LA-UR -86-4218

CONF-8609173--9

Los Alamos National Laboratory is operated by the University of California for the United States Department of Energy under contract W-7405-ENG-36

TITLE NUMERICAL COMPUTATION ON MASSIVELY PARALLEL HYPERCUBES

LA-UR--86-4218

DE87 003740

AUTHOR(S) Oliver A. McBryan

SUBMITTED TO Proceedings of the Second Conference on Hypercube Multiprocessors
Knoxville, Tennessee
October 1, 1986

DISCLAIMER

Los Alamos National Laboratory
Los Alamos, New Mexico 87545

FORM NO. 836 R4
ST. NO. 2629 5/81

Numerical Computation on Massively Parallel Hypercubes[1]

Oliver A. McBryan[2, 3]

C-3, MS-B265,
Los Alamos National Laboratory,[4]
Los Alamos, NM 87545.

ABSTRACT

We describe numerical computations on the Connection Machine, a massively parallel hypercube architecture with 65,536 single-bit processors and 32 Mbytes of memory. A parallel extension of COMMON LISP, provides access to the processors and network. The rich software environment is further enhanced by a powerful virtual processor capability, which extends the degree of fine-grained parallelism beyond 1,000,000.

We briefly describe the hardware and indicate the principal features of the parallel programming environment. We then present implementations of SOR, multigrid and pre-conditioned conjugate gradient algorithms for solving partial differential equations on the Connection Machine. Despite the lack of floating point hardware, computation rates above 100 megaflops have been achieved in PDE solution. Virtual processors prove to be a real advantage, easing the effort of software development while *improving* system performance significantly. The software development effort is also facilitated by the fact that hypercube communications prove to be fast and essentially independent of distance.

1. For presentation to the 2nd conference on Hypercube Multiprocessors, Knoxville, Oct 1, 1986.
2. Supported in part by DOE contract DE-AC02-76ER03077.
3. Supported in part by NSF grant DMS-83-12229.
4. Permanent address: Courant Institute of Mathematical Sciences, New York University, New York, N.Y. 10012.

Numerical Computation on Massively Parallel Hypercubes[1]

Oliver A. McBryan[2, 3]

C-3, MS-B265,
Los Alamos National Laboratory,[4]
Los Alamos, NM 87545.

1. Introduction

This paper is part of an ongoing effort to exploit parallelism in the solution of equations arising in Computational Fluid Dynamics. We have previously presented implementations of conjugate gradient and multigrid solvers on other parallel architectures, including the Denelcor HEP shared memory computer[1, 2, 3, 4, 5, 6], the 32 processor Caltech Mark II Hypercube and the Intel iPSC d7 processor, a 128 processor Hypercube[7, 8, 9, 10, 11]. We have also developed an extensive portable linear algebra package for such systems, see[10, 11]. The current work extends these studies to the range of massively parallel architectures, beginning with the Connection Machine, a 65,536 processor hypercube architecture. The Connection Machine (CM in the sequel) is different from other architectures we have worked with in several respects. We will discuss some of these differences in this introduction. For a more complete discussion of the Connection Machine, and of the implementation of numerical algorithms on the machine, we refer to our paper[12].

Most significantly, the CM is a massively parallel machine. The CM therefore requires that applications be decomposed at a very fine level, presenting an interesting challenge in implementing applications. The CM processors are simple one-bit

---

1. For presentation to the 2nd conference on Hypercube Multiprocessors, Knoxville, Oct 1, 1986.
2. Supported in part by DOE contract DE-AC02-76ER03077.
3. Supported in part by NSF grant DMS-83-12229.
4. Permanent address: Courant Institute of Mathematical Sciences, New York University, New York, N.Y. 10012.

processors. Floating point operations, which are supported in micro-code, consume many (up to 1,000) machine cycles, with resulting floating point performance on the order of 1 kflops per processor. Despite this, peak rates of 120 Mflops have been attained on 32-bit vector operations using 64k processors, demonstrating the power achievable with massive parallelism. Each processor is associated with only 512 bytes of memory. This is compatible with the fine-grained parallelism, although it ensures that inter-processor communication is required more frequently on the CM than on coarser-grained machines. Fortunately communication on the CM is fast and has minimal communication startup cost.

The CM is an SIMD machine rather than the MIMD architecture of other hypercubes. While all processors receive identical instructions on each cycle, some processors may choose to ignore an instruction, depending on the setting of an internal flag. Logical expressions are implemented using this facility, although some care is required as each nested binary branch will incur an effective increase in execution time for the expression of a factor of 2.

