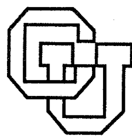
Performance of the Shallow Water Equations on the CM-200 and CM-5 Parallel Supercomputers

Oliver A. McBryan

CU-CS-634-92 December 1992



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

PERFORMANCE OF THE SHALLOW WATER EQUATIONS ON THE CM-200 and CM-5 PARALLEL SUPERCOMPUTERS*

Oliver A. McBryan†

Department of Computer Science
University of Colorado
Boulder, CO 80309

Abstract:

We describe the implementation of a fluid dynamical benchmark on two Thinking Machines Corporation parallel computers - the 65,536 processor CM-200 computer and the 1024-node CM-5 computer. The benchmark, the Shallow Water Equations, is frequently used as a model for both oceanographic and atmospheric circulation. We describe the steps involved in implementing the algorithm on the computers and we provide details of resulting performance.

We have measured 5.2 Gflops (64-bit arithmetic) and 8.1 Gflops (32-bit) on the CM-200 while the CM-5 delivers 22.1 Gflops (64-bit) and 24 Gflops (32-bit). For comparison, performance of 1.53 Gflops was measured for the same algorithm on the CRAY Y-MP/8, 1.28 Gflops on the 256-node SUPRENUM-1 and 0.54 Gflops was measured on the 128-node Intel iPSC/860.

---

* To appear in Proceedings of the Fifth Workshop of the European Centre for Medium-Range Weather Forecasts on "Uses of Parallel Processors in Meteorology", Nov. 23-27, 1992.

† Research supported by the Air Force Office of Scientific Research, under grant AFOSR-89-0422 and by NSF Grand Challenge Applications Group grant ASC-9217394.

1. INTRODUCTION

The Shallow Water Equations are a standard model for atmospheric and oceanographic processes. Implementations of the algorithm have been used as benchmarks for vector and parallel supercomputer performance for many years [1-5,8].

The Shallow Water code is very memory intensive, involving 14 variables per grid point, and accesses these using nine-point stencils, non-linear expressions and essential divisions. The combined effect provides a decidedly non-trivial test of any computer system. We have recently implemented the benchmark on the CM-200 and CM-5 supercomputers and report on the results in this paper.

The tests were run on a CM-200 at Thinking Machines Corporation and on the CM-5 at Los Alamos National Laboratory. Measured CM-200 performance was 5.25 Gflops (64-bit) and 8.09 Gflops (32-bit). Measured CM-5 performance was 22.1 Gflops for 64-bit and 24 Gflops for 32-bit computations. Numerical results agreed to high precision with those from other machines. We expect that even higher per-node CM-5 performance could be achieved by utilizing explicit optimizations. In fact the current TMC compiler, CMF 2.0, does a poor job of optimizing certain types of memory access. The worst case is the *cshift* operator which makes a complete copy of its array argument even if only local memory accesses are used. A new compiler release, CMF 2.1, will apparently overcome these inefficiencies.

We compare the performance announced here with the CRAY X-MP/4 which solves the equations at a rate of 560 Mflops on 4 processors, and with the CRAY Y-MP/8 where 1530 Mflops is attained, as well as with the performance on several of the other leading parallel systems.

2. THE CM-200 and CM-5 SUPERCOMPUTERS

The CM-200 SIMD computer is a scaled up version of the original CM-2 computer. As with the CM-2, the CM-200 features 64K bit-serial processors, and 2K floating point Weitek vector processors, interconnected to form a hypercube. There are 16 bit-serial processors on a chip, and pairs of chips share a vector processor. In typical use, 32 processors sharing a Weitek each send one or more floating point numbers to the Weitek, which returns a result operand to each processor. Because operands are supplied in bit-serial fashion by the processors but are expected in standard floating point format by the Weitek, a transpose of the incoming 32-bit wide bit-stream is required before data is provided to the Weitek. This is accomplished by a special chip (Sprint chip), without user intervention, although it does significantly slow performance. As shown here the key to good CM-200 (or CM-2) performance is to leave transposed vectors in the vector register block of the Weitek for as long as possible.

