The communications bandwidth of the computer would be at least 100M bytes per second per processor or a total bandwidth, for the entire machine, of 1T byte/second. If the communications speed could be increased beyond this point then programming the machine would be easier. The communications technology should support wormhole routing. In addition, hardware support for process to process communications rather than processor to processor communications should be provided. This would remove the need for the inefficient multiplexer/demultiplexer software solutions used for this

end in Chapters 5 and 3. Each processor would have a limited amount of memory available for program store but a program cache should be used and program fragments, or subroutines, could be brought in as required from slower, cheaper store, such as disk. The author estimates that such a computer could carry out a century integration of a T168L40 atmosphere model similar to that discussed in chapter 4 in approximately one week. In making this estimate the author has assumed that the time taken by the parameterization schemes is the same as that taken by the dynamics.

The design of the computer would need to take account of the hardware reliability. With so many components in a machine, the mean time between failure of some component, out of the many in the machine, would be rather short. The computer would probably need to automatically diagnose, remove and perhaps replace, the failed component. Software tools and subroutine libraries would be needed in order to use the machine effectively, examples of these include: communication libraries; load-balancing libraries; timing and analysis tools; and tools to aid programmers in their efforts to convert existing codes to run on such a machine.

Such a machine would be extremely powerful and produce a "fire-hose" of data; some mechanism would be required to rapidly store this data. Analysing the data produced by the massively parallel computer would be challenging; substantial computing resources would be required for this alone. However the primary requirement for this task is ease of use.

One final remark, the author believes that in order to solve the problems that face meteorologists increasingly powerful computers will be required. These computers will need to be parallel computers. It is hoped that this thesis has shown how a start could be made on implementing present atmospheric models on such computers.

failure of some component, out of the many in the machine, would be rather short. The computer would probably need to automatically diagnose, remove and perhaps replace, the failed component. Software tools and subroutine libraries would be needed in order to use the machine effectively, examples of these include: communication libraries; load-balancing libraries; timing and analysis tools; and tools to aid programmers in their efforts to convert existing codes to run on such a machine.

Such a machine would be extremely powerful and produce a “fire-hose” of data; some mechanism would be required to rapidly store this data. Analysing the data produced by the massively parallel computer would be challenging; substantial computing resources would be required for this alone. However the primary requirement for this task is ease of use.

One final remark, the author believes that in order to solve the problems that face meteorologists increasingly powerful computers will be required. These computers will need to be parallel computers. It is hoped that this thesis has shown how a start could be made on implementing present atmospheric models on such computers.

Appendix A

Terminology

This part of the appendix describes some of the notation used in the thesis, and a brief description of some technical terms. In addition explanations of some of the acronyms used is also given.

τ The number of words that can be transferred to a neighbouring processor along one link, in the time it takes to do one floating point operation.

τ_0 The startup time (in units of τ) for a single communication.

$\{a, b, \dots\} \rightarrow l$ means in order to compute values at a site labeled l values from the set of sites $\{a, b, \dots\}$ are required. This notation is based on that of Carver (1990).

τ is the time for a single floating point operation. τ is also used in several places to mean a time.

ν is the valency of the processor—how many communication links it has to other processors.

e is the efficiency.

c is the number of computation operations per site. It is normally used in comparison with communications speed. This measure is similar to the measure, f , of Hockney and Curington (1989) but for communications rather than memory access.

Time step the interval between results at the output end of a pipeline.

Startup time the time it takes for the first result to appear at the output end of a pipeline measured from the time when the first task enters the pipeline.

Wormhole routing message routing method where intermediate processors forward partial messages as they are received. Contrast to “store and forward” routing where the whole message must be received by an intermediate processor before passing it on.

MIMD Multiple Instruction, Multiple Data, computer architecture where each processor can do different operations on different data.

SIMD Single Instruction, Multiple Data. A computer architecture where all processors do the same operation on different data.

Data-Parallel a computer architecture where the same operation is performed on many data values simultaneously. Fortran 90 uses this model.

Scaled time the time taken multiplied by number of processors used. Used when the problem size remains constant as the number of processors increase.

Speedup Measure of decrease in time taken when using N processors compared to using one processor.

Scaled Speedup speedup measured when the problem size is allowed to grow with the number of processors used.

CSP Communicating Sequential Processes, a model of parallel computing in which processes interact by exchanging messages.

CCS Calculus of Communicating Systems. Another model of message-passing for parallel computing.

CPU Central Processing Unit.

ECS Edinburgh Concurrent Supercomputer.

FFT Fast Fourier Transform.

NWP Numerical Weather Prediction.

DAP Distributed Array Processor. A data-parallel computer.

CM Connection Machine. Another data-parallel computer.

Appendix B

Grid-point Model Description

B.1 Primitive Equations

In the following equations the symbols used and their meanings are p , the pressure, v , the horizontal velocity, ϕ the geopotential, f , the Coriolis term, T the temperature, q the specific humidity, and S an arbitrary scalar tracer. These are the equations for a coordinate $\eta(p_0, p_*)$ where $\eta(0, p_*) = 0$ and $\eta(p_*, p_*) = 1$. p_* is the surface pressure.

$$\frac{Dv}{Dt} + f\mathbf{k} \times v + \nabla\Phi + \frac{RT}{p}\nabla p = 0 \quad (\text{B.1})$$

$$\frac{DT}{Dt} - \frac{\kappa T\omega}{p} = 0 \quad (\text{B.2})$$

$$\frac{Dq}{Dt} = 0 \quad (\text{B.3})$$

$$\frac{DS}{Dt} = 0 \quad (\text{B.4})$$

$$\frac{\partial}{\partial \eta} \left(\frac{\partial p}{\partial t} \right) + \nabla \cdot \left(v \frac{\partial p}{\partial \eta} \right) + \frac{\partial}{\partial \eta} \left(\dot{\eta} \frac{\partial p}{\partial \eta} \right) = 0 \quad (\text{B.5})$$

$$\frac{\partial \Phi}{\partial \eta} + \frac{RT}{p} \frac{\partial p}{\partial \eta} = 0 \quad (\text{B.6})$$

where

$$\omega = \frac{Dp}{Dt} = - \int_0^\eta \nabla \cdot (v \frac{\partial p}{\partial \eta}) d\eta + v \cdot \nabla p \quad (\text{B.7})$$

