Connection Machine Application Performance

Oliver A. McBryan

CU-CS-434-89 April 1989

Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309

CONNECTION MACHINE APPLICATION PERFORMANCE*

Oliver A. McBryan †

Dept. of Computer Science,
University of Colorado,
Boulder, CO 80309, USA.

ABSTRACT

The Connection Machine CM-2 computer represents the state of the art in supercomputer performance at this time, with peak rates of over 20 Gflops in 32-bit precision. While theoretical peak rates are essentially never attained, remarkable performance is possible on real applications. We discuss a number of CM-2 applications including implicit and explicit PDE solvers as well as spectral methods. We demonstrate delivered performance over a gigaflop in each case, and ranging as high as 3.8 gigaflops in the case of conjugate gradient solution of elliptic PDE.

We describe 2D and 3D Fast Helmholtz and Poisson Direct Solvers for the CM-2 and provide performance data for them on grids with up to 4 million points and using 65,536 processors. Performance of 1.1 Gflops is attained in 2D, and over 850 Mflops in 3D. The solution of the Helmholtz equation on a 2048×2048 grid takes under half a second, and on a 128×128×256 3D grid it requires .54 seconds. We have iteratively solved more general elliptic PDE by conjugate gradient methods at 3.8 Gflops. The fast solver has been used to provide a pre-conditioner for the conjugate gradient solver, which is then limited in performance to 1.3 Gflops, but results in far fewer iterations.

We also describe several partial and complete applications ranging from oil reservoir simulation to oceanographic modeling. In the latter case we present the first results of spectral models running on the CM-2. We emphasize the issues involved in attaining these levels of performance and compare in most cases with CRAY-XMP performance for the same algorithm. All results are for algorithms written in a high-level language, in this case *Lisp - no assembly language (PARIS) programming was performed. Similar performance should therefore be attainable by Fortran programs once the CM-2 Fortran compiler is released.

Keywords: parallel, performance, pde, spectral, poisson

---

* To appear in Proceedings of the NASA-Ames Scientific Applications of the Connection Machine Conference, Horst Simon ed., 1989.

† Research supported in part by DOE contract DE-ACO2-76ER03077, by NSF grant DMS-8619856 and by NSF Cooperative Agreement DCR-84200944.

1. INTRODUCTION

The Connection Machine CM-2 computer consists of 65,536 bit-serial processors, 2048 Weitek floating point processors, and 512 Mbytes of memory. The processors are arranged 16 to a chip, and each pair of chips shares a Weitek unit. The chips are connected in a 12-dimensional hypercube, while the processors on each chip are themselves fully connected. Most logical and integer processing is performed within processors, but floating point operations require data to be communicated externally to the off-chip Weitek processors. The system is an SIMD architecture - all processors receive the same instruction on each cycle.

The CM-2 computer has the highest peak floating point performance (24 Gflops) of any supercomputer. It is essential to stress that this is a peak performance, almost never realized in practice, although certain specialized operations such as polynomial evaluation may come close to attaining such rates. Such operations typically require no inter-processor communication, and almost no communication between the processors and the Weitek floating point units. To achieve the latter requirement one typically loads the Weitek vector registers, and then computes entirely within the Weitek for a substantial time before returning the results to the processors.

Of far greater interest is the actual performance attainable on real applications. Real applications are typified by substantial local (e.g. nearest-neighbor) or global (e.g. FFT or global vector sum) communication patterns, as well as limited ability to stay resident in Weitek processors. Many applications require significant amounts of non homogeneous computation, for example in the case of irregular grids, adaptive refinements or treatment of boundary conditions.

In this paper we describe the implementation of a number of applications on the CM-2 computer, with particular emphasis on attainable performance. In section 2 we describe a highlight - the solution of Elliptic PDE at 3.8 Gflops using conjugate gradient methods, and we indicate the nature of the internal optimization issues involved in achieving such performance. In section 3 we describe the implementation of fast Poisson solvers for rectangular grids in two and three dimensions, and we demonstrate performance of 1.1 Gflops and 860 Mflops respectively for these cases. Section 4 presents an example illustrating that the CM-2 can outperform a CRAY computer by a factor of 40 on a selected application segment - in this case part of an oil reservoir simulator. Section 5 discusses a complete application - solution of the shallow water equations - and presents implementations using both explicit and spectral methods. All of these applications use essentially standard algorithms and are mapped in a straight forward way to the CM-2.

Not all applications are well suited to the CM-2. In previous papers[1,2] we have shown that standard Multigrid algorithms do not perform well on massively parallel architectures, due to the inherent difficulty of keeping 65,536 processors busy at all times. However a resolution was obtained by discovering a new class of intrinsically parallel multiscale algorithms[2-5] which utilize all processors while converging faster than standard multigrid. Undoubtedly further intrinsically parallel algorithms remain to be discovered for procedures that do not map well to the CM-2.

Throughout the paper we quote performance in terms of megaflops, or computation time per point which is a more useful quantity in general. Measurements in tables specify the number of processors used - from 4K up to 64K. To allow relative comparison, the

performance numbers have been scaled *linearly* to an "equivalent performance" on a 64K machine. Such a scaling is not valid for some algorithms (e.g. FFT), but it does provide a uniform way to present the performance data. In the case of 64K results, the numbers are of course the real measured values.

2. A PERFORMANCE HIGHLIGHT: 3.8 GFLOPS PDE SOLUTION

Discretization of elliptic partial differential equations such as the equation

$$\vec{\nabla}\cdot\vec{k}(\vec{r})\vec{\nabla}u = f(\vec{r}) \ ,$$

by finite element or finite difference methods, leads to systems of equations with sparse coefficient matrices. The fill-in of the matrix tends to follow diagonals and the bandwidth is about $dN^{1/2}$ or $dN^{2/3}$, for two or three dimensional space respectively, where N is the dimension of the matrix and d is the degree of the finite elements used for the discretization. Furthermore, typically only $O(1)$ diagonals have nonzero elements. We have developed a parallel preconditioned conjugate gradient algorithm on the Connection Machine to solve systems of equations with such coefficient matrix structures.

The Preconditioned Conjugate Gradient Method[6-10] finds the solution of the system of equations $Ax = f$, to a specified accuracy ε by performing the following iteration on the vector x, which has been appropriately initialized:

$r = f - Ax$
$p = Br$
loop
  $s = \langle r,Br\rangle / \langle p,Ap\rangle$
  $r = r - s\cdot Ap$
  $x = x + s\cdot p$
  $rbr = \langle r,Br\rangle$
  $s = \langle r,Br\rangle / old\langle r,Br\rangle$
  $p = Br + s\cdot p$
until *converged*

Here B is an approximate inverse of A, which is assumed to be positive definite symmetric, and $\langle x,y\rangle$ denotes the inner product of vectors x and y. The *preconditioning* operator B can be effective in improving substantially the convergence rate of the algorithm[11].

We parallelize the algorithm by exploiting parallelism in every operation of the iteration. All of the vectors in the algorithm are allocated as CM parallel variables (pvars). For our Poisson-like test problem with a 5-point discretization on a rectangle, the operation $x \rightarrow Ax$ is easily written using the *NEWS* grid addressing modes of the CM. See the following sub-section for details. For simplicity we have chosen the pre-conditioning operator B to be the diagonal of the operator A. The other communication intensive operations in the conjugate gradient algorithm are the several inner products of vectors which are required. These inner products perform at very high speeds on the CM by taking advantage of the hypercube structure to

evaluate the global sum. For full details on the implementation, we refer to our paper[1].