The CM-5 computer is a MIMD parallel computer featuring up to 1024 nodes and a fat-tree interconnection network. Each node consists of a Sparc scalar processor, 4 DASH vector units (each with a peak rate of 32 Mflops) and 32 MBytes of shared memory.

The CM-200 software used here consists primarily of the CMF Fortran compiler, which supports the array extensions of Fortran 90, plus several compiler directives that control CM-200 and CM-5 specific actions such as data placement and layout. In the case of the CM-200 there are two alternate modes of programming known as the Paris and Slicewise models respectively. The Paris model, supported by the compiler, focuses

on the 64K bit-serial processors (or their representation as an even larger number of virtual processors). All data is stored on these processors and is sent to the vector nodes for immediate return of operands as needed. In the Slicewise model the system is regarded as an assembly of 2048 vector nodes, each of which has a set of vector registers, and a large more expensive main memory storage (the memory associated with the bit-serial processors). The compiler recognizes the Slicewise model by generating code that attempts to maximize use of the vector registers, returning results to main memory only when absolutely essential. The distinction between the Paris and Slicewise models is effected by use of a compiler switch.

The CM-5 environment used here is also based on the use of the CMF compiler. Thus the CM-5 is regarded as a SIMD architecture and in that case any valid CM-200 program will run unchanged on the CM-5. There is no concept of Paris model for the CM-5 - the standard model corresponds to the Slicewise model for the CM-200.

We have also implemented the Shallow Water Equations using the MIMD CMMD message passing environment. Unfortunately it is not yet possible to use the vector nodes when using CMMD because node vector compilers for Fortran (F77 or F90) have not yet been released. Therefore CMMD programs can currently use only the scalar Sparc processors located at each node. Resulting performance is not reported on here since it is not a realistic estimate of CM-5 capabilities.

3. THE SHALLOW WATER EQUATIONS BENCHMARK

As an example of the current capabilities of the CM systems, we describe the implementation of a standard two-dimensional atmospheric model - the Shallow Water Equations - on the machines. These equations provide a primitive but useful model of the dynamics of the atmosphere. 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 [1-2]. Furthermore, the model has been extensively analyzed mathematically and numerically [6-7].

The Shallow Water Equations, without a Coriolis force term, take the form:

$$\partial u / \partial t - \varsigma v + \partial H / \partial x = 0,$$
$$\partial v / \partial t - \varsigma u + \partial H / \partial y = 0,$$
$$\partial P / \partial t + \partial Pu / \partial x + \partial Pv / \partial y = 0,$$

where u and v are the velocity components in the x and y directions, P is pressure, ς is the vorticity: $\varsigma = \partial v / \partial x - \partial u / \partial y$, 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,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 $U = Pu, V = Pv$ and the potential velocity $Z = \varsigma / P$, in terms of which the equations reduce to:

$$\frac{\partial U}{\partial t} - ZV + \frac{\partial H}{\partial x} = 0,$$
$$\frac{\partial V}{\partial t} + ZU + \frac{\partial H}{\partial y} = 0,$$
$$\frac{\partial P}{\partial t} + \frac{\partial U}{\partial x} + \frac{\partial V}{\partial y} = 0.$$

4. DISCRETIZATION

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 $\nabla \cdot v = 0$ 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 [1].

5. SERIAL FORTRAN IMPLEMENTATION

The Fortran code implementing the above 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. Execution of these three loops completes a single time step, which is then repeated until the desired temporal simulation interval has been achieved. A typical code sequence, used in the updating of the U, V and P variables, is:

```
do 10 j = 1,My
do 10 i = 1,Mx
  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))
10 continue
```

Each such loop is followed by code to implement the periodic boundary conditions. In the above case, the corresponding boundary code takes the form:

$do\ 20\ j = 1, n$

$unew(1, j) = unew(m+1, j)$

$vnew(m+1, j+1) = vnew(1, j+1)$

$pnew(m+1, j) = pnew(1, j)$

$20\ continue$

Note that there are such loops for both the horizontal and vertical boundaries, and in addition some corner values are copied as single items.