The current CM software environment is entirely LISP based, with the standard programming language being *LISP, a parallel extension of COMMON LISP. The powerful user-friendly software environment of the CM is unique among current parallel processors. Support for parallelism is fully integrated into the programming language. To a large extent the message passing characteristics of the hypercube network are hidden from the programmer. The system provides support for distributed data types and has facilities for parallel global memory reference.

A final feature of the CM architecture is the possibility of using *virtual processors*, extending the apparent degree of parallelism to over a million. Virtual processors perform more slowly and with less local memory than physical processors, but total machine throughput may actually be increased. The ability to program complex applications in terms of very fine-grain parallelism should not be underestimated. We

illustrate with the example of a typical grid-based computation. On medium-scale machines, it is necessary to perform two independent decompositions of data and associated code. First, the data is broken into blocks of grid points, with one block associated to each processor. Within each processor a further decomposition is required down to the single point level. On the CM, with up to a million virtual processors, one can generally avoid the first step entirely.

For the rationale behind the CM design see the book by Hillis[13]. Further information on the CM architecture is available in documents from Thinking Machines Corporation, for example[14].

Sections 2 and 3 introduce the CM hardware and software in more detail. Sections 4 and 5 describe the implementation of the multigrid and conjugate gradient algorithms on the CM, while section 6 presents performance measurements for the CM in PDE solution.

2. Connection Machine Hardware

The CM is accessed through a standard architecture front end computer, currently the Symbolics 3600, though a VAX interface is about to be released. Connection machine programs contain two types of statements - those operating on single data items, which are executed in the front end, and those operating on whole data sets which are executed in the CM. *LISP instructions for the CM are sent first to a micro-controller which expands them into a series of machine instructions. Floating point instructions are expanded in this way.

The CM consists of 4096 chips, each with 16 processors plus associated memory, for a total of 65536 processors. The chips form a 12 dimensional hypercube. Within each chip the processors are fully connected. Each chip also has a *router* module, a communication processor that allows any on-chip processor to communicate with any remote processor.

In addition to the router hypercube network there is a separate communication facility called the *NEWS grid*. Each processor is wired to its four nearest neighbors in a two-dimensional rectangular grid. Communication on the NEWS grid is extremely fast and is encouraged for those processes that can avail of the limited interconnections involved. Long range communication, even on a grid, is best done however with the router system.

An important feature of the CM system is support for *virtual processors*. The user may specify that each physical processor is to simulate a small rectangular array of virtual processors. The current machine provides up to 1,048,576 virtual processors. All system facilities are transparent to virtual processes, except that such processors appear to be correspondingly slower and have only a fraction of the 512 bytes of memory of a physical processor. The NEWS grid and hypercube communication facilities are supported between arbitrary virtual processors.

3. The CM Programming Environment

The programming languages available on the CM are a parallel extension of COMMON LISP, called *LISP, and an assembly language called PARIS. For a complete description of COMMON LISP see the book by Steele[15]. For further details of the *LISP language see the Thinking Machine Corporation's *LISP manual[16].

*LISP is an extension to COMMON LISP that includes facilities for utilizing parallelism. The primary data extension is the concept of a *pvar*, or parallel variable. Pvars are defined using the **defvar* function, in the same way as *defvar* is used for defining ordinary LISP variables. A pvar can be thought of as a sequence of ordinary LISP variables, one per processor. Parallel constants are supported with a special function "!!": if c is a LISP object, then (!! c) returns a pvar which contains the value c in every processor. Parallel versions of many standard LISP operators are defined, typically with !! appended to the LISP name. For example:

```
(*!! (!! 2) p )
```

applies the parallel multiply operation *!! to the constant pvar (!! 2) and the pvar *p*. Similarly there are parallel versions of logical operators such as <!! , *and*!! , and so on.

Logical and looping constructs parallel those in LISP and allow an operation to be performed over any subset of processors. This *selection* mechanism is very powerful and is a primary mechanism for providing non-homogeneous SIMD instructions. Selection scoping is block structured.

Since pvars are distributed across the CM, it is essential to be able to locate neighboring elements of a pvar, and to communicate in parallel with other processors. This is accomplished by a set of pvar functions which provide parallel global memory references. Processor addressing in these functions may be either absolute or relative, and may be performed using either grid or hypercube indexing.