The performance of this algorithm for a two-dimensional PDE discretized with a five-point formula on a 65,536 processor CM-2 is presented in Table 1, where we have given results for solution of equations on grids up to size 4096×4096. These measurements were made with a simple diagonal scaling pre-conditioner. As can be seen, the highest performance is attained with the largest grid size, which corresponds to the highest virtual processor ratio.

Table 1: Performance of Conjugate Gradient on CM-2

Grid Size	Mflops
512×512	1412
1024×1024	2357
2048×2048	3123
4096×4096	3812

We have also developed a fast-solver for the Poisson equation which uses an FFT algorithm developed by Thinking Machines Corporation and supplied in the CM-2 mathematics library. The fast solver runs at 1.1 Gflops, and has been used as a preconditioner for the conjugate gradient solver described above. This results in far fewer iterations on fine grids, although the overall performance of the code in terms of Mflops is then only 1.3 Gflops. However these are substantially more "useful" flops than in the diagonally preconditioned case. We describe the fast Poisson solver in section 3.

2.1. Optimization Issues

The conjugate gradient solver described above was implemented in the *Lisp programming language. *Lisp is a language at the level of Fortran, C or Lisp, far removed from the details of hypercube communication or single-bit and Weitek processors. *Lisp is an extension of Common Lisp, providing the possibility of declaring certain variables in a program to be parallel, and providing parallel operators such as multiplication that operate on such parallel variables. *Lisp is very analogous to the Fortran 8X array extensions to Fortran 77. A short review of *Lisp for numerical programming is contained in a previous paper[1].

A *Lisp program is usually no longer than an equivalent serial Fortran program, and indeed one can map each Fortran line one-to-one onto a *Lisp line. To attain the performance described above, very substantial optimization is required, for example to overlap communication and computation. Fortunately the *Lisp compiler is an excellent optimizing compiler, and we illustrate this with examples of actual code used in the conjugate gradient experiments.

For the simple case of a Poisson equation, the fundamental operation $v = Au$ takes the form (with r and s scalars):

$$v_{i,j} = s*u_{i,j} + r*(u_{i,j+1} + u_{i,j-1} + u_{i-1,j} + u_{i+1,j}) .$$

The *LISP version of a function *applya* for $v = Au$ is:

*Lisp uses !! to denote parallel objects or operations, and as a special case, !! s is a parallel

***Lisp Source for $v = Au$:**

```
(defun *applya (u v)

    (*set v  (-!! (*!! (!! s) u)
                  (*!! (!! r) (+!! (news!! u -1 0) (news!! u 1 0)
                                   (news!! u 0 -1) (news!! u 0 1)
)))))
```

replication of a scalar s. Here *news!! u dx dy* returns in each processor the value of parallel variable u at the processor dx processors away horizontally and dy away vertically. Thus $u(i,j+1)$ in Fortran would be replaced by *news!! 0 1*.

The actual assembly language (PARIS) code generated by the *Lisp compiler has the form (under version 4.3 of the CM-2 system) shown in the accompanying display.

Optimized PARIS Code:

```
(defun *applya (u v)
   (let* ((slc::stack-index *stack-index*)
          (-!!-index-2 (+ slc::stack-index 32))
          (pvar-location-u-11 (pvar-location u))
          (pvar-location-v-12 (pvar-location v)))

     (cm:get-from-west-always -!!-index-2 pvar-location-u-11 32)
     (cm:get-from-east-always *!!-constant-index4 pvar-location-u-11 32)
     (cmi::f+always -!!-index-2 *!!-constant-index4 23 8)
     (cmi::get-from-east-with-f-add-always -!!-index-2 pvar-location-u-11 23 8)

     (cmi::f-multiply-constant-3-always pvar-location-v-12 pvar-location-u-11 s 23 8)
     (cmi::f-subtract-multiply-constant-3-always pvar-location-v-12
                          pvar-location-v-12 -!!-index-2 r 23 8)

     (cm:get-from-north-always -!!-index-2 pvar-location-u-11 32)
     (cmi::f-always slc::stack-index -!!-index-2 23 8)
     (cmi::get-from-north-with-f-subtract-always pvar-location-v-12 pvar-location-u-11 23 8)

     (cm:get-from-south-always -!!-index-2 pvar-location-u-11 32)
     (cmi::float-subtract pvar-location-v-12 slc::stack-index -!!-index-2 23 8)
     (cmi::get-from-south-with-f-subtract-always pvar-location-v-12 -!!-index-2 23 8)
     )
)
```

Here the code has expanded to generate various low level instructions, with fairly recognizable functionality, including several that overlap computation and communication such as:

cmi:get-from-east-with-f-add-always. The development of an optimizing compiler of this quality addressing *communication* instructions as well as computational instructions is a major achievement of the CM-2 software system. Over a period of two years we have seen the *same* *Lisp code improve in performance from 600 Mflops to 3.8 Gflops entirely due to compiler advances.

We remark that we have recently tested the above *Lisp code on the CM-2 running the new 5.0 Software system. Performance was dramatically lower, down by a factor of 2 in some cases. On examining the generated PARIS code, the explanation appears to be that the more complex optimizations above are no longer performed. Thinking Machines assures us that these optimizations, along with others, will be restored in the 5.1 release of the system.

3. A FAST POISSON SOLVER

Many physical problems require the solution of the Helmholtz equation

$$-\Delta u + cu = f \ , \tag{1}$$

on a rectangular domain in two or three dimensions. In the case where $c = 0$, the equation reduces to Poisson's equation. One can therefore obtain a good Poisson solver for periodic data by taking c very small but non-zero. As discussed in the previous section, even in the case of more complex equations, availability of a fast Poisson solver as a pre-conditioner can greatly reduce computation time when iterative methods such as conjugate gradient are utilized.

While there are several approaches to developing a fast Poisson solver, we have based ours on the use of multi-dimensional Fast Fourier Transforms. While the basic ideas are simple, the technical issues related to the Connection Machine are non-trivial and are worth discussing.