Excluding the boundary computations, the three major loops in a time step involve 65 arithmetic operations per grid point. Furthermore 14 physical variables must be stored per grid point, which significantly limits the largest grid size that can be accommodated in a single node.

6. CM IMPLEMENTATION

Since the algorithm involves rectangular grid arrays, and a nine-point stencil, the parallelization of the code is straightforward on the CM-200 or CM-5 under CMF. All arrays are declared as CMF arrays of the appropriate size without reference to the number of processors actually in use. The compiler automatically assigns arrays to processors in an efficient away, preserving locality of rectangular array elements. The primary code modification required is to rewrite all arithmetic on array elements using F90 array operations. Most importantly, all indexing offsets are replaced by *cshift* array operations. For example we replace $u(i, j+1)$ by $cshift(u, 2, 1)$.

Normally periodic boundary conditions require copying data between processors at opposite edges of the processor array - which typically involves long-range communication In the case that a grid dimension is a power of two, the periodic boundary condition in the corresponding dimension may be implemented by nearest neighbor communication due to the fact that the CM systems are logical torii. The experiments reported here all use this feature.

On the CM-200 we have used three strategies for parallelizing the code, which we refer to as using Paris, Slice and SerialSlice models. On the CM-5, the Paris model does not make sense so we used only the Slice and SerialSlice models. We now explain the ideas behind these approaches. In order to illustrate what the code looks like for these three approaches we will work with the much simpler example of a relaxation or averaging process:

$$v_{i,j} = u_{i,j-1} + u_{i,j+1} + u_{i-1,j} + u_{i+1,j},$$

and for each model we will show the corresponding code that implements the model. The above operation is to be executed at every point of an $Mx \times My$ grid, with periodic continuation at the boundary.

Paris Model

The Paris model represents the code in the obvious way as a F90 program which is then compiled using the -paris compiler option. In this model each arithmetic operand is fetched from CM memory, and transposed in the Sprint chip before partaking in an operation. Immediately after the operation the result is re-transposed and transmitted to CM memory. There is consequently significant internal data traffic and delay between bit-serial and vector processors. Until the advent of the Slicewise compiler this was the

only way to program the CM-2 and CM-200 computers. For the grid-average example the appropriate F90 code would be:

$$real, array(Mx,My) :: u, v$$
$$v = cshift(u,1,-1) + cshift(u,1,1) + cshift(u,2,-1) + cshift(u,2,1).$$

The resulting program would be compiled with the -paris compiler switch.

Slice Model

The Slice model modifies the code slightly to recognize the need for vector register allocation. Specifically we declare variables to store various communicated quantities - for example a variable *uwest* might be used to store the values of u to the west, as returned by $uwest = cshift(u,1,-1)$, and similarly for other quantities and directions. Thus we first prefetch the needed quantities by assignment to these "register variables". The compiler then (hopefully) assigns the quantities to vector registers allowing a block of F90 arithmetic-only code to execute unimpeded by the need to get involved in data transpositions between vector and bit-serial format (as occurs automatically for the Paris case). To accomplish this the code must be compiled with the -slicewise switch. There is a limitation to the effectiveness of the Slice model related to the number of available vector registers. For our simplified example the resulting code would be:

$$real, array(Mx,My) :: u, v, unorth, usouth, ueast, uwest$$
$$unorth = cshift(u,1,1)$$
$$usouth = cshift(u,1,-1)$$
$$ueast = cshift(u,2,1)$$
$$uwest = cshift(u,2,-1)$$
$$v = unorth + usouth + ueast + uwest.$$

The prefetching aspect is not essential here - in principal the compiler could do this. However early releases of the slicewise compiler were not so sophisticated and as a result we hand-code the prefetches as described above. For the example, the explicit prefetching highlights the slicewise idea - to get as many variables into registers before beginning numeric computations with them. Ideally we would now perform a series of arithmetic operations that reuse these variables several times before the next communication operation. In this respect the grid averaging example is too simple.

