A Study of the Shallow Water Equations on Various Parallel Architectures

D. Abramson †
M. Dix ‡
P. Whiting †

† Division of Information Technology
C.S.I.R.O.
55 Barry St,
Carlton, 3053

‡ Division of Atmospheric Research
C.S.I.R.O.

ABSTRACT

This paper discusses the parallelisation of the shallow water equations, which provide a simple mathematical model of the behaviour of a tank of shallow water. The paper begins with a discussion of the shallow water equations, and the common techniques for solving them. It then gives a very brief overview of the computer architectures we have considered in this study. A description of the techniques used to map the code onto these machines is given, followed by some experimental results.

1. INTRODUCTION

The application of parallel computers to Numeric Weather Prediction has attracted a great deal of attention over the past few years [1,2]. As the computational demands of realistic models have already exceeded the resources of even the very high speed uni-processors, parallelisation is seen as the only effective way to solve the models. This paper discusses the parallelisation of the shallow water equations. While such equations are much simpler than those required to solve the general weather modelling problem, they provide insight into the computational demands of large scale systems.

The paper begins with a discussion of the shallow water equations, and the common techniques for solving them. It then gives a very brief overview of the computer architectures we have considered in this study. A description of the techniques used to map the code onto these machines is given, followed by some experimental results.

2. THE SHALLOW WATER EQUATIONS

2.1 Physical interpretation

The shallow water equations describe the motion of a incompressible fluid with a free surface, with the constraint that the horizontal scales of motion are much larger than the vertical. For more details see [3,4].

The equations may be applied on any domain from a small tank to the global atmosphere or ocean provided it is modelled as a single layer. Applied to the atmosphere the shallow water equations can be taken to represent the flow averaged through the troposphere. Though obviously a great simplification of the full range of dynamics and processes of the real atmosphere (and as simulated in modern climate and weather prediction models) they are still very useful. On the largest horizontal scales atmospheric motions are approximately barotropic (i.e. don't change with height) and the shallow water equations are a reasonable approximation to the dynamics. Even simpler models (with fixed upper and lower boundaries) were used for the earliest numerical weather prediction experiments and showed some skill [5].

The shallow water equations are a favoured choice for experiments with various model structures and numerical schemes. Although a very simple representation of the atmosphere they do include the two types of horizontal wave motion important in more realistic models, gravity waves and Rossby waves. They are also useful for experiments in parallel computing because they include much of the structure (and so problems with data communication and synchronisation) of more complicated models.

2.2 Basic equations

The simplest form of the shallow water equations is

$$\frac{d\mathbf{V}}{dt} = -f\,\mathbf{k} \times \mathbf{V} - \nabla\Phi \qquad (1)$$

$$\frac{d\Phi}{dt} = -\Phi\nabla\cdot\mathbf{V} \qquad (2)$$

where V is the horizontal vector wind, Φ the geopotential height (the geometric height times the gravitational acceleration), k the vertical unit vector and f the Coriolis parameter (zero in a non-rotating frame). Expanding the total time derivatives gives

$$\frac{\partial\mathbf{V}}{\partial t} = -(\xi+f)\,\mathbf{k}\times\mathbf{V} - \nabla\left(\Phi+\frac{\mathbf{V}\cdot\mathbf{V}}{2}\right) \qquad (3)$$

$$\frac{\partial\Phi}{\partial t} = -\nabla\cdot(\Phi\mathbf{V}) \qquad (4)$$

where $\xi = \mathbf{k}\cdot\nabla\times\mathbf{V}$ is the vertical component of the vorticity.

Even for this very simple model the equations are nonlinear and cannot be solved directly, requiring the use of numerical methods. One approach is to discretise the equations on a grid, replacing the spatial derivatives by finite differences. The other main approach is a spectral method where the fields are defined as sums over a set of suitable basis functions. The derivatives are evaluated analytically from the basis functions, giving more accuracy, subject of course to the problems of truncation. Spectral methods are popular in meteorological modelling because the basis functions usually chosen, the spherical harmonics, are similar to the natural large scale modes of the atmosphere, and so can give an accurate representation of the flow with relatively few degrees of freedom.