3.1. 2D Discretization

We are required to solve (1) on a rectangular domain of size $XL \times YL$. For simplicity, we discretize the 2D equation on a rectangular $mx \times my$ grid using finite differences and periodic boundary conditions. This leads to the discrete form:

$$\left(\frac{2}{hx^2} + \frac{2}{hy^2} + c\right) u_{ix,iy} - \frac{1}{hx^2}\left(u_{ix-1,iy} + u_{ix+1,iy}\right) - \frac{1}{hy^2}\left(u_{ix,iy-1} + u_{ix,iy+1}\right) = f_{ix,iy} \ ,$$

where $hx = XL/mx$, $hy = YL/my$ and $0 \le ix < mx$, $0 \le iy < my$. The periodic boundary conditions require that:

$$u_{ix,my} = u_{ix,0} \ , \qquad u_{mx,iy} = u_{0,iy} \ .$$

To solve this equation we apply the 2D discrete Fourier Transform defined by:

$$\hat{u}_{kx,ky} = (mx \cdot my)^{-1/2} \sum_{ix=0}^{mx-1} \sum_{iy=0}^{my-1} e^{-i\,2\pi kx \cdot ix} \; e^{-i\,2\pi ky \cdot iy} \; u_{ix,iy} \tag{2}$$

where again $0 \le kx < mx$, $0 \le ky < my$. This results in the diagonalized form:

$$\hat{H}_{kx,ky} \; \hat{u}_{kx,ky} \equiv \left(-\hat{\Delta}_{kx,ky} + c\right) \hat{u}_{kx,ky} = \hat{f}_{kx,ky}$$

$$-\hat{\Delta}_{kx,ky} \equiv \frac{2}{hx^2}(1-\cos(2\pi kx/mx)) + \frac{2}{hy^2}(1-\cos(2\pi ky/my)),$$

from which we solve for $\hat{u}_{kx,ky}$ by division by $\hat{H}_{kx,ky}$. We then obtain $u_{ix,iy}$ by applying the inverse Fourier transform, which differs from (2) only in the sign of the two exponents:

$$u_{ix,iy} = (mx \cdot my)^{-1/2} \sum_{kx=0}^{mx-1} \sum_{ky=0}^{my-1} e^{i2\pi kx \cdot ix} e^{i2\pi ky \cdot iy} \hat{u}_{kx,ky} . \quad (3)$$

3.2. 2D Implementation:

The Connection Machine Scientific Math Library (CMSML) includes a routine *cfft* that implements a multi-dimensional discrete complex FFT as in (2) or (3). We have based our Poisson solver on a beta-release of this subroutine. We refer to[12, 13] for discussion of the implementation of the FFT on hypercubes in general, and to[14] for details specific to the Connection Machine.

The routine *cfft* is written in micro-code to provide high-efficiency, and as such is best regarded as a black-box by the user. However it does provide some handles into the black box in the form of input parameters that can be used to control the operation of the FFT. Since the FFT accounts for almost all of the computation time of the solver, it is essential to optimize the use of the FFT on the hardware. As we discuss below, this is far from trivial as the hardware provides multiple approaches to the use of an FFT.

3.2.1. Real vs. Complex FFT

To use *cfft*, one configures the CM-2 as a rectangular grid of $mx \times my$ virtual processors, with processor ix,iy containing the complex variable $u_{ix,iy}$. In practice one supplies the $u_{ix,iy}$ as a parallel variable u. The *cfft* routine accepts the complex parallel variable u as input and returns the complex variable $\hat{u}$ as output, with $\hat{u}_{kx,ky}$ stored in processor kx,ky.

Unfortunately only a complex FFT is supported, whereas for maximal efficiency one would prefer to use a real FFT. Real FFT algorithms reduce to complex FFT's of half-length, and therefore are about twice as fast as applying a complex FFT to real data. In order to convert a real FFT to a half-length complex FFT, it is necessary to perform operations in each dimension of the form:

$$w_{ix} = u_{ix} + iu_{mx-ix} , \quad 0 \le ix \le mx/2 . \quad (4)$$

On a distributed memory computer, we recognize such operations as involving substantial long-range communication. Great care is needed in implementing such an operation, or it will dominate the cost of the subsequent FFT. In particular, the operation needs to be micro-coded on the CM-2, which means that it will have to be done by TMC internally as micro-code compilers for the CM-2 are not distributed.

Because of the issue discussed in the preceding paragraph, we have implemented the Helmholtz solver with the capability of solving either one or two equations simultaneously. In many physical problems, one actually needs to solve more than one Helmholtz or Poisson equation at a time, for example for different components of a field. With two right-hand sides f and g, we form the complex variable $h = f + ig$ and then solve the equation $(-\Delta + c)u = h$, using *complex* FFT of full length. In the single equation case we simply take $g = 0$. Either way, solution time is the same, with the double solution case essentially as efficient as using

two real FFT algorithms. All of the times quoted below are times *per solution* in the double solution case.

3.2.2. Hypercube Addresses and Grid Orderings

The CM-2 supports two addressing modes called *NEWS* and *send* addressing. *NEWS* addressing provides an n-tuple of coordinates $n_1,n_2,\cdots$ describing the location of a virtual processor in a rectangular grid. On the other hand *send* addresses describe the location of a processor on the hypercube directly, and are composed of a physical part, describing the physical processor address, and a virtual part, which points to a location within that physical processor. The equations to be solved are grid-based, and thus the most natural addressing of CM-2 processors is the *NEWS* grid addressing provided by CM-2 software.

There are however many ways to map a rectangular grid onto a hypercube, and the CM-2 software supports two distinct mappings or *orderings* from among these. The order of a grid dimension describes how *NEWS* coordinates for that dimension are mapped onto physical processors. In *NEWS* ordering, the embedding of a virtual grid onto the physical hardware is done such that processors with adjacent *NEWS* coordinates are also neighbors in the physical grid. In *send* ordering, it is assured that if one processor has smaller *NEWS* coordinates than another, then its physical address will also be smaller than the other's. Note that, despite the terminology, these are simply different versions of *NEWS* addressing. Generally *NEWS* order is most efficient for grid operations while *send*-order is more efficient for long-range communication.

Since the FFT algorithm involves substantial long-range communication, which can be performed effectively using the underlying hypercube hardware of the CM-2 network, it is to be expected that performance may be better with *send* ordering rather than *NEWS* ordering. We have allowed a choice between *NEWS* or *send* ordering in implementing our solver. As will be seen, *send* ordering allows a substantial increase in performance over *NEWS* ordering.

3.2.3. Bit-Reversal in the FFT

Standard FFT algorithms for hypercubes involve a "bit-reversal" in the output data. This means that the basic algorithms provide the correct transformed quantities, but in an incorrect order. Typically if v is the output of the FFT algorithm applied to u, a vector of length $m = 2^b$, then we recover the Fourier transform $\hat{u}$ of u from:

$$\hat{u}_k = v_{r(m,k)} \tag{5}$$

where $r(m,k)$ is the integer obtained by regarding the integer k as a b-bit integer, and then reversing the order of the b bits. As an example, with $m = 16$ and $k = 7$, we have $b = 4$, $k = 0111$, and therefore $r(m,k) = 1110 \equiv 14$.

In a multi-dimensional FFT such as (2), such a bit-reversal will occur in *each* of the coordinate directions i.e. in kx and ky separately. It is of course a simple operation to reorder the bit-reversed quantities according to this formula on a sequential computer. However, on a massively parallel computer, an operation such as (5) involves substantial long-range communication. In fact it turns out that the reordering of the bit-reversed data can take substantially longer than computing the FFT.

There is a simple approach that allows us to avoid the overhead of bit-reversal in the case of our solver. Suppose that we were to leave the Fourier transform of u in bit-reversed form v. We could then divide v by the *bit–reversal* $B\hat{H}$ of the coefficients $\hat{H}_{kx,ky}$:

$$B\hat{H}_{kx,ky} = \hat{H}_{r(mx,kx),\, r(my,ky)} ,$$

which is properly "matched" to the bit-reversal in v. Applying the inverse FFT, introduces a second bit-reversal, and thus the final transform u is obtained in normal order. The advantage of this procedure is that the bit-reversal of the coefficients $\hat{H}_{kx,ky}$ need be performed only once, no matter how many equations are to be solved with those coefficients. In iterative procedures such as conjugate gradient solution, or in time-stepping algorithms, the effect is to reduce bit-reversal overhead to a negligible level. As with the address ordering issue, we have provided our solver with an option to use bit-reversed transforms, or to re-order these to normal ordering.