4. Parallel Multigrid

The basic multigrid idea[17, 18, 19, 20, 21, 22, 7] involves two aspects - the use of relaxation methods to dampen high-frequency errors and the use of multiple grids to allow low-frequencies to be relaxed inexpensively. For the Connection Machine, there is an important difference in approach from that on other parallel machines. The CM has only 512 bytes of memory per processor, which means that it is not possible to store a substantial subgrid of points per processor as was done in our other implementations[7]. Instead, we have chosen to assign only one grid point per processor. This may seem wasteful of memory, but is not really so because of the CM's virtual processor ability. By using sufficiently many virtual processors all memory may be fully utilized.

In our multigrid implementation, the coarse grid points are always assigned to the same processor as their corresponding fine grid points. Because of memory limitations

this implies a restriction on the number of multigrid levels that can be accommodated. When relaxation is performed on coarse grids, most processors are idle. As a result, V-cycles are more favored than W-cycles on the CM, and multigrid iterations with more than about 5 levels are also undesirable. This is borne out by our experiments presented in the results section.

We have used parallel versions of modified Jacobi relaxation and of red-black Gauss-Seidel relaxation. There is little obvious advantage to using red-black relaxation on the CM since it requires 2 successive sweeps each involving only half of the grid points. Consequently half of the processors are unused at any time. Thus the known improved convergence rate of red-black Gauss-Seidel iteration, which is twice as fast for the Poisson Equation, is canceled by the 50% reduction in attainable CM utilization. We note however that if 2 or more virtual processors are used per physical processor, then this disadvantage disappears - all of the physical processors may remain active at all times. We also note that with one grid point per processor, and a completely parallel execution, lexicographic Gauss-Seidel relaxation reduces to ordinary Jacobi relaxation.

We accomplish the distribution of data to various grid levels, by representing the solution, error, and right-hand-side on each grid level as a *LISP pvar. In particular the hypercube is organized as a rectangular mesh of virtual processors. Each grid is associated with a selection pvar called *domain* which is simply a 1-bit pvar initialized to be true at all virtual processors that contain a grid point of that grid, and false elsewhere. Relaxation is performed only for domain points for that grid. The same mechanism allows irregular rectangular grids to be handled as easily as regular grids.

The fine-grid residual equation is projected to the coarse grid using full injection in the modified Jacobi case or half-injection in the red-black Gauss-Seidel case. As discussed previously, no communication is involved here since the relevant fine-grid point is in the same processor as the target coarse-grid point.

To control termination of iteration, norms of error values or residuals need to be calculated. These are evaluated by using the *LISP function **sum* which tree-sums the elements of a pvar over a set of processors. In terms of **sum* we can define a *norm-vector* function, to evaluate the norm of a vector stored as a pvar on an arbitrary subset of processors, again specified by selection.

Having solved the error equation on the coarse grid, the solution on the fine grid has to be updated by addition of a suitable interpolation of the computed coarse grid error. We have used linear interpolation at this point. This step does involve communication, in fact over long distances in the case of very coarse grids, in which case hypercube communication channels are used.

5. Parallel Conjugate Gradient

Discretization of elliptic partial differential equations by finite element or finite difference methods leads to systems of equations with sparse coefficient matrices. The parallel conjugate gradient method we have developed on the CM, solves systems of equations with such coefficient matrix structures. This allows us to parallelize the solution of finite element discretizations of arbitrary and even variable degree with high efficiency.

The preconditioned conjugate gradient method[23, 24, 25, 26, 27, 28] finds the solution of the system of equations $Ax = f$ (A is assumed to be positive definite symmetric) to a specified accuracy ϵ by performing an iteration on the vector x, which has been appropriately initialized. Apart from simple vector linear algebra, this iteration involves only the operations $x \rightarrow Ax$, $x \rightarrow Bx$, $\langle x,y \rangle$, and simple vector linear algebra operations. Here B is an approximate inverse of A , which is also assumed to be positive definite symmetric, and $\langle x,y \rangle$ denotes the inner product of vectors x and y. A carefully chosen *preconditioning* operator B can be effective in improving substantially the convergence rate of the unpreconditioned algorithm (case $B=I$)[25].

We parallelize the algorithm by exploiting parallelism in every operation of the iteration. All of the vectors in the algorithm are allocated as *LISP pvars. The communication-intensive operation $p \rightarrow Ap$ is implemented as a *LISP function which stores the value of Ap into a pvar ap. For our Poisson equation test problem with a 5-point discretization on a rectangle, this function is easily written using the global memory access functions mentioned in section 3. For simplicity we have chosen the pre-conditioning operator to be the identity operator. The other communication intensive operations in the conjugate gradient algorithm are the several inner products of vectors which are required. These are trivially implemented using the **sum* *LISP function.