SerialSlice Model

The idea behind the SerialSlice model is that the communication inherent in the Shallow Water Equations should only occur on the boundary of rectangular grids. In both the Paris and Slice models, all points are treated equally and because the CM-200 is a SIMD system, communication is actually performed at every grid point. To overcome this we introduce an explicit decomposition of the grid into rectangular subgrids as one might do on a MIMD messaging passing system. Specifically, let the global periodic grid for the Shallow Water Equations be an $Mx \times My$ grid, and assume that we subdivide it into a $Px \times Py$ array of subgrids, each of size $Nx \times N$. We can then perform the required Shallow Water Equation computation by looping over the grid points of each subgrid. Additionally on the boundary of each subgrid we must execute *cshift* communication operations to obtain data from the neighboring processors.

A standard organization is to provide an extra boundary row and column surrounding each subgrid, into which the neighbor boundary data are copied before each loop. Parallelism may now be exploited either by parallelizing the loop within each subgrid or by parallelizing across the subgrids - executing operations at all corresponding grid-points of subgrids simultaneously. We will choose this latter route, which is therefore called SerialSlice because it is Serial within subgrids but parallel (and using the slicewise model) across subgrids. Because the computation is serial within subgrids, the required communication operations can be executed only on the boundary points.

A convenient way to parallelize across subgrids is to introduce two extra indices for each array. The first two indices are then used to represent location within a subgrid while the other two are used to represent the 2D array of subgrids. The intent to process the first two dimensions serially may then be signaled to the compiler by a CMF layout directive:

```
      real u(0:Nx+1,0:Ny+1,Px,Py)
cmf$layout   u(:serial,:serial,:news,:news),
```

which specifies that rectangular grid parallelism (:news) be used in the second pair of dimensions. All loops now remain serial loops in the first two variables, and simply represent the parallelism in the other two dimensions using the ":" notation of F90:

```
      do 20  i = 1,Nx
      do 20  j = 1,Ny
        unew(i,j,:,:) = uold(i,j,:,:) + ...
20    continue
```

Returning to our example, the code would now take the form:

```
      real, array(0:Nx+1,0:Ny+1,Px,Py) :: u,v
cmf$layout   u(:serial,:serial,:news,:news), v(:serial,:serial,:news,:news)

      do j = 1,Ny
        u(0,j,:,:) = cshift(u(Nx,j,:,:),1,1)
        u(Nx+1,j,:,:) = cshift(u(1,j,:,:),1,-1)
      do i = 1,Nx
        u(i,0,:,:) = cshift(u(i,Ny,:,:),2,1)
        u(i,Ny+1,:,:) = cshift(u(i,1,:,:),2,-1)

      do i = 1,Nx
      do j = 1,Ny
        v(i,j,:,:) = u(i,j-1,:,:) + u(i,j+1,:,:) + u(i-1,j,:,:) + u(i+1,j,:,:).
```

Once again all communication has been completed before we begin the arithmetic loops (remember that the first two indices do not involve communication). Furthermore

communication now occurs only on the external boundary of each subgrid, whereas in the Slice model the shift operations ocurred at all grid points.