To illustrate the substantial affect that bit-reversal and *send* ordering can have on final performance, we have compiled in table 2 the effects of toggling each of these choices for a 512×1024 grid solution on an 8K CM-2.

Table 2: Effects of Bit-Reversal and Send Addressing

Machine Size	Grid Size	Bit-Reverse	Addresses	Mflops
8192	512×1024	reverse	*send*	1062
8192	512×1024	normal	*send*	625
8192	512×1024	reverse	*NEWS*	465
8192	512×1024	normal	*NEWS*	79

3.2.4. Physical Communication Channels

A final implementation issue is the question of exactly which communication "wires" or channels are assigned to each coordinate direction in the CM-2. Connection Machine software provides complete freedom of choice to the user in this respect.

The Poisson solver algorithm is running on a virtual machine configured as an $mx \times my$ grid of processors. Let bx and by be the number of bits required to represent mx and my. Then the virtual machine is a $b = bx+by$ dimensional hypercube, and each processor then has b virtual channels connecting it to neighboring processors. Let P be the actual number of physical processors, and let $b_{phys} = \log_2 P$. Then b_{phys} of the b virtual channels are actual physical channels.

There are two different kinds of physical channel: on-chip and off-chip. An on-chip channel connects to a neighboring processor on the same chip. An off-chip channel connects to a processor on another chip. Since each CM-2 chip contains 16 processors, it follows that there are exactly 4 on-chip physical channels, and $b_{phys} - 4$ off-chip physical channels. The remaining $v = b - b_{phys}$ virtual channels are true virtual channels, and involve communication only between virtual processors assigned to the same physical processor.

In configuring the CM-2 as a rectangular grid, the various channels described above are separately assigned to the x and y directions. Thus in general we will have onx physical on-bit channels in the x direction and ony physical on-bit channels in the y direction. Similarly we will have $offx$ physical off-chip channels in the x direction and $offy$ off-chip channels in the y direction. Finally there will be vx virtual channels in the x direction and vy virtual channels

in the y direction. Obviously these quantities are not all independent, but instead satisfy the constraints:

$$\begin{aligned} onx + ony &= 4 \ , \\ offx + offy &= b_{phys} - 4 \ , \\ onx + offx + vx &= bx \ , \\ ony + offy + vy &= by \ . \end{aligned}$$

The *create–detailed–geometry* CM-2 library routine allows the CM-2 user to specify the quantities *onx*, *offx*, *ony* and *offy* when configuring the CM-2 as a rectangular grid. The particular choices made can have a major effect on the FFT performance as we will see below.

A few observations can be helpful in determining the best choice of channel assignments. It is frequently advantageous to have all of the on-chip channels, and one off-chip channel, in the x direction. This corresponds to the fact that 32 processors share a single Weitek processor with fast access to it. It is also advantageous to concentrate all virtual channels into one direction, and to have the off-chip channels in a direction be a multiple of four. As will be clear from Tables 3-5 below, these are guidelines and not rules.

3.2.5. The *cfft* Routine

Fortunately the CM-2 *cfft* routine allows the user full control over all of the features described in previous sub-sections. The calling sequence is essentially:

cfft(*u*,*v*,*geometry*,*operations*,*in_orders*,*out_orders*,*in_address*,*out_address*,*scales*)

Here u and v are the complex variable and its transform, and *geometry* is a data-structure that summarizes the grid rank, dimensions, and the on-chip and off-chip assignments for each dimension. Typically the *geometry* object will have been created previously by a call to *create–detailed–geometry*. The *geometry* object also involves some pre-computed data structures to speed the FFT computation, but we will not dwell on this aspect. The *operations* parameter is an array of operations to be performed, one for each dimension. Possible operations are *fft*, *inverse_fft* or *no–operation*. The *in_orders* parameter is an array specifying the ordering status (*normal* or *bit–reversed*) of the incoming data u in each of the coordinate directions. Similarly for *out_order* with respect to the output data v. The *in_address* array specifies the addressing order - *NEWS* or *send* in use in each dimension for the incoming data, and similarly for *out_address*. Finally *scales* is an array specifying a choice of scaling in each dimension - either full scaling, square-root scaling or no scaling.

3.2.6. 2D Results

In implementing the Poisson solver we have considered all possible choices of the various parameters to *cfft*. For each grid size $mx \times my$ and physical machine size P, we have performed *bitsweep* operations in which we ran the solver with every possible assignment of the *on* and *off*-chip bits. Essentially this amounts to providing different *geometry* arguments to *cfft*.

We have also allowed the possibility of using either *NEWS* or *send* ordering modes for the computation. We have provided a choice of using or omitting an explicit bit-reversal after each FFT. It is obviously advantageous to scale once on the inverse transform, rather than scale by the square root (as in (2) and (3)) in both directions. Performance depends heavily on

all of these choices.

By far the most significant issue is bit-reversal. Omitting the explicit bit-reversal speeds the solver by a factor of three. Use of *send* ordering speeds the solver about a further 20%. Finally by varying the choice of *on* and *off* bits, we can affect performance by about 30%.

In table 3 we present the optimal performance attained across a range of machine and grid sizes. The optimal performance is defined as the maximum from varying all of the appropriate parameters. Since in each case the optimum occurs with bit-reversal, and *send* ordering, we have supplied only the on and off chip bit data in the table. We provide both the megaflops rating and the more meaningful computation time per grid-point per solution.

To give a feeling for the potential variability as a function of physical channel assignments, we provide in Table 4, all of the data used to determine the last line in Table 3 - the case of a 2048×2048 grid on 64K processors.

3.3. 3D Discretization

We are required to solve (1) on a rectangular domain of size $XL \times YL \times ZL$. For simplicity, we discretize the 3D equation on a rectangular $mx \times my \times mz$ grid using finite differences and periodic boundary conditions. This leads to the discrete form:

$$\left(\frac{2}{hx^2} + \frac{2}{hy^2} + \frac{2}{hz^2} + c\right) u_{ix,iy,iz} - \frac{1}{hx^2}\left(u_{ix-1,iy,iz} + u_{ix+1,iy,iz}\right)$$
$$- \frac{1}{hy^2}\left(u_{ix,iy-1,iz} + u_{ix,iy+1,iz}\right) - \frac{1}{hz^2}\left(u_{ix,iy,iz-1} + u_{ix,iy,iz+1}\right) = f_{ix,iy,iz} \ ,$$

where $hx = XL/mx$, $hy = YL/my$, $hz = ZL/mz$ and $0 \le ix < mx$, $0 \le iy < my$, $0 \le iz \le mz$. The periodic boundary conditions require that u satisfy:

$$u_{mx,iy,iz} = u_{0,iy,iz} \ , \quad u_{ix,my,iz} = u_{ix,0,iz} \ , \quad u_{ix,iy,mz} = u_{ix,iy,0} \ .$$