Both finite difference and spectral models treat the time integration in the same way, replacing the time derivative by a finite difference form. Simple leapfrog time integration is usually used, in order to ensure numerical stability, though more complicated schemes are possible. Using $(\,)^n$ to denote time level n, this scheme can be written as $X^{n+1} = X^{n-1} + 2\Delta t(dX/dt)^n$ where $(dX/dt)^n$ is the time tendency of X at the central time, n.

A simpler approach using only a single time level is not possible because such schemes are unconditionally numerically unstable for wave motion [4].

2.3 Grid point model

The grid point model is based on that of Sadourny [10] and uses the shallow water equations in a slightly different form. Equation (3) is modified to

$$\frac{\partial\mathbf{V}}{\partial t} = -\eta\,\mathbf{k}\times(\Phi\mathbf{V}) - \nabla\left(\Phi+\frac{\mathbf{V}\cdot\mathbf{V}}{2}\right) \qquad (6)$$

where $\eta = (\mathbf{k}\cdot\nabla\times\mathbf{V})/\Phi$ is the potential vorticity.

This form of the equations makes it easier to design a finite difference scheme that conserves potential vorticity. This is one of the properties of the full equations which it may be desirable to preserve for reasons of accuracy and stability. The model is cast on a doubly periodic plane rather than a sphere and the Coriolis term is dropped. A model on the globe has extra problems in considering the poles. This means the calculations at each latitude are no longer the same which can complicate the extraction of parallelism.

The grid is staggered with the velocities defined on different points to the geopotential and vorticity, as shown in Figure 1. This is desirable for numerical reasons and has no effect on the problem of parallelising.

2.4 Spectral model

The spectral model of the shallow water equations is based on that described by Bourke [11]. Rather than using the components of the wind velocities directly, the model is based on vorticity (the curl of the vector wind) and divergence. The usual components of the vector wind are not suitable for a spectral expansion because they are discontinuous at the poles. The shallow water equations are based upon vorticity and divergence and take the form:

$$\frac{\partial \xi}{\partial t} = - \nabla\cdot(\xi+f)\ \mathbf{V} \qquad (7)$$

$$\frac{\partial D}{\partial t} = - \mathbf{k}\cdot\nabla\times(\xi+f)\ \mathbf{V} - \nabla^2(\Phi' + \frac{\mathbf{V}\cdot\mathbf{V}}{2}) \qquad (8)$$

$$\frac{\partial \Phi'}{\partial t} = - \nabla\cdot(\Phi'\mathbf{V}) - \Phi\bar{D} \qquad (9)$$

where $D = \nabla\cdot\mathbf{V}$ is the horizontal divergence and $\Phi = \bar{\Phi}+\Phi'$ separates the geopotential into a time-independent global mean and a time varying departure. The basic fields of the model (ξ,D,Φ) are defined by a truncated series of spherical harmonics.

3. THE ARCHITECTURES

The four architectures chosen for the study presented in this paper represent four different approaches to providing high performance computation. The first is a *conventional* pipelined supercomputer, the CRAY Y/MP. The second, an Encore Multimax, which is an example of the current generation of shared memory multiprocessors. It is worth noting that the peak performance of the Multimax (about 4 MFlops) is many times less than that of the CRAY (333 MFlops). However, it provides measurement of the relative speedups due to multiprocessing, and the absolute times should not be compared to the CRAY. The third class of machine is a distributed memory multiprocessor. We ran two different experiments under this category. The first used the Encore as a message passing machine. In this mode we could get a benchmark for the best performance of a distributed memory machine could offer, because the communication cost is almost negligible. The second involved a network of SUN workstations, connected via Ethernet. This provided us with a benchmark for a machine with poor communications performance. The final architecture used was a SIMD machine, a 1024 processor MasPar. This machine has a peak performance of 100 MFlops.