B.2 Finite Difference Scheme Used

The following notation is used throughout the rest of the appendix;

$$\overline{f(i,j)}^\lambda = \frac{f(i + \frac{1}{2}, j) + f(i - \frac{1}{2}, j)}{2}$$

and

$$\delta_\lambda f(i,j) = \frac{f(i + \frac{1}{2}, j) - f(i - \frac{1}{2}, j)}{\delta\lambda}$$

The model uses the hybrid coordinates (η) used by the European centre for Medium Range Weather Forecasting and described by Simmons and Strüfing (1981). For each model level η is defined implicitly by the following relationship for the half model level pressures.

$$p_{k+\frac{1}{2}} = A_{k+\frac{1}{2}} p_0 + B_{k+\frac{1}{2}} p_* \quad (\text{B.8})$$

$A_{k+\frac{1}{2}}$ and $B_{k+\frac{1}{2}}$ are constants which are defined by the model. p_0 is a constant pressure and, as usual, p_* is the surface pressure.

B.2.1 Adjustment Step

Each adjustment step consist of:

$$p_*^{n+1} = p_*^n - \delta t \sum_{m=1}^{NLEV} \nabla \cdot v_m \overline{\Delta p_m}^{\lambda\theta} \quad (\text{B.9})$$

The vertical advection is computed at this point and is given by

$$\left(\dot{\eta} \frac{\partial p}{\partial \eta} \right)_{k+\frac{1}{2}} = \left\{ \left(\frac{\partial p}{\partial p_*} \right)_{k+\frac{1}{2}} \sum_{r=1}^{NLEV} \nabla \cdot (v_r \overline{\Delta p_r}^{\lambda\theta}) \right\} - \sum_{r=1}^k \nabla \cdot (v_r \overline{\Delta p_r}^{\lambda\theta}) \quad (\text{B.10})$$

which using Equation B.8 can be rewritten as:

$$\left(\dot{\eta} \frac{\partial p}{\partial \eta}\right)_{k+\frac{1}{2}} = \{B_{k+\frac{1}{2}} \sum_{r=1}^{\text{NLEV}} \nabla \cdot (v_r \overline{\Delta p_r^{\lambda\theta}})\} - \sum_{r=1}^k \nabla \cdot (v_r \overline{\Delta p_r^{\lambda\theta}}) \quad (\text{B.11})$$

for $k = 0$ and $k = \text{NLEV}$ $\left(\dot{\eta} \frac{\partial p}{\partial \eta}\right)_{k+\frac{1}{2}} = 0$. In order to conserve energy the following expression for the vertical advection is used,

$$\left(\dot{\eta} \frac{\partial F}{\partial \eta}\right)_k = \frac{\left(\dot{\eta} \frac{\partial p}{\partial \eta}\right)_{k+\frac{1}{2}}(F_{k+1} - F_k) + \left(\dot{\eta} \frac{\partial p}{\partial \eta}\right)_{k-\frac{1}{2}}(F_k - F_{k-1})}{2\Delta p_k} \quad (\text{B.12})$$

where F is the quantity being advected.

The finite difference approximation to the $\frac{\kappa T \omega}{p}$ term in Equation B.2 is

$$\begin{aligned} \frac{\kappa T_k \omega_k}{p} &= \kappa T_k \{ -[\{\ln \frac{p_{k+\frac{1}{2}}}{p_{k-\frac{1}{2}}} \{ \sum_{r=1}^{k-1} \nabla \cdot (v_r \overline{\Delta p_r^{\lambda\theta}}) \} \\ &\quad + \alpha_k \nabla \cdot (v_k \overline{\Delta p_k^{\lambda\theta}})] \} / \Delta p_k \\ &\quad + v_k \cdot \nabla \left(\frac{p_{k+\frac{1}{2}} \ln p_{k+\frac{1}{2}} - p_{k-\frac{1}{2}} \ln p_{k-\frac{1}{2}}}{\Delta p_k} \right)^{\lambda\theta} \} \end{aligned} \quad (\text{B.13})$$

giving for the finite difference approximation to Equation B.2:

$$T^{n+1} = T^n - \delta t \left\{ \left(\dot{\eta} \frac{\partial T}{\partial \eta}\right)_k - \frac{\kappa T_k \omega_k}{p_k} \right\} \quad (\text{B.14})$$

where the first and second term on the lefthand side are defined in Equations B.10 and B.13 respectively.

$$q^{n+1} = q^n - \delta t \left(\dot{\eta} \frac{\partial q}{\partial \eta}\right)_k \quad (\text{B.15})$$

The equation for the update for any scalar is identical in form to that of Equation B.15. These are all done with variables from timestep n .