To solve this equation we apply the 3D discrete Fourier Transform defined by:

$$\hat{u}_{kx,ky,kz} = (mx \cdot my \cdot mz)^{-1/2} \sum_{ix=0}^{mx-1} \sum_{iy=0}^{my-1} \sum_{iz=0}^{mz-1} e^{-i\,2\pi kx \cdot ix} \; e^{-i\,2\pi ky \cdot iy} \; e^{-i\,2\pi kz \cdot iz} \; u_{ix,iy,iz} \ ,$$

where again $0 \le kx < mx$, $0 \le ky < my$, $0 \le kz \le mz$. This results in the diagonalized form:

$$\hat{H}_{kx,ky,kz} \; \hat{u}_{kx,ky,kz} \equiv (-\hat{\Delta}_{kx,ky,kz} + c) \; \hat{u}_{kx,ky,kz} = \hat{f}_{kx,ky,kz}$$
$$-\hat{\Delta}_{kx,ky,kz} \equiv \frac{2}{hx^2}(1 - \cos(2\pi kx/mx)) + \frac{2}{hy^2}(1 - \cos(2\pi ky/my)) + \frac{2}{hz^2}(1 - \cos(2\pi kz/mz)) \ ,$$

from which we solve for $\hat{u}_{kx,ky,kz}$ by division by $\hat{H}_{kx,ky,kz}$. We then obtain $u_{ix,iy,iz}$ by applying the inverse Fourier transform, which differs from (2) only in the sign of the three exponents:

$$u_{ix,iy,iz} = (mx \cdot my \cdot mz)^{-1/2} \sum_{kx=0}^{mx-1} \sum_{ky=0}^{my-1} \sum_{kz=0}^{mz-1} e^{i\,2\pi kx \cdot ix} \; e^{i\,2\pi ky \cdot iy} \; e^{i\,2\pi kz \cdot iz} \; \hat{u}_{kx,ky,kz} \ .$$

The discussion now parallels exactly that for the 2D case. Again we are faced with the issues of addressing order (*NEWS* or *send*), bit-reversal and choice of physical assignment channels. The most efficient FFT procedure, as represented by the *cfft* library routine, is again obtained when the output is bit-reversed in each coordinate direction relative to the input. Therefore we choose to bit-reverse the coefficients $\hat{H}_{kx,ky,kz}$ once, and avoid thereby the need to perform two explicit bit-reversals in the middle of each solution. After performing the FFT of f we divide

Table 3: Optimal Performance of 2D Helmholtz Solver							
Machine Size	Grid Size	onx	offx	ony	offy	Mflops	Time per Point
4K	64×64	4	2	0	6	495	1.253e-07
4K	64×128	4	1	0	7	729	9.190e-08
4K	128×128	4	1	0	7	908	7.931e-08
4K	128×256	3	1	1	7	845	9.116e-08
4K	256×256	0	8	4	0	898	9.129e-08
4K	256×512	3	1	1	7	902	9.648e-08
4K	512×512	0	8	4	0	939	9.799e-08
8K	64×128	1	5	3	4	517	1.296e-07
8K	128×128	2	5	2	4	612	1.177e-07
8K	128×256	4	1	0	8	1025	7.511e-08
8K	256×256	4	1	0	8	1066	7.695e-08
8K	256×512	3	1	1	8	994	8.749e-08
8K	512×512	0	9	4	0	1020	9.023e-08
8K	512×1024	0	9	4	0	1060	9.153e-08
16K	128×128	3	4	1	6	543	1.327e-07
16K	128×256	2	5	2	5	613	1.256e-07
16K	256×256	4	2	0	8	674	1.217e-07
16K	256×512	4	1	0	9	947	9.191e-08
16K	512×512	4	1	0	9	998	9.220e-08
16K	512×1024	4	1	0	9	972	9.980e-08
16K	1024×1024	0	10	4	0	1028	9.925e-08
32K	128×256	4	3	0	8	579	1.330e-07
32K	256×256	1	7	3	4	683	1.200e-07
32K	256×512	4	4	0	7	709	1.227e-07
32K	512×512	4	5	0	6	792	1.162e-07
32K	512×1024	4	5	0	6	868	1.118e-07
32K	1024×1024	4	1	0	10	1015	1.005e-07
32K	1024×2048	3	1	1	10	1056	1.014e-07
64K	256×256	3	5	1	7	625	1.313e-07
64K	256×512	4	4	0	8	669	1.300e-07
64K	512×512	4	5	0	7	775	1.187e-07
64K	512×1024	4	5	0	7	821	1.181e-07
64K	1024×1024	4	6	0	6	793	1.287e-07
64K	1024×2048	4	1	0	11	1041	1.028e-07
64K	2048×2048	4	1	0	11	1097	1.021e-07

it by $B\hat{H}_{kx,ky,kz}$, where

$$B\hat{H}_{kx,ky,kz} = \hat{H}_{r(mx,kx),\, r(my,ky),\, r(mz,kz)},$$

and $r(m,k)$ is the integer bit-reversal function defined earlier.

Table 4: Performance as a Function of Channel Assignments							
Machine Size	Grid Size	onx	offx	ony	offy	Mflops	Time/Point
64K	2048×2048	4	6	0	6	597	1.877e-07
64K	2048×2048	0	7	4	5	656	1.707e-07
64K	2048×2048	2	3	2	9	709	1.579e-07
64K	2048×2048	4	2	0	10	724	1.547e-07
64K	2048×2048	1	10	3	2	738	1.518e-07
64K	2048×2048	2	6	2	6	769	1.456e-07
64K	2048×2048	0	11	4	1	771	1.453e-07
64K	2048×2048	2	4	2	8	776	1.444e-07
64K	2048×2048	4	3	0	9	776	1.444e-07
64K	2048×2048	3	3	1	9	778	1.440e-07
64K	2048×2048	3	7	1	5	779	1.438e-07
64K	2048×2048	0	10	4	2	781	1.433e-07
64K	2048×2048	3	2	1	10	782	1.432e-07
64K	2048×2048	2	7	2	5	784	1.428e-07
64K	2048×2048	1	7	3	5	790	1.417e-07
64K	2048×2048	0	6	4	6	791	1.416e-07
64K	2048×2048	3	8	1	4	791	1.416e-07
64K	2048×2048	1	6	3	6	792	1.415e-07
64K	2048×2048	3	6	1	6	793	1.412e-07
64K	2048×2048	4	4	0	8	821	1.365e-07
64K	2048×2048	3	4	1	8	835	1.341e-07
64K	2048×2048	1	4	3	8	846	1.324e-07
64K	2048×2048	2	9	2	3	853	1.313e-07
64K	2048×2048	0	9	4	3	855	1.309e-07
64K	2048×2048	1	5	3	7	856	1.309e-07
64K	2048×2048	4	5	0	7	857	1.307e-07
64K	2048×2048	1	9	3	3	861	1.302e-07
64K	2048×2048	3	5	1	7	861	1.301e-07
64K	2048×2048	0	5	4	7	862	1.300-07
64K	2048×2048	2	5	2	7	863	1.298e-07
64K	2048×2048	1	8	3	4	864	1.296e-07
64K	2048×2048	2	8	2	4	865	1.295e-07
64K	2048×2048	4	7	0	5	867	1.292e-07
64K	2048×2048	0	8	4	4	868	1.290e-07
64K	2048×2048	4	1	0	11	1097	1.021e-07

3.3.1. 3D Performance

We have tested the performance of the 3D solver by making sweeps over all possible physical channel bit assignments, choosing the optimal one found in each case. Results of these optimization computations are shown in Table 5. Note that for example it was necessary to solve 335 2048×2048 Helmholtz problems in order to construct just the last line of the table. Because of the enormous effort involved, we were unable to complete sufficient runs to find the optimal values for all grids and machine sizes. It would be very desirable if Thinking Machines would provide a routine to correctly define the optimum in each case - perhaps by table look-up from data generated by a range of experiments such as these.