3.1 The CRAY Y/MP

The CRAY Y/MP is a member of the original CRAY-1 family of supercomputers. It provides overlapped scalar execution with a number of functional units as well as a pipelined vector unit. The scalar performance of the CRAY is not much higher than current workstations. Most of the high performance comes from the vectorisation of inner program loops. Scalar instructions are issued providing there is a free arithmetic unit of the correct type and that other shared resources are free in the correct time slot. This allows more than one scalar instruction to be executing at the same time.

Vectors are processed via a number of vector registers and special vector functional units. Vectors of up to 64 words can be pipelined through a functional unit, with an effective rate of one word per clock cycle. The compiler uses the scalar and vector registers as a program managed cache, reducing the latency which would be experienced if the data were fetched from the memory subsystem.

The CRAY style of machine has traditionally been used for executing climate modelling programs. They generally achieve very high performance because they contain many vector operations. The experiments performed on the shallow water equations were coded in both Fortran and C. The latter programs failed to vectorise for various reasons, and demonstrate the effect of scalar arithmetic on such programs. The Fortran compiler for the CRAY automatically vectorises all inner loops providing no data dependencies are present.

3.2 A Shared Memory Encore Multimax

The Encore Multimax is a typical symmetric shared memory multiprocessor. It consists of a number of processors connected to memory via a shared bus. To reduce bus and memory traffic, each processor has its own high speed cache memory, which can hold both private and shared data items. A cache coherence mechanism guarantees that a processor never receives stale data from a variable which is being manipulated by more than one processor. The details of the coherence policy are not important to this study, and can be found in [6].

The advantage of a coherent shared memory multiprocessor is that much sequential code can be converted to parallel forms with relative ease. Variables which are required by more than one processor can be placed in shared memory. Synchronisation between processes can be provided by using indivisible memory locks and barriers [7]. The main disadvantage of this class of multiprocessor is that it is very difficult to scale the machine up to large configurations. The problem is that the bus and memory subsystem must provide fast access to all processors. As the number of processors grows, the competition for these shared resources also grows, and the performance of the entire machine is degraded. The only large shared memory machines currently available, such as the BBN Butterfly [8] cannot achieve their peak performance unless the data is partitioned into local memories, even though any processor can address the memory of any other.

The parallel programs for the Encore were written in C and were processed with the Argonne Portable Programming Macros to provide access to multiple processes, shared data and to control synchronisation [9]. These macros are available for a number of multiprocessors and allow the user to write machine independent parallel code.

3.3 A Distributed Memory Machine

Unlike shared memory machines, distributed memory multiprocessors can be scaled up to very large configurations. This is because there are fewer totally shared resources in the system, and thus the configuration can grow without a single bottleneck constraining the performance. The interconnection network varies widely between manufacturers, but basically provides logical communication between any two processors. Some communications networks provide a uniform delay between processors while others favor local connections.

Distributed memory machines are usually programmed by passing messages between otherwise sequential processes. If one processor does not have a particular piece of data, then it can be passed from the processor which does have the data using send and receive primitives. These ideas are embodied in languages such as OCCAM [7]. The biggest disadvantage of distributed memory machines is that the program must not only know when to synchronise with other co-operating processes, but also which pieces of data are required for a computation. Further, a process is not only responsible for receiving data it requires, but also for sending data to other processes working on the same problem.

The experiments performed on the shallow water programs were done on two different configurations of distributed memory machine. As stated above, the first experiments were done as message passing processes, but on a shared memory machine. This allowed us to measure the performance of the programs when there was no communication delay between sending and receiving a message. The second set of experiments were done on a network of SUN workstations, using Ethernet and TCP/IP as the communication mechanism. These trials show the effect of a rather slow interconnect. Other multiprocessors will probably have communication speeds between these two extremes.

The programs were again coded in C using the Argonne macros. These only require a simple modification to move the programs from the Encore environment to the SUN configuration.