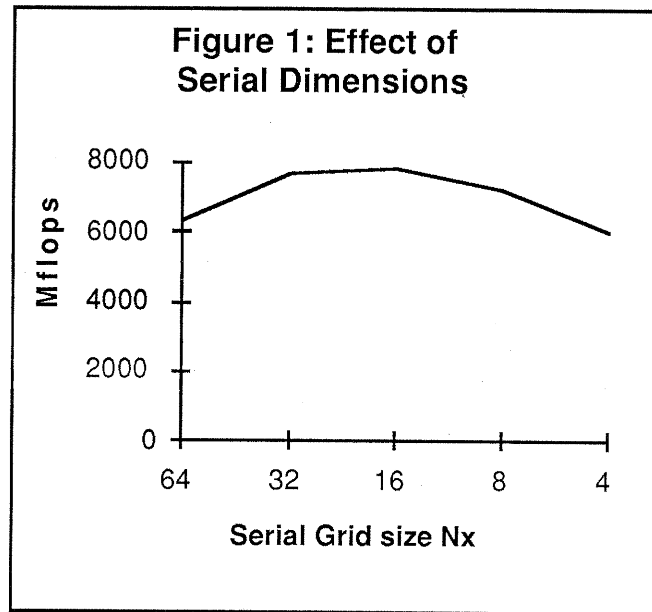
7. PERFORMANCE RESULTS: CM-200

CM-200 performance is presented in Table 1. The measurements were made on systems of 16,384 processors, but all mflops results in the table have been scaled to a system of full size (65,536 processors). The algorithm described here should, and does, scale linearly on the CM-200, as we have checked carefully up to 65,536 processors. This assumes that the scaling is done in such a way that the number of grid points per processor is unchanged. Typically we take the grid size to be the largest that will fit in memory, since this ensures the highest performance. Substantially smaller grid problems are better solved on systems with fewer processors. These comments apply equally to the CM-5 computations. The table includes a column called Vpr which is the *virtual processor ratio*. This quanity is the number of grid-points per CM-200 bit-serial processor, or the number of grid-points per CM-5 vector processor respectively.

Table 1: CM-200 Performance

Method	Mx	My	Bits	Nodes	Vpr	Mflops
Serial Slice	4096	8192	32	16384	2048	8086
Slice	4096	4096	32	16384	1024	6001
Paris	4096	4096	32	16384	1024	3598
Serial Slice	4096	4096	64	16384	1024	5249
Slice	2048	4096	64	16384	512	3546
Paris	2048	4096	64	16384	512	1834

The most striking feature of these results is the importance of using the SerialSlice model. Indeed this method is seen to be about 2.5 times faster than the Paris model for both 32-bit and 64-bit results, and is 30% to 50% more efficient than using the Slice strategy. As discussed earlier, the SerialSlice model introduces an additional parameterization of the computation in terms of the grid size of the serial grid. Performance depends strongly on the dimensions of that grid. As the serial grid increases in size, its perimeter becomes smaller in relation to its area, resulting in increased communication efficiency. On the other hand one does not want the serial grid to become too large because then the available parallelism across distinct subgrids will be reduced to the point that the vector nodes can no longer operate at peak performance. We have studied this effect in order to find the optimal serial grid size and we present the results for a typical case in the graph below.



The results for SerialSlice reported in Table 1 are the best results observed for any subgrid decomposition of the global grid. Thus they correspond to the peak point on the curve in Figure 1.

8. PERFORMANCE RESULTS: CM-5

CM-5 systems with vector nodes became available only in the last month. We have made preliminary measurements using both 64 node and 1024 node systems. Table 2 presents the results, from the 1024 node system, including both 32-bit and 64-bit floating point arithmetic.

Table 2: CM-5 Performance

Mx	My	Bits	Nodes	Vpr	Mflops
4096	4096	32	1024	4096	17526
16384	16384	32	1024	65536	23971
4096	4096	64	1024	4096	17164
8192	8192	64	1024	16384	22139

As in the CM-200 case, we have performed several hundred measurements using the SerialSlice strategy. In contrast to the CM-200, we found that in general SerialSlice is slower than Slice. The explanation for this proves to be related to the CMF 2.0 compiler. The compiler does not realize that the sequential loop within subgrids should be executed within the processors - instead it actually executes it sequentially on the

partition manager (front end system). This problem will be alleviated in the CMF 2.1 compiler.

The table presents measurements for the largest grid that would fit in the system, but also a second measurement for a grid 16 times smaller. This indicates that good performance is still obtained on problems that are more ideally suited to a smaller system.