6. Computational Results

As a test problem we have solved a Poisson equation on a rectangle using SOR relaxation, multigrid and conjugate gradient methods. Zero Dirichlet boundary conditions were imposed on all sides. Iterations were continued until the initial residual was reduced by a specified factor, usually .001. Most of the results reported were obtained on a 16k processor CM, with some measurements on 32k processors. However all indications are that results demonstrated here on 16k and 32k systems scale essentially linearly to the 64k system.

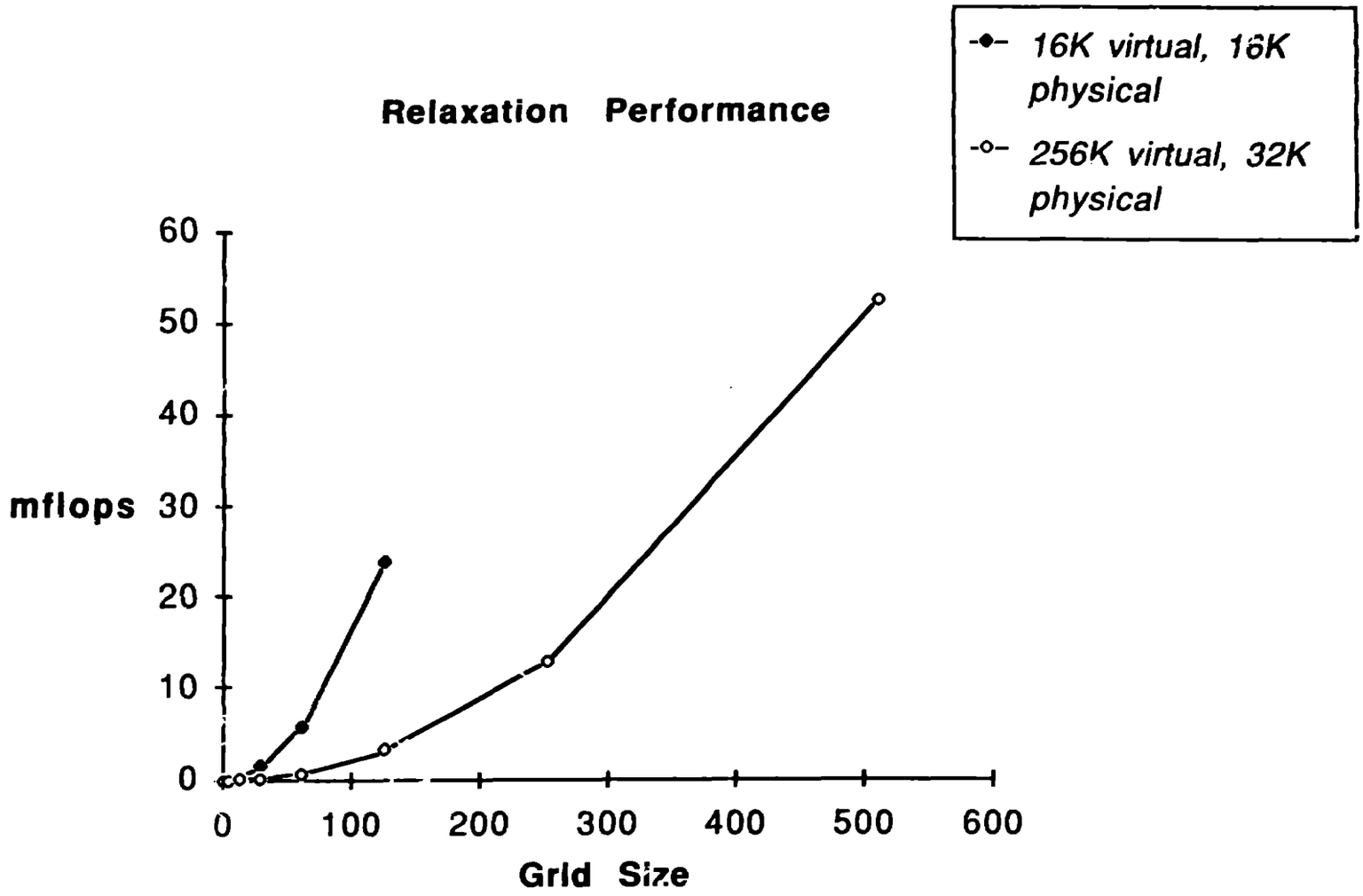
We performed various tests using 4 and 8 virtual processors per physical processor, with up to 256k virtual processors. Such a configuration provides an excellent test of the ability to effectively use very fine-grained parallelism. We plot two quantities that describe system performance. The *megaflops* attained in a computation is the average number of millions of floating point operations executed per second. We count only standard floating point arithmetic operations such as addition and multiplication. A closely related quantity methods is the *time per iteration*, defined by taking the total computation time for a solution and dividing it by the number of iterations

used. The *grid size* label on the following graphs denotes the number of grid points in each dimension.