3.5 A SIMD Machine - MASPAR

SIMD machines contain many more processors than most MIMD machines, but only execute one instruction stream. The data is partitioned across the processors, and each CPU executes the same instruction at the same time on all enabled CPUs. The Maspar machine used in this study contained 1024 processors, organised as a 32 x 32 array. Each processor can communicate with any of its eight neighbours using a network called *xnet*. Communication is also allowed via a multi level cross bar router called the *Router*. The *Router* allows random global communication patterns, but is generally slower than *Xnet* communication. Each processor contains 16k bytes of local storage, and can only access memory in another processor by the *Xnet* or the *Router*.

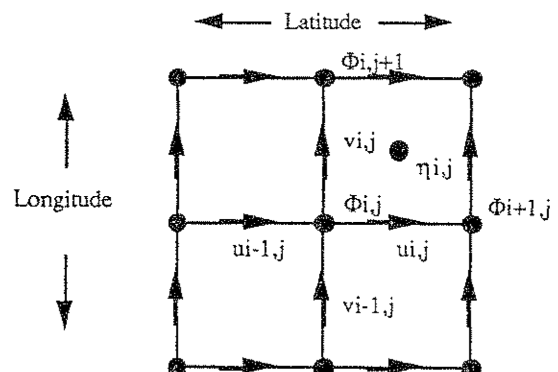
The main programming language for the Maspar is a modified form of C, called MPL. MPL has intrinsic structures for accessing the router and the xnet. It also has a new data type called plural, which duplicates variables so that a copy exists on all processors.

4. FORMULATION

4.1 Finite Difference Formulation

This section describes the formulation of the finite difference equations. Figure 1 shows the arrangement of the model variables on the staggered grid. The model grid is equally spaced and has an equal number of latitude and longitude points. The main time stepping loop of the model has three stages. (As implemented, each is a subroutine running over all latitude and longitude points.)

1) Calculate values of ΦV, η and $(\Phi + 1/2\ V.V)$ required in Equation (6). For example, $(\Phi u)_{i,j} = 1/2\ (\Phi_{i+1,j} + \Phi_{i+1,j})\ u_{i,j}$ and is defined at the same points as u. The complete definitions of these products are given by Sadourny [10]. This phase is called **calcuvzh**.
2) Apply the horizontal difference operators to calculate the time tendencies, $\partial u/\partial t$, $\partial v/\partial t$ and $\partial \Phi/\partial t$. This phase is called **timetend**.
3) Use the time tendencies to do a leapfrog time step as in (5). This phase is called **tstep**.



Section of computational grid for the finite difference model showing the arrangement of the variables.

Figure 1 - information flow between processes in finite difference scheme

4.1.1 Finite Difference on the CRAY Y/MP

Mapping the finite difference form onto the CRAY simply involves expressing the equations in terms of their grid co-ordinates, and then solving for all grid points. The Fortran Compiler automatically vectorises the inner loop (which scans all longitudes at a given latitude).

4.1.2 Finite Difference on the Shared Memory Machine

The finite difference form of the equations have very few inter-loop interaction dependencies for a given time step, so it is possible to slice the outer loop and distribute this across the processors. In this way, each processor is responsible for a number of complete latitudes. After each of the major phases described in 4.1 a barrier is used to wait for all slices to be complete. The next time step is also not started until all of the processes have completed the time step. Information can flow down the grid from one processor to another through shared memory, so that no explicit data movement between processors is required.

4.1.3 Finite Difference on a Distributed Memory Machine

The distributed memory program for the finite difference form of the equations is very similar to the shared memory program except that it is not only necessary for all processors to synchronise with each other when they have completed a computational phase, or a complete step, but they must also pass the data which is required between processors. Figure 2 shows the information flow which must occur before and after each phase.