Followed by for each level, the computation of values of velocity by;

$$u^{n+1} = w_1 u^n + w_2 v^n - w_3 (P_\lambda + \left(\dot{\eta} \frac{\partial u}{\partial \eta}\right)) - w_4 (P_\theta + \left(\dot{\eta} \frac{\partial v}{\partial \eta}\right)) \quad (\text{B.16})$$

$$v^{n+1} = w_1 v^n - w_2 u^n - w_3 (P_\theta + \left(\dot{\eta} \frac{\partial v}{\partial \eta}\right)) + w_4 (P_\lambda + \left(\dot{\eta} \frac{\partial u}{\partial \eta}\right)) \quad (\text{B.17})$$

where

$$\nabla \cdot (X, Y) = \frac{1}{a \cos \theta} \{ \delta_\lambda \bar{X}^\theta + \delta_\theta (\overline{\cos \theta Y})^\lambda \} \quad (\text{B.18})$$

$$\nabla P = \frac{1}{a} \left(\frac{\delta_\lambda \bar{P}^\theta}{\cos \theta}, \delta_\theta \bar{P}^\lambda \right) \quad (\text{B.19})$$

$$w_1 = \frac{1 - F^{n^2} \delta t^2 / 4}{1 + F^{n^2} \delta t / 4} \quad (\text{B.20})$$

$$w_2 = \frac{F^n \delta t}{1 + F^{n^2} \delta t^2 / 4} \quad (\text{B.21})$$

$$w_3 = \frac{\delta t}{1 + F^{n^2} \delta t^2 / 4} \quad (\text{B.22})$$

$$w_4 = \frac{F^n \delta t^2 / 2}{1 + F^{n^2} \delta t^2 / 4} \quad (\text{B.23})$$

$$F^n = \left(f + \frac{u^n \tan \theta}{a} \right) \quad (\text{B.24})$$

$$P_\lambda = \frac{1}{a \cos \theta} \left(\delta_\lambda \Phi_k + R \bar{T}_k^{-\lambda} \delta_\lambda \frac{p_{k+\frac{1}{2}} \ln p_{k+\frac{1}{2}} - p_{k-\frac{1}{2}} \ln p_{k-\frac{1}{2}}}{\Delta p_k} \right) \quad (\text{B.25})$$

$$P_\theta = \frac{1}{a} \left(\delta_\theta \Phi_k + R \bar{T}_k^{-\theta} \delta_\theta \frac{p_{k+\frac{1}{2}} \ln p_{k+\frac{1}{2}} - p_{k-\frac{1}{2}} \ln p_{k-\frac{1}{2}}}{\Delta p_k} \right) \quad (\text{B.26})$$

with

$$\Phi_{k-\frac{1}{2}} = \Phi_{k+\frac{1}{2}} + R(T_v)_k \ln \frac{p_{k+\frac{1}{2}}}{p_{k-\frac{1}{2}}} \quad (\text{B.27})$$

$$\Phi_k = \Phi_{k+\frac{1}{2}} + \alpha_k R(T_v)_k \quad (\text{B.28})$$

$$\alpha_k = 1 - \frac{p_{k-\frac{1}{2}} \ln \frac{p_{k+\frac{1}{2}}}{p_{k-\frac{1}{2}}}}{\Delta p} \quad (\text{B.29})$$

$$(T_v)_k = T(1 + 0.61q) \quad (\text{B.30})$$

In order to stop grid separation the following "cross-grid" term is added,

$$p_* = p_* - \sum_{\tau=1}^{NLEV} \left[\frac{\delta_\lambda [\overline{\Delta p}^\lambda \delta_\lambda \Phi + R \bar{T}^\lambda \delta_\lambda \Delta p]}{\cos \theta} + \frac{\delta_\theta [\overline{\Delta p}^\theta \delta_\theta \Phi + R \bar{T}^\theta \delta_\theta \Delta p]}{\cos \theta} \right. \\ \left. - \frac{\delta_\lambda [\overline{\Delta p}^{\lambda\theta} \delta_\lambda \Phi^\theta + R \bar{T}^{\lambda\theta} \delta_\lambda \overline{\Delta p}^\theta]}{\cos \theta} + \frac{\delta_\theta [\overline{\Delta p}^{\lambda\theta} \delta_\theta \Phi^\lambda + R \bar{T}^{\lambda\theta} \delta_\theta \overline{\Delta p}^\lambda]}{\cos \theta} \right].$$

This adjustment step is carried out three times.

B.2.2 Advection Scheme

The advection step consists of a modified Lax-Wendroff scheme. Only the advection step for T is shown, both q and S (an arbitrary scalar) will have the same form.

$$u^{n+\frac{1}{2}} = [\bar{u}^{\lambda\theta} - \frac{\Delta t}{2a} \left\{ \frac{\bar{u}^{\lambda\theta}}{\cos \theta} \delta_\lambda \bar{u}^\theta + \bar{v}^{\lambda\theta} \delta_\theta \bar{u}^\lambda \right\}]^n \quad (\text{B.32})$$

$$v^{n+\frac{1}{2}} = [\bar{v}^{\lambda\theta} - \frac{\Delta t}{2a} \left\{ \frac{\bar{u}^{\lambda\theta}}{\cos \theta} \delta_\lambda \bar{v}^\theta + \bar{v}^{\lambda\theta} \delta_\theta \bar{v}^\lambda \right\}]^n \quad (\text{B.33})$$

$$T^{n+\frac{1}{2}} = [\bar{T}^{\lambda\theta} - \frac{\Delta t}{2a} \left\{ \frac{u}{\cos \theta} \delta_\lambda \bar{T}^\theta + v \delta_\theta \bar{T}^\lambda \right\}]^n \quad (\text{B.34})$$

where $\Delta t = 3\delta t$, followed by

$$u^{n+1} = u^n - \frac{\Delta t}{a} \left\{ \frac{\bar{u}^{\lambda\theta}}{\cos \theta} [(1+c)\delta_\lambda \bar{u}^\theta - c\left\{ \frac{2}{3}\delta_{3\lambda} \bar{u}^\theta + \frac{1}{3}\delta_{3\lambda} \bar{u}^{3\lambda} \right\}] \right. \\ \left. + \bar{v}^{\lambda\theta} [(1+c)\delta_\theta \bar{u}^\lambda - c\left\{ \frac{2}{3}\delta_{3\theta} \bar{u}^\lambda + \frac{1}{3}\delta_{3\theta} \bar{u}^{3\theta} \right\}] \right\}^{n+\frac{1}{2}} \quad (\text{B.35})$$