Table 5: Optimal Performance of 3D Helmholtz Solver

P	Grid Size	onx	offx	ony	offy	onz	offz	Mflops	Time/Pt
8K	16×16×32	3	1	1	3	0	5	493	1.360e-07
8K	16×32×32	4	0	0	5	0	4	654	1.101e-07
8K	32×32×32	4	1	0	4	0	4	755	1.020e-07
8K	32×32×64	4	1	0	4	0	4	790	1.038e-07
8K	32×64×64	4	1	0	4	0	4	823	1.057e-07
8K	64×64×64	0	5	1	4	3	0	758	1.213e-07
8K	64×64×128	4	2	0	0	0	7	805	1.205e-07
16K	64×128×128	0	6	1	4	3	0	785	1.299e-07
32K	128×128×128	0	7	0	4	4	0	812	1.318e-07
64K	128×128×256	0	1	0	7	4	4	863	1.298e-07

4. AN EXAMPLE FROM RESERVOIR SIMULATION

We have ported substantial segments of an oil reservoir simulator to the Connection Machine. In particular the slowest part, the solver for a set of non-symmetric linear equations, is currently running at 7 times its performance on a CRAY-XMP[15]. Remarkably higher speed-ups over a CRAY are found in other parts of the code, and we illustrate this with the following example of a code fragment from a simulator:

```
      do 100 j = 1,nc
        do 970 kz = 1,nz
          do 971 jy = 1,ny
            do 972 ix = 1,nx
              tden(ix,jy,kz,j) = avmw(ix,jy,kz,j)/gvolph(ix,jy,kz,j)
              dumr(ix,jy,kz) = grk(ix,jy,kz,j)/
     &                  gvis(ix,jy,kz,j)/gvolph(ix,jy,kz,j)
  972         continue
  971       continue
  970     continue
```

In table 6 below we present the performance of this code segment on a CRAY 2 processor, and on the CM-2.

Table 6: Performance on Reservoir Simulator Fragment

Machine	Processors	nx	ny	nz	nc	**Performance**
CRAY 2	1	16	16	32	2	18 Mflops
CM-2	65536	64	64	64	2	741 Mflops

The CRAY-2 performance is better than the CRAY-XMP performance and the grid sizes shown for the CRAY-2 represent the limits of the capability of the XMP to handle the overall problem in memory. Part of the CRAY problem is related to the compiler which does not unroll loops of this complexity. As a result the CRAY is dealing with nested loops, each of short vector length. However even with a manually unrolled loop, CRAY performance remains over 10 times worse than the CM-2. In addition, the CRAY treats a division as four floating point operations, although we have counted it as one in the table below. One should of course observe that it is easy to create code fragments which behave well on a CRAY, but poorly on a CM-2 - for example a piece of scalar code. What makes the example here interesting is that the code in question is typical of code segments found in highly vectorized algorithms.

5. ATMOSPHERIC/OCEANOGRAPHIC SIMULATION

As another example of the current capabilities of massively parallel architectures, we describe the implementation of a standard two-dimensional atmospheric model - the Shallow Water Equations - on the Connection Machine. These equations provide a primitive but useful model of the dynamics of the atmosphere or of certain ocean systems. Because the model is simple, yet captures features typical of more complex codes, the model is frequently used in the atmospheric sciences community to benchmark computers[16]. Furthermore, the model has been extensively analyzed mathematically and numerically[17, 18]. We have recently implemented the shallow water equations model on the Connection Machine, in collaboration with R. Sato and P. Swarztrauber of the National Center for Atmospheric Research in Boulder, and compared the performance on the CM-2 with the CRAY-XMP. We have used both explicit[2] and spectral[19] solution methods for the equations.

The shallow water equations, without a Coriolis force term, take the form

$$\frac{\partial u}{\partial t} - \zeta v + \frac{\partial H}{\partial x} = 0 ,$$

$$\frac{\partial v}{\partial t} - \zeta u + \frac{\partial H}{\partial y} = 0 ,$$

$$\frac{\partial P}{\partial t} + \frac{\partial Pu}{\partial x} + \frac{\partial Pv}{\partial y} = 0 ,$$

where u and v are the velocity components in the x and y directions, P is pressure, ζ is the vorticity: $\zeta = \frac{\partial v}{\partial x} - \frac{\partial u}{\partial y}$ and H, related to the height field, is given by: $H = P + (u^2 + v^2)/2$. It is required to solve these equations in a rectangle $a \le x \le b$, $c \le y \le d$. Periodic boundary conditions are imposed on u, v, and P, each of which satisfies $f(x+b,y) = f(x+a,y)$, $f(x,y+d) = f(x,y+c)$.

A scaling of the equations results in a slightly simpler format. Introduce mass fluxes $U=Pu$ and $V=Pv$ and the potential velocity $Z=\zeta/P$, in terms of which the equations reduce to:

$$\frac{\partial u}{\partial t} - ZV + \frac{\partial H}{\partial x} = 0 ,$$

$$\frac{\partial v}{\partial t} + ZU + \frac{\partial H}{\partial y} = 0 ,$$

$$\frac{\partial P}{\partial t} + \frac{\partial U}{\partial x} + \frac{\partial V}{\partial y} = 0 .$$

5.1. Discretization - Explicit Case

We have discretized the above equations on a rectangular staggered grid with periodic boundary conditions. The variables P and H have integer subscripts, Z has half-integer subscripts, U has integer and half-integer subscripts, and V has half-integer and integer subscripts respectively.

Initial conditions are chosen to satisfy $\vec{\nabla}\cdot\vec{v} = 0$ at all times. We time difference using the Leap-frog method. We then apply a time filter to avoid weak instabilities inherent in the leap-

frog scheme:

$$F^{(n)} = f^{(n)} + \alpha\,(f^{(n+1)} - 2f^{(n)} + f^{(n-1)})\,,$$

where α is a filtering parameter. The filtered values of the variables at the previous time-step are used in computing new values at the next time-step. For a complete description of the discretization we refer to[16].

5.2. Discretization - Spectral Case

In the case of the spectral version, the physical variables u, v, h, are discrete Fourier transformed to frequency space at the start of each time-step, and all spatial derivatives are computed in that space. An inverse Fourier transform is applied at each time-step to recover the derivatives in grid space in order to compute the time derivatives and the propagation.