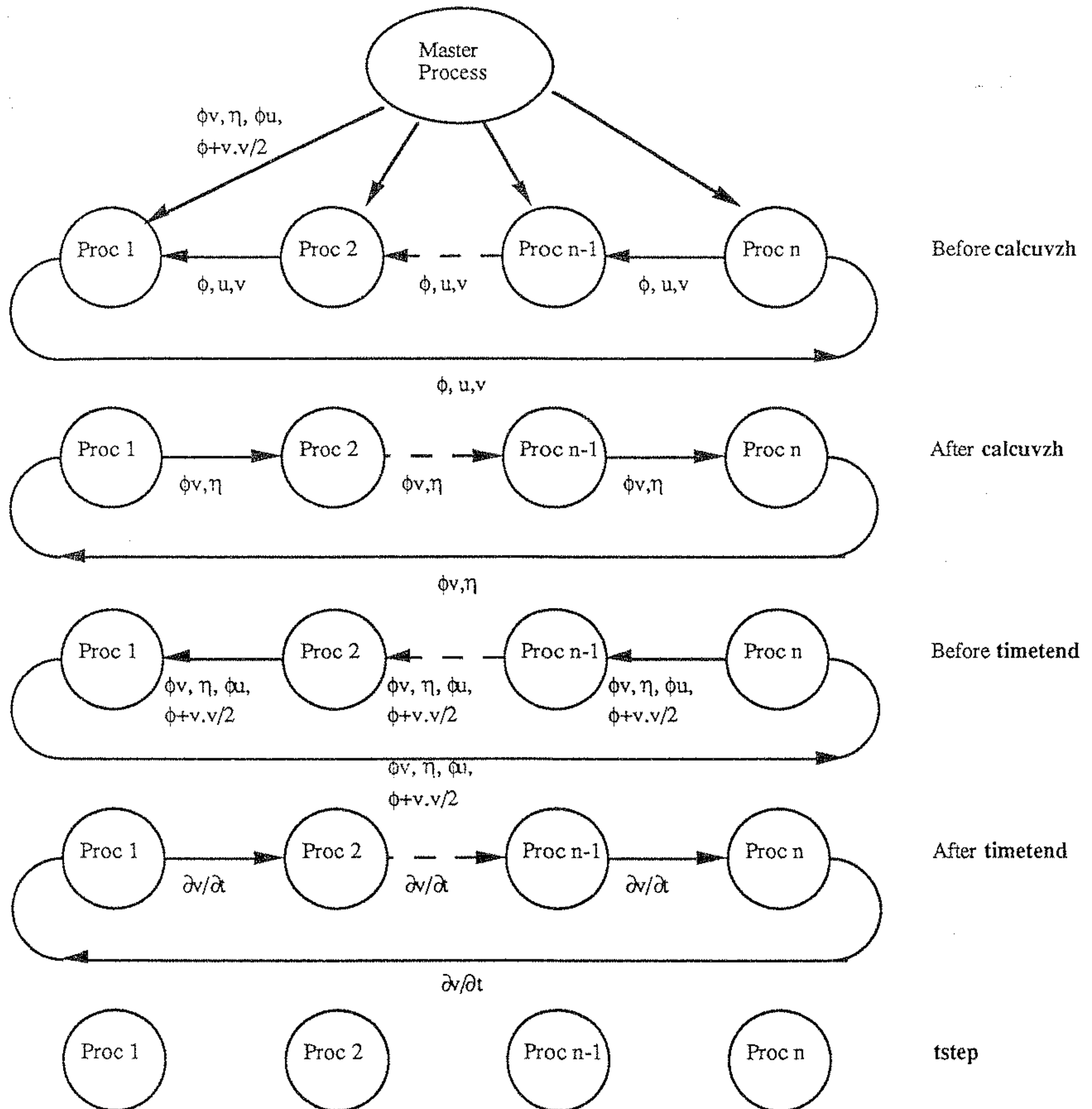


Figure 2- information flow between processes in finite difference scheme

4.1.4 Finite Difference on a SIMD

The finite difference grid is mapped exactly onto the processor array of the MasPar, and each processor is responsible for the variables at one grid cell. The loops which were required in the sequential code for covering all grid points are completely removed and all cells are computed concurrently. Communication is only required between neighbouring cells, and this is provided by the *xnet* router network.

4.2 Spectral Method Formulation

This section briefly describes the formulation of the spectral method. For more detail on the spectral transforms see Bourke [11]. The progress of the data through spectral, Fourier and grid space in the transform loop is shown in Figure 3.