$$v^{n+1} = v^n - \frac{\Delta t}{a} \left\{ \frac{\bar{u}^{\lambda\theta}}{\cos \theta} [(1+c)\delta_\lambda \bar{v}^\theta - c\left\{ \frac{2}{3}\delta_{3\lambda} \bar{v}^\theta + \frac{1}{3}\delta_{3\lambda} \bar{v}^{3\lambda} \right\}] \right. \\ \left. + \bar{v}^{\lambda\theta} [(1+c)\delta_\theta \bar{v}^\lambda - c\left\{ \frac{2}{3}\delta_{3\theta} \bar{v}^\lambda + \frac{1}{3}\delta_{3\theta} \bar{v}^{3\theta} \right\}] \right\}^{n+\frac{1}{2}} \quad (\text{B.36})$$

$$T^{n+1} = T^n - \frac{\Delta t}{a} \left\{ \frac{u}{\cos \theta} [(1+c)\delta_\lambda \bar{T}^\theta - c\left\{ \frac{2}{3}\delta_{3\lambda} \bar{T}^\theta + \frac{1}{3}\delta_{3\lambda} \bar{T}^{3\lambda} \right\}] \right. \\ \left. + v [(1+c)\delta_\theta \bar{T}^\lambda - c\left\{ \frac{2}{3}\delta_{3\theta} \bar{T}^\lambda + \frac{1}{3}\delta_{3\theta} \bar{T}^{3\theta} \right\}] \right\}^{n+\frac{1}{2}} \quad (\text{B.37})$$

where

$$c = 3/4(1 - \mu^2); \\ \mu = \left(\frac{u^2 \Delta t^2}{(a \cos \theta \Delta \lambda)^2} + \frac{v^2 \Delta t^2}{\Delta y^2} \right)^{\frac{1}{2}}.$$

Note that all the blocks in Equation B.32 have an I_R of 1 while some of the blocks in Equation B.35 have an I_R of 2.

Appendix C

Technical Proofs

C.1 Derivations for the FFT

C.1.1 Mapping Proof

Proof that $j = 2^{q-l}k^{-1} + i$ satisfies properties 4.4, 4.5 and 4.6. These desired properties are;

$$\{j(l, k, i) + 1, j(l, k, i + 2^{q-(l+1)}) + 1\} \mapsto j(l+1, k, i) + 1$$
$$\forall i \in \{0, \dots, 2^{q-(l+1)-2}\}, \quad k \in \{0, \dots, 2^l - 1\}$$

$$j(l+1, k, i) = j(l, k, i),$$
$$\forall k \in \{0, \dots, 2^l - 1\}, \quad \forall i \in \{0, \dots, 2^{q-l} - 1\}$$

$$j(l+1, k + 2^l, i) = j(l, k, i + 2^{q-(l+1)}),$$
$$\forall k \in \{0, \dots, 2^l - 1\}, \quad \forall i \in \{0, \dots, 2^{q-l} - 1\}$$

To prove that the expression satisfies requirement 4.4 first consider $j(l, k, i + 1), \forall i < 2^{(q-l)} - 1$.

$$j(l, k, i + 1) = 2^{(q-l)}k^{-1} + i + 1, \quad \forall i \in \{0, \dots, 2^{(q-l)} - 1\}$$
$$\Rightarrow j(l, k, i + 1) = j(l, k, i) = 1 \quad \forall i \in \{0, \dots, 2^{(q-l)} - 1\}$$

Similarly $j(l, k, i + 2^{(q-(l+1))} + 1)$ is $j(l, k, i + 2^{(q-(l+1))}) + 1 \quad \forall i \in \{0, \dots, 2^{(q-l)} - 1\}$.

Now consider

$$\{j(l, k, i + 1), j(l, k, i + 2^{(q-(l+1))} + 1)\} \mapsto j(l + 1, k, i + 1)$$

and substitute the expressions above for j to obtain

$$\{j(l, k, i) + 1, j(l, k, i + 2^{(q-(l+1))} + 1)\} \mapsto j(l + 1, k, i) + 1$$

To prove that the expression for j satisfies requirements 4.5 and 4.6 consider $k' \in \{0, 2^{l-1} - 1\}$ then,

$$\begin{aligned} \widetilde{k'}^l &= 2\widetilde{k'}^{(l-1)} \\ \widetilde{k' + 2^{l-1}}^l &= 2\widetilde{k'}^{(l-1)} + 1 \end{aligned}$$

A proof that the expression for j satisfies requirement 4.5 is given below,

$$\begin{aligned} j(l + 1, k, i) &= 2^{(q-(l+1))} \widetilde{k}^{(l+1)} + i, \quad \forall k \in \{0, \dots, 2^l - 1\} \\ \Rightarrow j(l + 1, k, i) &= 2^{(q-(l+1))} 2\widetilde{k}^l \\ \Rightarrow j(l + 1, k, i) &= j(l, k, i). \end{aligned}$$

A proof that the expression for j also satisfies requirement 4.6 follows,

$$\begin{aligned} j(l + 1, k + 2^l, i) &= 2^{(q-(l+1))} \widetilde{k + 2^l}^{l+1} + i, \quad \forall k \in \{0, \dots, 2^l - 1\} \\ \Rightarrow j(l + 1, k + 2^l, i) &= 2^{(q-(l+1))} (2\widetilde{k}^l + 1) + i \\ \Rightarrow j(l + 1, k + 2^l, i) &= j(l, k, i + 2^{(q-(l+1))}) \end{aligned}$$