One significant issue arises in the spectral discretization. The arrays to be transformed are all real arrays and for optimal efficiency a real FFT should be used instead of a complex FFT. The real FFT is effectively a complex FFT of half the length and will therefore run over twice as fast. Thinking Machines Corporation has developed a complex FFT, routine, but not a real version. Furthermore the basic algorithms to convert a real FFT to a complex one of half-length involve substantial long-range communication. An efficient version of this stage would need to be micro-coded, otherwise it would likely dominate the FFT routine which is already micro-coded. Rather than get involved with micro-code we used a different strategy.

The basic idea is to do more than one real FFT at a time. We have already encountered this situation in discussing the fast Poisson Solver, section 3.2.1, and the treatment here will be similar. A pair of real FFT's may be done as a single complex FFT of the same length. For example, we need to transform the real functions u, v in order to compute the derivatives du/dx, dv/dx, du/dy, and dv/dy. In frequency space, this amounts to multiplying the Fourier transform of u and v by matrices *dxk* and *dyl* and then inverse transforming:

```
cu   = fft(u)
cdxu = dxk*cu
dudx = inv_fft(cdxu)

cdyu = dyl*cu
dudy = inv_fft(cdyu)

cv   = fft(v)
cdxv = dxk*cv
dvdx = inv_fft(cdxv)

cdyv = dyl*cv
dvdy = inv_fft(cdyv)
```

The more efficient route is to perform the computations of du/dx and dv/dx simultaneously as follows:

```
cuv = fft(u + iv)
cdxuv = dxk*cuv
dudx + idvdx = inv_fft(cdxuv)
cdyuv = dyl*cuv
dudy + idvdy = inv_fft(cdyuv)
```

The first version requires 6 real FFTs, while the second version uses 3 complex FFT of the same length. Thus we obtain the speedup associated with real FFT but without incurring the difficulties with long-range communication we alluded to earlier.

A variant can be used if only one function was involved:

```
cu = fft(u+i0)
cu = (dxk + idyl)*cu
dudx + idvdx = inv_fft(cu)
```

The first FFT is still a "real FFT", which will be performed inefficiently as a complex FFT, but the second FFT is a full complex transform. Thus this trick allows us to do three real FFT's by using two complex ones instead. There is a potential further improvement of 33% that could be achieved here by doing the first FFT as a *real* real FFT, but at least it is not a factor of 2. In our case there are three functions and six derivatives to be computed. In the Connection Machine case we have accomplished this with 5 complex FFT, as against a slightly more efficient 9 real FFT used in the CRAY-XMP algorithm. Note that Mflops rates are insensitive to whether we use a real FFT or a full complex FFT. In the latter case the computation takes twice as long, but does twice as much computation, so the Mflops appear to be the same. However obviously the flops used in evaluating a real transform using a full complex transform are not efficiently utilized, and are only half as valuable as equivalent flops used in a true real FFT. A much better measure of useful work is the total computation time, or equivalently, the computation time per grid-point per timestep. We present both measures in the result table. While Mflops are not affected by the pairing of real transforms described above, the time per grid-point is reduced by a factor of 9/5.

Other issues discussed in detail in section 3, also reoccur in implementing the spectral model. These include the advantage of using *send* ordering rather than *NEWS* ordering, the importance of avoiding an explicit bit-reversal, and finally the essential importance of an optimal choice of assignments to specific physical hypercube channels in each coordinate direction. The bit-reversal issue is best handled by storing and bit-reversing the coefficients of the derivative matrices *dxk* and *dyl* during initialization. These bit-reversed matrices may then be used at every timestep.

5.3. CRAY Fortran Code for Explicit Algorithm

It is interesting to compare the actual code for the CRAY and CM-2 implementations in the explicit case. The Fortran code implementing the above explicit algorithm involves a 2D rectangular grid with variables: $u(i,j)$, $v(i,j)$, $p(i,j)$, $z(i,j)$, $psi(i,j)$, $h(i,j)$. There are three main loops, two corresponding to the leap-frog time propagation of various quantities, and one for the filtering step. A typical code sequence, used in the updating of the U, V and P variables, is:

```
do 200 j=1,n
do 200 i=1,m
  unew(i+1,j) = uold(i+1,j)+
              tdts8*(z(i+1,j+1)+z(i+1,j))*(cv(i+1,j+1)+cv(i,j+1)+cv(i,j)
              +cv(i+1,j))-tdtsdx*(h(i+1,j)-h(i,j))
  vnew(i,j+1) = vold(i,j+1)-tdts8*(z(i+1,j+1)+z(i,j+1))
              *(cu(i+1,j+1)+cu(i,j+1)+cu(i,j)+cu(i+1,j))
              -tdtsdy*(h(i,j+1)-h(i,j))
  pnew(i,j) = pold(i,j)-tdtsdx*(cu(i+1,j)-cu(i,j))
              -tdtsdy*(cv(i,j+1)-cv(i,j))
200 continue
```

Each such loop is followed by code to implement the periodic boundary conditions. Note that there are loops for both the horizontal and vertical boundaries, and in addition some corner values are copied as single items.

5.4. CM-2 Code for Explicit Algorithm

For the Connection Machine implementation, we specify that the the machine is to be organized as a 2D rectangular grid of virtual processors, with one virtual processor (vp) per grid point i,j. The grid variables u, v, p, z, h are then allocated as *Pvars* - parallel variables. The connection machine software automatically stores them according to the specified grid format.

The Connection Machine code corresponding to the main double loop given previously in Fortran, is actually simpler than on the CRAY. To begin with, the loops disappear from the code. This is because all global do loops are replaced by parallel operations. A second simplification is that relative, rather than absolute, addressing is provided for. Because of the local nature of the discretization equations, such relative addressing is far more convenient. The form of the CM code is then:

```
unew = uold + tdts8*(east(z) + z)*(east(cv) + south(east(cv)) + south(cv) + cv) - tdtsdx*(h - south(h))

vnew = vold - tdts8*(north(z) + z)*(north(cu) + cu + west(cu) + north(west(cu))) - tdtsdy*(h - west(h))

pnew = pold - tdtsdx*(north(cu) - cu) - tdtsdy*(east(cv) - cv)
```

Here *north*, *south*, *east* and *west* are specific relative addressing modes understood by the CM software when dealing with rectangular grids. Note that no explicit communication routines are evident, such as one would usually see in corresponding code for other hypercube processors. We have implemented the shallow water equations in two CM languages - C^* and **LISP*. The C^* code is essentially exactly as above, whereas the **LISP* version differs in employing reverse polish notation in writing the expressions.

All boundary loops are replaced by parallel operations with processor *selection*. Basically we create boolean grid variables which record which processors lie on each of the four boundaries of the rectangle. These operations will in fact be particularly slow, since they are effectively gather-scatter operations - data must be fetched from points at the opposite side of

the grid, and are consequently remotely and irregularly located within the hypercube.

There is an essential simplification that occurs in the case that the grid dimensions are both powers of two. On a hypercube, power-of-two grids *are* periodic. Thus in such cases the code for boundary copying may simply be omitted, and in fact we modified the translated program to detect such cases automatically. This is in fact the *only* change we made in translating the original Fortran program to *LISP, apart from the introduction of parallel variables and operators, and the coding of loops using processor selection.