The time stepping loop begins with spectral fields of ξ, D and Φ.

1) Calculate spectral fields of U (=u cosϕ) and V(=v cosϕ).
2) For each of ξ, Φ, U and V expand the sum of associated Legendre functions to give Fourier series at each latitude. (Note that a grid field of D is not needed.)
3) Apply inverse Fourier transforms to calculate grid fields of ξ, Φ, U and V.
4) Calculate the non-linear product terms.
5) Apply Fourier transforms to calculate Fourier series of the product terms at each latitude.
6) Calculate contributions to the Gaussian quadrature of the tendencies at each latitude.
7) Sum these contributions to complete the Legendre transform.
8) Add linear (i.e. calculated spectrally, not via transform process) contributions to time tendencies.
9) Use time tendencies to update spectral fields of ξ, D and Φ.

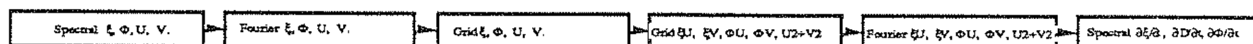


Figure 3 - progress of spectral code through spectral, fourier and grid space

4.2.1 Spectral Method on the CRAY

Like the finite difference program, the spectral program did not require any special mapping to the CRAY Y/MP. The Fortran compiler automatically vectorised the inner loops which did not have data dependencies. When in spectral space and grid space, the computations are completely data independent and thus proceed at the maximum rate. The transforms between these spaces causes data movement and produces data dependent loops.

5. RESULTS

This section contains the results of the many experiments that were conducted. Because this work is still in early stages, not all of the programs have been coded. Notably, the spectral model code has not been ported to the distributed memory machine nor the Maspar. There were also system limits which prevented running all of the configurations we would have liked. For example, some configurations of the message passing code on the SUN network caused the C compiler to fail. Other configurations required more shared memory or semaphores than configured on the machine. Where possible, a number of different sized problems were run to illustrate the effect of increased communication or computation on the results.

5.1 CRAY Results

We ran two different model sizes for each of the spectral and the finite difference programs, and used the Fortran and C languages. The Fortran compiler on the CRAY Y/MP automatically vectorised inner loops where possible. The C compiler (scc) was unable to vectorise the same program because the arrays were passed into subroutines as reference parameters, and the compiler was therefore unable to determine whether the input parameters were different arrays.

The finite difference program was run with grid sizes of 32 x 32 and 96 x 96. The efficiency of the vectorised program improved on the larger problem because the vector overhead was relatively lower than in the smaller problem. The percentage of vectorised instructions (% vectorised) was measured by dividing the number of floating point operations executed in vector mode over the total number of floating point operations. The MFlops rate for the 96 x 96 model was probably as high as one can achieve with this type of computation on a CRAY Y/MP.

The Spectral code was executed with grid sizes of 128 x 96 and 64 x 48, with corresponding resolutions of 32 and 16. The spectral model spends a substantial portion of its time moving from spectral space to grid space, and thus achieves a lower percentage of vector operations. The FFT1 version of the program used the CRAY Y/MP supplied vectorised FFTs.

Program	Problem Size	Time(secs)	MFlops (average)	% vectorised
Finite Diff C	32 x 32	6.5	10.8	0.1
Finite Diff C	96 x 96	66.61	9.4	0.1
Finite Diff Fortran	32 x 32	0.68	121.8	94.0
Finite Diff Fortran	96 x 96	4.0	180.5	99.0
Spectral Fortran	32 x 128 x 96	9.14	29.8	60.9
Spectral Fortran FFT[1]	32 x 128 x 96	3.41	64.0	91.8
Spectral Fortran	16 x 64 x 48	2.04	22.7	46.2
Spectral Fortran FFT[1]	16 x 64 x 48	0.59	53.8	92.1
Spectral C	16 x 64 x 48	4.0	9.4	0.2
Spectral C	32 x 128 x 96	24.82	8.9	0.2

5.2 Shared Memory results

The shared memory finite difference program was measured with grid sizes of 16 x 16, 32 x 32 and 96 x 96. The Spectral code was executed with grid sizes of 128 x 96 and 64 x 48, with corresponding resolutions of 32 and 16. The speedup exhibited by these programs is close to optimal.