C.1.2 Proofs for the Real packing of the FFT

It is necessary to prove that the following 2 mappings are equivalent to

$$m \mapsto M/2 - m \quad \forall m \in \{1, \dots, M/2 - 1\}$$

$$\text{shiftright}(\text{reverse}(M/2 - m, M/2), M/2)$$

$$\text{reverse}(\text{shiftright}(M/2 - m, M/2), M/2)$$

Begin by considering $\text{shiftright}(\text{reverse}(M/2 - m, M/2), M/2)$. Use definition 4.1 for shiftright and reverse to obtain;

$$\begin{aligned}
 & \text{shiftright}(\text{reverse}(m, M/2)) \\
 & \equiv \text{reverse}(m, M/2) \mapsto (\text{reverse}(m, M/2) + 1) \bmod M/2 \\
 & \equiv m \mapsto ((M/2 - m + 1) \bmod M/2 + 1) \bmod M/2 \\
 & \equiv m \mapsto (M/2 - m) \bmod M/2 \\
 & \equiv m \mapsto M/2 - m
 \end{aligned}$$

The last two steps in the above proof use the limits on the range of m . Step 3 requires that $(M/2 - m + 1) \in \{0, \dots, M/2 - 1\}$ which is true because of the limits on the range of m . Step 4 requires that $M/2 - m \in \{0, \dots, M/2 - 1\}$ which is also true.

Now consider $\text{reverse}(\text{shiftright}(m, M/2))$

$$\begin{aligned}
 & \text{reverse}(\text{shiftright}(m, M/2)) \\
 & \equiv \text{shiftright}(m, M/2) \mapsto (M/2 - (\text{shiftright}(m, M/2))) \bmod M/2 \\
 & \equiv m \mapsto (M/2 - ((m - 1) \bmod M/2 + 1) \bmod M/2 \\
 & \equiv m \mapsto M/2 - m
 \end{aligned}$$

In the above proof the limited range of m is also used.

C.2 Derivations for the Legendre Tree

Define e , the efficiency as $\frac{P_{\text{useful}}}{P_{\text{total}}}$. P_{useful} is the number of processors doing useful work, the number of leaf processors and level $d_L - 1$ processors. P_{total} is the total number of processors.

$$P_{\text{useful}} = (v - 1)^{d_L} + (v - 1)^{d_L - 1} \quad (\text{C.1})$$

$$P_{\text{total}} = \sum_{n=0}^{d_L} (v-1)^{d_L} \tag{C.2}$$

⇒

$$e = \frac{v(v-1)^{-1}}{\sum_{n=0}^{D_L} (v-1)^{-n}} \tag{C.3}$$

Utilize the following relationship,

$$\sum_{n=0}^k x^n = \frac{1-x^{(k+1)}}{1-x} \tag{C.4}$$

to rewrite C.3 as

$$\begin{aligned} e &= \frac{v(v-1)^{-2}(v-2)}{1-(1/(v-1))^{d_L}} \\ \Rightarrow e &= \frac{v(v-2)}{(v-1)^2 - (v-1)^{-(d_L-1)}} \end{aligned} \tag{C.5}$$

This is the required expression for Equation 4.23. The next proof required is this in terms of the total number of processors. The total number of processors in the tree is $\frac{(v-1)^{(d_L+1)}-1}{v-2}$. After some manipulation this can be rewritten into the following form,

$$(v-1)^{-(d_L-1)} = \frac{(v-1)^2}{P(v-2)+1} \tag{C.6}$$

and substituting this into expression C.5 to obtain,

$$e = \frac{v(v-2)}{(v-1)^2} + \frac{v(v-2)}{(v-1)^2 P} \tag{C.7}$$

C.3 Vertical Iterations Grouping factor

Begin with the expression for the efficiency (a copy of Equation 5.20)

$$e = \frac{Ku}{GS_0 + \frac{K-G}{G} \max(Gu, Gr + r_0)} \quad (C.8)$$

The aim is to maximise e , which is done by minimising the denominator. If the denominator is rewritten $\max(d_1, d_2)$ where

$$d_1 = GS_0 + (V - 1)(Gr + r_0) + (K - G)u \quad (C.9)$$

$$d_2 = GS_0 + (V - 1)(Gr + r_0) + (K - G)(r + r_0/G). \quad (C.10)$$

There are three cases that require consideration.

1. $d_1 > d_2 \quad \forall G$ i.e. $u > r + r_0$.
2. $d_1 < d_2 \quad \forall G$ i.e. $u < r$
3. $u \geq r$ and $u \leq r + r_0$

In case 1 d_1 should be minimised to maximise e .

$$\frac{\partial d_1}{\partial G} = S_0 + (V - 1)r - u \quad (C.11)$$

But from the definition of S_0 in Subsection 5.2.2, S_0 is always greater than u and therefore $\frac{\partial d_1}{\partial G}$ is always positive. Thus the minimum value of d_1 occurs at $G = 1$, giving a value for G_{\min} of 1.

In case 2 d_2 should be minimised to maximise e

$$\frac{\partial d_2}{\partial G} = S_0 + (V - 2)r - Kr_0/G^2 \quad (C.12)$$

The minimum of d_2 occurs at $Kr_0/G^2 = S_0 + (V - 2)r$ which gives for G_{\min} the following expression

$$G_{\min} = \max \left(1, \sqrt{\frac{Kr_0}{S_0 + (V - 2)r}} \right) \quad (C.13)$$