9. A COMPARISON OF CRAY, CM-200, CM-5, iPSC/860 AND SUPRENUM-1

We have compared the CM-200 and CM-5 performance with that on the CRAY X-MP and Y-MP computers, on the Intel iPSC/860 hypercube and on the SUPRENUM-1 computer. Results are presented in Table 3. The performance on a single processor of a CRAY-X-MP was 148 Mflops. The CRAY X-MP4/8 executed the Shallow Water Equations at 560 Mflops using 4 processors on a 512^2 grid, the largest that could be handled directly (i.e. without SSD coding). The CRAY-Y-MP with 8 processors runs the Shallow Water Equations at 1,530 Mflops on a 512^2 grid. The iPSC/860 performance was 543 Mflops on 128 nodes using the largest grid size that would fit in memory. Finally SUPRENUM-1 performance of 1280 Mflops was measured on a 256-node machine, again using the largest grid possible. The rationale for using such large grids is that the benchmark is a guide to behavior of realistic 3D applications where such grid sizes would be quite realistic. From the performance viewpoint, it is essential to use a maximal grid size per processor in order to minimize the interprocessor communication overheads. The CRAY measurements were made by Dr. R. Sato of the National Center for Atmospheric Research. The iPSC/860 and SUPRENUM results are described in more detail in [5,10].

TABLE 3: COMPARISON OF ARCHITECTURES

Machine	Processors	Grid Size	Mflops
CM-5 (32-bit)	1024	256M	23971
CM-5 (64-bit)	1024	64M	22139
CM-200 (32-bit)	2048	32M	8086
CM-200 (64-bit)	2048	16M	5249
CRAY Y-MP	8	256K	1530
CRAY X-MP	4	256K	560
SUPRENUM-1	256	8M	1280
Intel iPSC/860	128	2M	543

ACKNOWLEDGMENTS

We would like to thank Thinking Machines Corporation and Los Alamos National Laboratory for providing access to the CM-5 systems used here. We would like to thank the GMD for providing access to the SUPRENUM-1. We would like to thank NASA-Ames Laboratory for providing access to the Intel iPSC/860. Finally we thank. Dr. R. Sato of NCAR for providing CRAY X-MP and Y-MP measurements.

REFERENCES

1. G.-R. Hoffmann, 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.

2. O. McBryan, "New Architectures: Performance Highlights and New Algorithms," Parallel Computing, vol. 7, pp. 477-499, North-Holland, 1988.

3. O. McBryan and R. Pozo, "Performance Evaluation of the Myrias SPS-2 Computer," CS Dept Technical Report CU-CS-505-90 (to appear in Concurrency: Practice and Experience), University of Colorado, Boulder, 1990.

4. O. McBryan and R. Pozo, "Performance Evaluation of the Evans and Sutherland ES-1 Computer," CS Dept Technical Report CU-CS-506-90, University of Colorado, Boulder, 1990.

5. O. McBryan, "A Comparison of the Intel iPSC860 and SUPRENUM-1 Parallel Computers," University of Colorado Tech. Report CU-CS-499-90 and Supercomputer, vol. 41, no. 1, pp. 6-17, 1991.

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

7. G.L. Browning and H.-O. Kreiss, "Reduced systems for the shallow water equations," JAS, vol. 44, 1987.

8. R. Pozo, "Performance Modeling of Parallel Architectures for Scientific Computing," PhD Thesis, Department of Computer Science, University of Colorado at Boulder, 1991.

9. O. McBryan and E. Van de Velde, Hypercube Algorithms and Implementations, SIAM J. Sci. Stat. Comput., 8, pp. 227-287, 1987.

10. O. McBryan, "Software Issues at the User Interface," in Frontiers of Supercomputing II: A National Reassessment, ed. W.L. Thompson, University of Colorado CS Dept. Tech Report CU-CS-527-91 and MIT Press, 1992, to appear.

ANY OPINIONS, FINDINGS, AND CONCLUSIONS OR RECOMMENDATIONS EXPRESSED IN THIS PUBLICATION ARE THOSE OF THE AUTHOR(S) AND DO NOT NECESSARILY REFLECT THE VIEWS OF THE AGENCIES NAMED IN THE ACKNOWLEDGMENTS SECTION.