Program	Problem Size	Parallel Times 1	2	4	8	16	Peak Speedup
Finite Difference	96 x 96	9480	4976	2407	1211	626	15.1
Finite Difference	32 x 32	690	350	177	93	53	13.0
Finite Difference	16 x 16	161	81	43	23	15	10.7
Spectral	32 x 96 x 128	2248	1127	579	291	150	14.98
Spectral	16 x 64 x 48	354	177	89	46	25	14.0

5.3 Distributed Memory results

The first set of experiments on the message passing form of the program were run on the Encore, and the message buffers were placed in shared memory. This allowed us to measure the performance of the message passing code with a zero communication cost. The spectral version of the code has not been implemented yet. The results show the loss of performance due to the message passing overheads, even though they are passed though shared memory. It is notable that for the smallest problem (16 x 16) the code executes slower with 16 processors than with 2. The largest problem (96 x 96) performs more computations between communication and therefore exhibits superior speedup.

DISTRIBUTED MEMORY RESULTS (Shared Memory)

Program	Problem Size	Parallel Times 1	2	4	8	16	Peak Speedup
Finite Difference	16 x 16	224	149	130	210	417	1.7
Finite Difference	32 x 32	774	417	270	265	443	2.9
Finite Difference	96 x 96	9229	4733	2474	1435	1046	8.8

The second set of experiments used UNIX sockets instead of shared memory, but all processes were executed on the same machine. Thus, these results show the decrease in performance due to the extra socket communication code, even though no data is transferred between processors.

As with the last set of results, the speedup is superior for a larger problem. We could not run a 96 by 96 grid due to memory constraints, nor could we run the problem on 16 processors.

DISTRIBUTED MEMORY RESULTS (Sockets)

Program	Problem Size	Parallel Times					Peak Speedup
		1	2	4	8	16	
Finite Difference	16 x 16	219	179	163	235	-	1.3
Finite Difference	32 x 32	774	457	322	308	-	2.5

The final set of experiments were conducted on a network of SUN 3/50's. In this case the message passing includes the cost of transferring the data through Ethernet from one SUN to another. They illustrate that on the small problem (16 x 16) it is not possible to achieve any speedup; in fact the programs execute slower as processors are added. We were not able to run a large problem due to a bug in the SUN C compiler. The experiments demonstrate that there must be substantial amount of computation between communication if a loosely coupled network of machines is to yield a speedup.

DISTRIBUTED MEMORY RESULTS (SUN Network)

Program	Problem Size	Parallel Times			Peak Speedup
		1	2	4	
Finite Difference	16 x 16	233	440	678	1

5.4 SIMD MasPar Results

The MasPar has the highest peak performance of any of the parallel machines we used in this study, and therefore achieved the highest performance apart from the CRAY. We only ran one grid size, which was chosen to match the size of the machine. Smaller grids would not have executed any faster, and larger grids require each processor to handle more than one grid cell. The MasPar has a peak performance of 30% of the CRAY, and the CRAY runs the problem 3.9 times faster than the MasPar.

MASPAR RESULTS

Program	Problem Size	Time (secs)
Finite Difference	32 x 32	2.7

6. PROGRAMMING CONSIDERATIONS

Each of the parallel programs produced in this study were *hand crafted* for each particular parallel architecture. In some cases we found this programming relatively easy, where as in others it was extremely difficult. The table below gives some indication of the relative complexity of the programs by comparing the number of lines of source code. We have also added a subjective measure *difficulty*, which was a rough measure of how hard we thought the task was.