For case 3 then the minimum value will occur at the intersection of the functions d_1 and d_2 . This occurs at $u = r + r_0/G$ giving,

$$G_{\min} = \frac{r_0}{u - r} \quad (\text{C.14})$$

Bibliography

- Arfken, G. 1985 *Mathematical Methods for Physicists*. Academic Press, New York.
- Active Memory Technology 1990 *DAP series FORTRAN Plus Language (Enhanced)*. AMT Ltd, 65 Suttons Park Avenue, Reading RG6 1AZ.
- Batchelor, G. K. 1967 *An Introduction to Fluid Dynamics*. Cambridge University Press, Cambridge.
- Bell, R. and Dickinson, A. 1987 The Meteorological office operational numerical prediction scheme. Scientific Paper 41, U.K. Meteorological Office, Her Majesty's Stationary Office, London.
- Bengtsson, L. 1991 Advances and prospects in numerical weather prediction. *Q. J. R. Meteorol. Soc.*, **117**(501), 855-902.
- Boillat, J., Brugé, F., and Kropf, P. 1991 A dynamic load balancing algorithm for molecular dynamics simulation on multi-processor systems. *Journal of Computational Physics*, **96**, 1-14.
- Boillat, J. 1989 Load balancing and poisson equation in a graph. Preprint.
- Bourke, W. 1974 A multi-level spectral model.I. formulation and hemispheric integrations. *Monthly Weather Review*, **102**, 687-701.
- Bourke, W. 1988 Spectral methods in global climate and weather prediction models. In Schlesinger, M. E., editor, *Physically-Based Modelling and Simulation of Climate and Climate Change*. Kluwer Academic Publishers.

- Bowler, K., Kenway, R., Pawley, G., and D.Roweth 1987 *An Introduction to Occam 2 Programming*. Chartwell-Bratt, Sweden.
- Bowler, K., Kenway, R., and Wallace, D. J. 1989 The Edinburgh concurrent supercomputer: project and applications. In C.R.Jesshope and Reinartz, R., editors, *CONPAR 88*, 635 – 642. Cambridge University Press.
- Branković, C., Palmer, T. N., Molteni, F., Tibakdi, S., and Cubasch, U. 1990 Extended range predictions with ECMWF models: Time-lagged ensemble forecasting. *Q. J. R. Meteorol. Soc.*, **116**(494), 867–912.
- Brent, R. P. 1974 The parallel evaluation of general arithmetic expressions. *Journal of the Association for Computing Machinery*, **21**(2), 201–206.
- Carver, G. 1988 A spectral meteorological model on the ICL DAP. *Parallel Computing*, **8**, 121–126.
- Carver, G. 1990 *Meteorological Modelling on the ICL Distributed Array Processor and other parallel computers*. PhD thesis, Edinburgh.
- Cats, G., Middelkoop, H., Streefland, D., and Swierstra, D. 1990 A meteorological model on a transputer network. In Hoffman, G.-R. and Maretis, D. K., editors, *The Dawn of Massively Parallel Processing in Meteorology*. Springer-Verlag.
- Chamberlain, R. M. and Chesshire, G. 1990 Some computational fluid dynamics applications on the Intel iPSC/2. In Hoffman, G.-R. and Maretis, D. K., editors, *The Dawn of Massively Parallel Processing in Meteorology*, 277–286. Springer-Verlag.
- Charney, J. G., Fjørtoft, R., and von Neumann, J. 1950 Numerical integration of the barotropic vorticity equation. *Tellus*, **2**(4), 237–254.
- Clarke, L. and Wilson, G. 1990 Tiny: An efficient routing harness for the INMOS Transputer. Preprint EPCC-TR90-04.

- Courtier, P and Geleyn, J.-F. 1988 A global numerical weather prediction model with variable resolution: Application to the shallow water equations. *Quarterly Journal of the Royal Meteorological Society*, 114(483), 1321-1346.
- Cybenko, G. 1989 Dynamic load balancing for distributed memory multiprocessors. *Journal of Parallel and Distributed Computing*, 7, 279 - 301.
- Dally, W. 1990 Network and processor architecture for message-driven computers. In Suaya, R. and Birtwhistle, G., editors, *VLSI and Parallel Computation*, 140-222. Morgan Kaufmann, Palo Alto, CA.
- Dent, D. 1988 The multitasking spectral model at ECMWF. In G.-R. Hoffmann and Snelling, D., editors, *Multiprocessing in Meteorological Models*. Springer-Verlag, Berlin.
- Dickinson, A. 1990 Autotasking forecasting and climate models on a Cray-YMP. In Pritchard, E. J., editor, *Science and Engineering on Supercomputers*, 35-47. Springer-Verlag, Berlin.
- Fisher, M. 1987 The Met O 20 stratosphere-mesosphere model. DCTN 52, U.K. Meteorological Office, Meteorological Office (Met. O. 20) London Road Bracknell Berkshire RG12 2SZ.
- Flynn, M. J. 1972 Some computer organisations and their effectiveness. *IEEE Transactions on Computers*, C-21, 948-960.
- Fourtney, M. 1990 Parallel processing at Cray Research, Inc. In Hoffman, G.-R. and Maretis, D. K., editors, *The Dawn of Massively Parallel Processing in Meteorology*, 140-158. Springer-Verlag.
- Fox, G., Johnson, M., Lyzenga, G., Otto, S., Salmon, J., and Walker, D. 1988 *Solving Problems on Concurrent Processors*, volume 1 General Techniques and Regular Problems. Prentice-Hall International, London.