5.5. Performance Results: CRAY-XMP/48 vs. CM-2

We have tested both the explicit and spectral models on a CRAY-XMP* and on the CM-2. In each case we have solved the largest grid size that would fit in memory.

The CRAY-XMP4/8 performed the explicit computations at 560 Mflops with 4 processors on a 512×512 grid. The corresponding performance on a single processor was 148 Mflops.

The CRAY spectral code performed poorly on grids that were a multiple of 64 due to memory bank conflicts and consequently we tried various grid sizes. The 500×500 performance of 122 Mflops was the best observed from among a large range of power-of-two and other grids.

The CM-2 performed the explicit model at 1,714 Mflops with 65,536 processors on a 2048×2048 power of two grid. In addition to being over three times faster than the four-processor CRAY, the CM-2 can clearly handle much larger problems in memory than can the CRAY. Solving a 2048×2048 grid problem on the CRAY would necessitate recoding the whole problem to use a solid state disk (SSD), and would result in lower performance.

The CM-2 has a severe disadvantage on the explicit code when processing a grid that is not a power of two in each direction. For such grids, CM performance dropped by a factor of 2.4 due to boundary effects. The essential problem here is the non-local nature of the periodic boundary condition, implying the need for long-range communication. The presence of Neumann or Dirichlet boundary conditions on a non power of two grid would have much less serious results since only local communication is involved.

Maximum spectral performance for the CM-2 was 1,167 Mflops, again on a 2048×2048 grid. This was accomplished using *send* ordering of the *NEWS* grid, no explicit bit-reversal of output from FFT's, and by performing a sweep of all possible physical channel assignments to locate an optimal choice. We have covered the latter issue in great detail in section 3.

In table 7 we list the results of representative performance of both the explicit and spectral codes on the CRAY-XMP and CM-2 computers. A four processor result for the XMP spectral code is not yet available as some significant work is required to multi-task the specialized assembly language FFT routines used on the XMP.

A comment is in order about the use of Mflops in the comparison. The Mflops being performed are essentially equivalent in "usefulness" between the CRAY and the CM-2, but not between explicit and spectral methods. This was measured by computing the actual processing time per grid-point, which is in fact a better measure of performance than Mflops. This quantity behaved in the same way as the Mflops, apart from an expected logarithmic term due to the inherent logarithmic time dependence of the FFT.

* All CRAY-XMP measurements were performed by R. Sato of NCAR.

Table 7: Performance of Shallow Water Equations				
Machine	**Processors**	**Algorithm**	**Grid Size**	**Performance**
CRAY-XMP	1	Explicit	256×256	148 Mflops
CRAY-XMP	4	Explicit	512×512	560 Mflops
CM-2	65536	Explicit	2048×2048	1714 Mflops
CRAY-XMP	1	Spectral	500×500	122 Mflops
CM-2	65536	Spectral	2048×2048	1167 Mflops

ACKNOWLEDGEMENTS

Most of the development work for the computations described here was performed on the CM-2 computer at the Center for Applied Parallel Processing (CAPP), University of Colorado, which was provided to CAPP by the Defense Advanced Research Projects Agency. All of the large grid computations (over 500,000 points) were performed on the CM-2 computer at the Los Alamos National Laboratory. We would also to thank Thinking Machines Corporation for providing a beta release of their FFT software, and R. Krawitz of TMC for helpful discussions.

References

1. O. McBryan, "The Connection Machine: PDE Solution on 65536 Processors," *Parallel Computing*, vol. 9, pp. 1-24, North-Holland, 1988.
2. O. McBryan, "New Architectures: Performance Highlights and New Algorithms," *Parallel Computing*, vol. 7, pp. 477-499, North-Holland, 1988.
3. P. O. Frederickson and O. McBryan, "Parallel Superconvergent Multigrid," in *Multigrid Methods: Theory, Applications and Supercomputing*, ed. S. McCormick, Math Applications Series, vol. 110, pp. 195-210, Marcel-Dekker Inc., New York, 1988.
4. P. O. Frederickson and O. McBryan, "Superconvergent Multigrid Methods," Cornell Theory Center Preprint, May 1987.
5. P. Frederickson, "Totally Parallel Multilevel Algorithms," RIACS Technical Report 88.34, Nov, 1988.
6. C. Lanczos, "An Iteration Method for the Solution of the Eigenvalue Problem of Linear Differential and Integral Operators," *J. Res. Nat. Bur. Standards*, vol. 45, pp. 255-282, 1950.
7. M. R. Hestenes and E. Stiefel, "Methods of conjugate gradients for solving linear systems," *J. Res. Nat. Bur. Standards*, vol. 49, pp. 409-436, 1952.
8. J. K. Reid, "On the method of Conjugate Gradients for the Solution of Large Sparse Systems of Linear Equations," in *Large Sparse Sets of Linear Equations*, ed. J. K. Reid, pp. 231-54, Academic Press, New York, 1971.

9. P. Concus, G. H. Golub, and D. P. O'Leary, ''A Generalized Conjugate Gradient Method for the Numerical Solution of of Elliptic Partial Differential Equations,'' in *Sparse Matrix Computations*, ed. D. J. Rose, Academic Press, New York, 1976.

10. G. H. Golub and C. F. Van Loan, *Matrix Computations,* John Hopkins Press, Baltimore, 1984.

11. M. Engeli, Th. Ginsburg, H. Rutishauser, and E. Stiefel, *Refined Iterative Methods for Computation of the Solution and the Eigenvalues of Self-Adjoint Boundary Value Problems,* Birkhauser Verlag, Basel/Stuttgart, 1959.

12. L. Johnsson, C.-T. Ho, Michel Jacquemin, and Alan Ruttenberg, ''Computing Fast Fourier Transforms on Boolean Cubes and Related Networks,'' *Advanced Algorithms and Architectures for Signal Processing II*, vol. 826, pp. 223-231, Society of Photo-Optical Instrumentation Engineers, 1987.

13. P. Swarztrauber and R. Sweet, ''Vector and Parallel Methods for the Direct Solution of Poisson's Equation,'' *J. Computational and Applied Mathematics*, 1989, to appear.

14. L. Johnsson, R. Krawitz, and R. Frye, ''Computing radix-2 FFT on the Connection Machine,'' Technical Report, Thinking Machines Corp..

15. O. McBryan and L. Padmanabhan, ''Conjugate Residuals on the Connection Machine - Application to Oil Reservoir Simulation,'' in *Proceedings of Copper Mountain Multigrid Conference*, ed. S. McCormick, 1989, to appear.

16. G.-R. Hoffman, P.N. Swarztrauber, and R.A. Sweet, ''Aspects of using multiprocessors for meteorological modeling,'' in *Multiprocessing in Meteorological Models*, ed. D. Snelling, pp. 126-195, Springer-Verlag, Berlin, 1988.

17. R. Sadourny, ''The dynamics of finite difference models of the shallow water equations,'' *JAS*, vol. 32, pp. 680-689, 1975.

18. G.L. Browning and H.-O. Kreiss, ''Reduced systems for the shallow water equations,'' *JAS*, to appear.

19. O. McBryan, ''Connection Machine Application Performance,'' in *Scientific Applications of the Connection Machine, Proceedings of the NASA-Ames Conference on Massively Parallel Computing*, ed. Horst Simon, 1989.