Program	Lines of Code	Difficulty
Sequential Finite Difference in FORTRAN	226	-
Sequential Finite Difference in C	300	-
Parallel Finite Difference for shared memory	392	Easy
Parallel Finite Difference for distributed memory	1356	Hard
Parallel Finite Difference in MPL	170	Easy
Sequential Spectral model in FORTRAN	740	-
Sequential Spectral model in C	818	-

Parallel Spectral model for shared memory	1011	Easy

The increase in source lines experienced when the program was translated from sequential Fortran into sequential C was mainly due to the improved quality of the C program. We removed global variables in common, added subroutines, and generally tidied up the code. Each of the shared memory algorithms was fairly easy to implement, and added very little code. The distributed memory form of the finite difference program was significantly larger than the shared memory form because it required code for message passing, message packing and unpacking, and many new message data type declarations. All of the programs were measured before the Argonne macros were expanded.

The MPL program was shorter than the sequential counterpart because it did not require any loops or subroutines. In spite of the relative ease of programming this program, we would not recommend MPL be used for large programs, because it requires very detailed knowledge of the machine organisation, and many simple mistakes caused erroneous results. The program was further simplified because it mapped exactly onto the 1024 processor array.

7. CONCLUSIONS

The work described in this paper is still in early stages. We have not yet coded all forms of the spectral code, and thus do not have a complete set of results. Also, because of bugs in the SUN C compiler and in the configuration of some of our machines, we could not conduct experiments on larger systems. All of these obstacles should be removed over the coming months. We also plan to implement the distributed memory algorithms on a transputer array. This will allow us to measure the performance of the code on machines which have been explicitly designed for message passing.

Once we have completed a study of the one level atmospheric model, we will add some of the effects of multi level physics. We will also experiment with other programming systems to determine a suitable language and environment for programming parallel weather modelling programs.

Acknowledgements

The work described in this paper was conducted as a collaborative project between the CSIRO Divisions of Information Technology and Atmospheric Research. Special thanks go to Barry Hunt from Atmospheric Research for initiating the project. Also thanks go to Rhys Francis for editing a draft of this paper.

References

[1] G.-R. Hoffmann and D.F. Snelling, "Multiprocessing in Meteorological Models", Springer-Verlag, 1988.

[2] G.-R. Hoffmann and D.K. Maretis, "The Dawn of Massively Parallel Multiprocessing in Meteorology", Springer-Verlag, 1990.

[3] J. Pedlosky, "Geophysical Fluid Dynamics", Springer-Verlag, 1979.

[4] G.J. Haltiner and R.T. Williams, "Numerical Prediction and Dynamic Meteorology", Wiley, 1980.

[5] J. Charney, R. Fjortoft and J. von Neumann, "Numerical integration of the barotropic vorticity equation", Tellus, Vol 2, pp 237-254, 1950.

[6] Encore Corporation. "The Multimax Technical Summary".

[7] R.H. Perrott, "Parallel Programming",Addison Wesley Publishing Company, 1987.

[8] W. Crowther, J. Goodhue, R. Gurwitz, R. Rettburg, and R. Thomas, "The Butterfly (TM) Parallel Processor", IEEE Computer Architecture Technical Committee Newsletter, pp 18-45, Sep-Dec 1985.

[9] Lusk, Overbeek, et al, "Portable Programs for Parallel Processors", Holt, Rinehart and Winston Inc, 1987.

[10] R. Sadourny, "The dynamics of finite-difference models of the shallow water equations",J. Atmos. Sci., Vol 32, pp 680-689, 1975.

[11] W. Bourke, "An efficient, one level, primitive equation spectral model", "Mon. Wea. Rev.", Vol 100", pp 683-689, 1972.

[12] S.A. Orszag, "Transform method for the calculation of vector-coupled sums: application to the spectral form of the vorticity equation", J. Atmos. Sci., Vol 27, pp 890-895, 1970.

ABRAHAMSON

Australian Computer Science Communications

Volume 13, Number 1

February, 1991

Proceedings of the
Fourteenth Australian Computer Science Conference
6 – 8 February, 1991

Department of Computer Science,
University of New South Wales,
Kensington, NSW 2033

ACSC-14