- Gadd, A. 1978 A numerical advection scheme with small phase speed errors. *Quarterly Journal of the Royal Meteorological Society*, **104**, 583–592.
- Gadd, A. 1980 Two refinements of the split explicit integration scheme. *Quarterly Journal of the Royal Meteorological Society*, **106**, 215–220.
- Gelernter, D. 1981 A DAG-based algorithm for prevention of store-and-forward deadlock in packet networks. *IEEE Transactions on Computers*, **C-30**(10).
- Gerndt, H. M. 1989 *Automatic Parallelization for Distributed-Memory Multiprocessing Systems*. PhD thesis, Rheinischen Friedrich-Wilhelms-Universität.
- Gill, A. E. 1982 *Atmosphere–Ocean Dynamics*. Academic Press, New York.
- Girard, C. and Jarraud, M. 1982 Short and medium range forecast differences between a spectral and grid point model. an extensive quasi-operational comparison. Technical Report 32, European Centre for Medium Range Weather Forecasting, ECMWF, Shinfield Park, Reading.
- Gregory, D. and Rowntree, P. R. 1990 A mass flux convection scheme with representation of cloud ensemble characteristics and stability-dependent closure. *Monthly Weather Review*, **118**(7), 1483–1506.
- Grønas, S. 1988 Parallel integration in the Norwegian prediction model. In *Multiprocessing in Meteorological Models*, 407–418. Springer-Verlag.
- Hack, J. J. 1989 On the promise of general-purpose parallel computing. *Parallel Computing*, **10**, 261–275.
- Haltiner, G. J. and Williams, R. T. 1980 *Numerical Prediction and Dynamic Meteorology (second edition)*. John Wiley and Sons, New York.

- Hillis, D. 1984 *The Connection Machine*. M.I.T. Press, Cambridge, Mass. U.S.A.
- Hoare, C. A. R. 1985 *Communicating Sequential Processes*. Prentice Hall, London.
- Hockney, R. W. and Curington, I. J. 1989 $f_{1/2}$: A parameter to characterize memory and communication bottlenecks. *Parallel Computing*, **10**, 277–286.
- Hockney, R. W. and Jesshope, C. 1988 *Parallel Computers 2*. Adam Hilger, Bristol.
- Hoffman, G.-R. and Maretis, D. K., editors 1990 *The Dawn of Massively Parallel Processing in Meteorology*. Springer-Verlag, Berlin.
- Hoffmann, G.-R. and Snelling, D. F., editors 1988 *Multiprocessing in Meteorological Models*. Springer-Verlag, Berlin.
- Hogan, T. F. and Rosmond, T. E. 1991 A description of navy operational global atmospheric prediction system's spectral forecast model. *Monthly Weather Review*, **119**(8), 1786–1815.
- Holloway, Jr., J. L., Spelman, M. J., and Manabe, S. 1973 Latitude-longitude grid suitable for numerical time integration of a global atmospheric model. *Mon. Wea. Rev.*, **101**, 69–78.
- Holton, J. R. 1979 *An Introduction to Dynamic Meteorology*, volume 23 of *International Geophysics Series*. Academic Press, New York, second edition.
- Horiguchi, S. and Miranker, W. L. 1989 A parallel algorithm for finding the maximum value. *Parallel Computing*, **10**, 101–108.
- Hoskins, B. J. and Simmons, A. J. 1975 A multi-layer spectral model and the semi-implicit method. *Quart. J. Roy. Met. Soc.*, **101**, 637–655.
- Houghton, J. T., Jenkins, G. J., and Ephraumus, J. J. 1990 *Climate Change The IPCC Scientific Assessment*. Cambridge University Press.
- Houghton, J. T. 1986 *The Physics of Atmospheres*. Cambridge University Press, second edition.

- Houghton, J. 1991 The Bakerian lecture, 1991: The predictability of weather and climate. *Phil. Trans. R. Soc. Lond A*, **337**, 521–572.
- Ibbet, R. N. 1982 *The architecture of high performance computers*. McMillan, London.
- INMOS Limited 1988 *The Transputer Reference Manual*. Prentice Hall.
- James, I. N. and Hoskins, B. J. 1990 An overview of the UK universities' atmospheric modelling project. Technical Report 12, UK Universities' Global Atmospheric Modelling Project, Dept. of Meteorology, University of Reading, Reading RG6 2AU.
- James, I. N. and James, P. M. 1989 Ultra-low-frequency variability in a simple atmospheric circulation model. *Nature*, **342**, 53–55.
- Jones, G. 1987 *Programming in Occam*. Prentice-Hall International, London.
- Juckes, M. N. and McIntyre, M. E. 1987 A high-resolution one-layer model of breaking planetary waves in the stratosphere. *Nature*, **328**, 590–595.
- Kauranne, T. 1990 Asymptotic parallelism in weather models. In Hoffman, G.-R. and Mareis, D. K., editors, *The Dawn of Massively Parallel Processing in Meteorology*. Springer-Verlag.
- Kung, H. T. 1979 Let's design algorithms for VLSI systems. In *Proceedings of Conference on VLSI: Architecture, Design, Fabrication at Caltech*. Also technical report no CMU-CS-79-151 from Carnegie-Mellon University, Computer Science Department.
- Lazou, C. 1988 *Supercomputers and their Use*. Oxford Science Publications, Oxford.
- Lorenc, A. C., Bell, R. S., and Macpherson, B. 1991 The Meteorological Office analysis correction data assimilation scheme. *Quart. J. Roy. Met. Soc.*, **117**, 59–91.

- Lorenz, E. N. 1982 Atmospheric predictability experiments with a numerical model. *Tellus*, **34**(6), 505–513.
- Machenhauer, B. 1979 The spectral method. In *Numerical methods used in atmospheric models*, volume II of *GARP No. 17*, 121–275. ICSU/WMO.
- Machenhauer, B. and Rasmussen, E. 1972 On the integration of the spectral hydrodynamical equations by a transform method. Technical Report 3, Københavns Universitet Institut for Teoretisk Meteorologi, Available in U.K. Meteorological Library, Bracknell, U.K.
- McIntyre, M. E. 1989 On the antarctic ozone hole. *Journal of Atmospheric and Terrestrial Physics*, **51**, 29–43.
- Meiko 1991 *CS-Tools for SunOS ver. 1.10–1.14*. Meiko Scientific, 650 Aztec West, Bristol, BS12 4SD, UK. Ref. No. 83–009A00–02.02.
- Messinger, F. and Arakawa, A. 1975 *Numerical methods used in atmospheric models*, volume I of *GARP No. 17*. ICSU/WMO.
- Metcalf, M. and Reid, J. 1989 *Fortran 8x Explained*. Clarendon Press, Oxford, revised edition.
- Milner, R. 1989 *Communication and Concurrency*. Prentice Hall, London.
- Mitchel, J. F. B., Senior, C. A., and Ingram, W. J. 1989 CO₂ and climate: a missing feedback? *Nature*, **341**(6238), 132–134.
- Murphy, J. M. 1988 The impact of ensemble forecasts on predictability. *Q. J. R. Meteorol. Soc.*, **114**(480), 463–493.
- Murphy, J. M. 1990 Assessment of the practical utility of extended range ensemble forecasts. *Q. J. R. Meteorol. Soc.*, **116**(491), 89–125.

- Orzag, S. A. 1970 Transform method for the calculation of vector-coupled sums: application to the spectral form of the vorticity equation. *Journal of Atmospheric Science*, **27**, 890–895.
- Press, W. H.,
Flannery, B. P.,
Teukolsky, S. A., and
Vetterling, W. T. 1986 *Numerical Recipes: the Art of Scientific Computing*. Cambridge University Press.
- Prior, D., Radcliffe, N.,
Norman, M., and
Clarke, L. 1990 What price regularity ? *Concurrency, Practice and Experience*, **2**, 55–78.
- Richardson, L. F. 1922 *Weather Prediction by Numerical Process (1965 reprint)*. Dover, New York.
- Ritchie, H. 1987 Semi-Lagrangian advection on a Gaussian grid. *Monthly Weather Review*, **115**, 608–619.
- Robert, A. 1982 A semi-Lagrangian and semi-implicit numerical integration scheme for the primitive meteorological equations. *Journal of the Meteorological Society of Japan*, **60**, 319–325.
- Simmons, A. J. and
Burrige, D. M. 1981 An energy and angular momentum conserving vertical finite-difference scheme and hybrid vertical coordinates. *Monthly Weather Review*, **109**, 758–766.
- Simmons, A. and
Strüfing, R. 1981 An energy and angular momentum conserving finite-difference scheme, hybrid coordinates and medium-range weather prediction. Technical Report 28, European Centre for Medium Range Weather Forecasting, ECMWF, Shinfield Park, Reading.
- Simmons, A. J.,
Burrige, D. M.,
Jarraud, M.,
Girard, C., and
Wergen, W. 1989 The ECMWF medium-range prediction models: Development of the numerical formulations and the impact of increased resolution. *Meteorology and Atmospheric Physics*, **40**, 28–60.

- Smith, M. and Wilson, G. 1991 Dynamic load-balancing on a one-dimensional mesh. Tech. Report EPCC-TN91-11, Edinburgh Parallel Computing Centre, EPCC, Kings Buildings, Mayfield Rd, Edinburgh, EH9 1JT.
- Snelling, D. F. and Tanqueray, D. A. 1988 Performance modelling of the shallow water equations on the FPS-T series. In *CONPAR 88 B*, 156–159. British Computer Society - Parallel Processing Specialist Group.
- Swarztrauber, P. N. and Sato, R. 1990 Solving the shallow water equations on the Cray X-MP/48 and the Connection Machine 2. In Hoffman, G.-R. and Maretis, D. K., editors, *The Dawn of Massively Parallel Processing in Meteorology*, 260–276. Springer-Verlag.
- Temperton, C. 1983 Self-sorting mixed-radix fast fourier transforms. *Journal of Computational Physics*, **52**, 1–23.
- Thinking Machines Corp. 1991 *The Connection Machine CM-200 Series, Technical Summary*. , Cambridge, Mass, U.S.A.
- Tibaldi, S., Palmer, T. N., Branković, C., and Cubasch, U. 1990 Extended range predictions with ECMWF models: Influence of horizontal resolution of systematic errors and forecast skill. *Q. J. R. Meteorol. Soc.*, **116**(494), 835–866.
- Tracton, M. S., Mo, K., Chen, W., Kalany, E., Kistler, R., and White, G. 1989 Dynamical extended range forecasting (DERF) at the national meteorological center. *Monthly Weather Review*, **117**(7), 1604–1635.
- Trew, A. and Wilson, G., editors 1991 *Past, Present, Parallel—A Survey of Available Parallel Computing Systems*. Springer-Verlag, Berlin.
- Turing, A. M. 1936 On computable numbers, with an application to the Entscheidungs problem. *Proceedings of the London Mathematical Society*, **42**, 230–265.

- Wallace, D. J. 1988 Scientific computation on SIMD and MIMD machines. *Phil. Trans. R. Soc. Lond. A*, **326**, 481–498.
- Wallace, D. J. 1990 Supercomputing with transputers. *Computing Systems in Engineering*, **1**(1), 131–141.
- Washington, W. M. and Parkinson, C. L. 1986 *An Introduction to Three-Dimensional Climate Modelling*. University Science Books, Mill Valley, California.
- White, P. W., Cullen, M. J. P., Gadd, A. J., Flood, C. R., Palmer, T. N., Pollard, K., and Shutts, G. 1987 Advances in numerical weather prediction for aviation forecasting. *Proc. R. Soc. London A*, **410**, 255–268.
- Williamson, D. L. and Browning, G. L. 1973 Comparison of grids and difference approximations for numerical weather prediction over a sphere. *J. Appl. Meteor.*, **12**, 264–274.
- Williamson, D. L. 1979 Difference approximations for fluid flow on a sphere. In *Numerical methods used in atmospheric models*, volume II of *GARP No. 17*, 51–120. ICSU/WMO.
- Zemansky, M. W. and Dittman, R. H. 1981 *Heat and Thermodynamics*. McGraw-Hill.