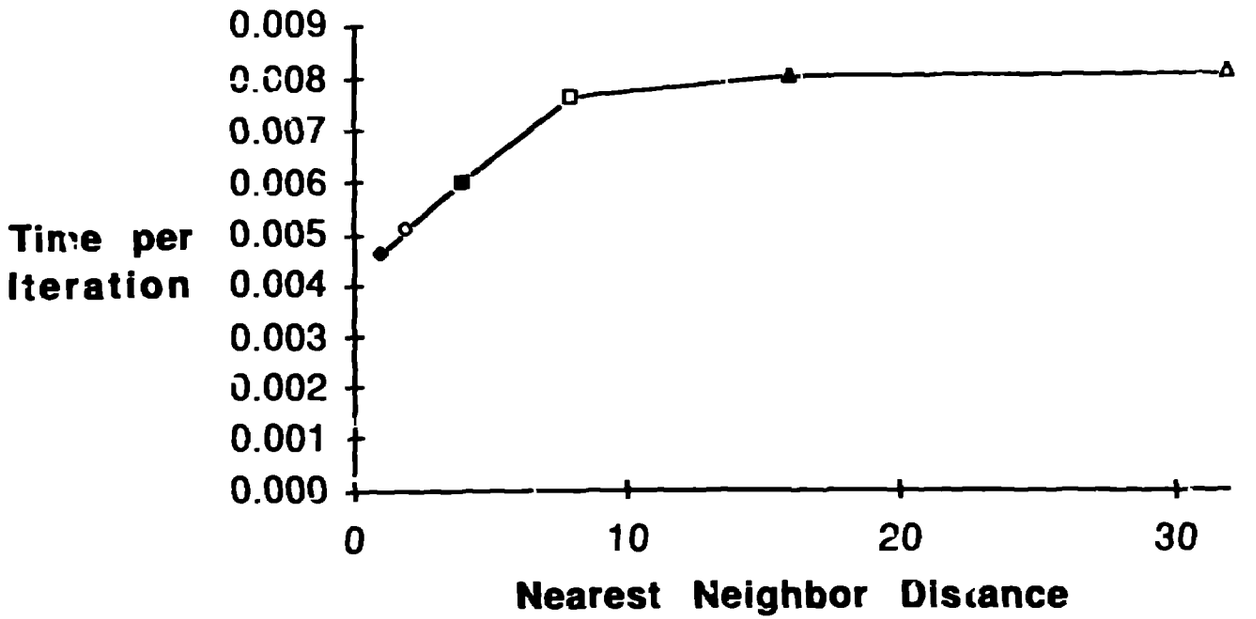
6.1. Relaxation

In Figure 1 we present the results for solution of the equations using straight SOR relaxation. The relaxation parameter $\omega = .8$ for successive over-relaxation in Jacobi iteration has been used. Two curves are presented in Figure 1, the shorter one is for a 16k processor machine with 1 virtual processor per physical processor, while the second curve is for a 32k processor machine with 8 virtual processors per physical processor. The peak rate of 52 Mflops obtained in the latter case, corresponds to about 104 Mflops for a full 64k processor Connection Machine. Relaxation performance improves rapidly with grid size, until it reaches its optimal value with a grid which fills the virtual processor network.

The relaxation results in Figure 1 involved only nearest neighbor communication patterns. Multigrid requires relaxation on coarser grid levels, where communication over substantially longer distances is required. We measure the effects of long range communication in Figure 2, where we plot time per SOR relaxation versus the inter-processor distance involved. A series of grids was used each of which had the same number of points, but with the distance between grid-points, which we refer to as the *nearest neighbor distance,* successively increasing. We used 32 different nearest neighbor distances. As mentioned previously, there are two distinct communication facilities on the CM - the rectangular NEWS grid and the hypercube Router network. Each of the relaxation tests was performed using whichever communication method was faster. The flat later part of the curve in Figure 2 indicates the remarkable fact that hypercube communication on the CM is essentially independent of distance.


Relaxation Performance
16K virtual, 16K physical
256K virtual, 32K physical
mflops
60
50
40
30
20
10
0
0
100
200
300
400
500
600
Grid Size

Relaxation using 128*128 processor grid and optimal choice between News and Hypercube



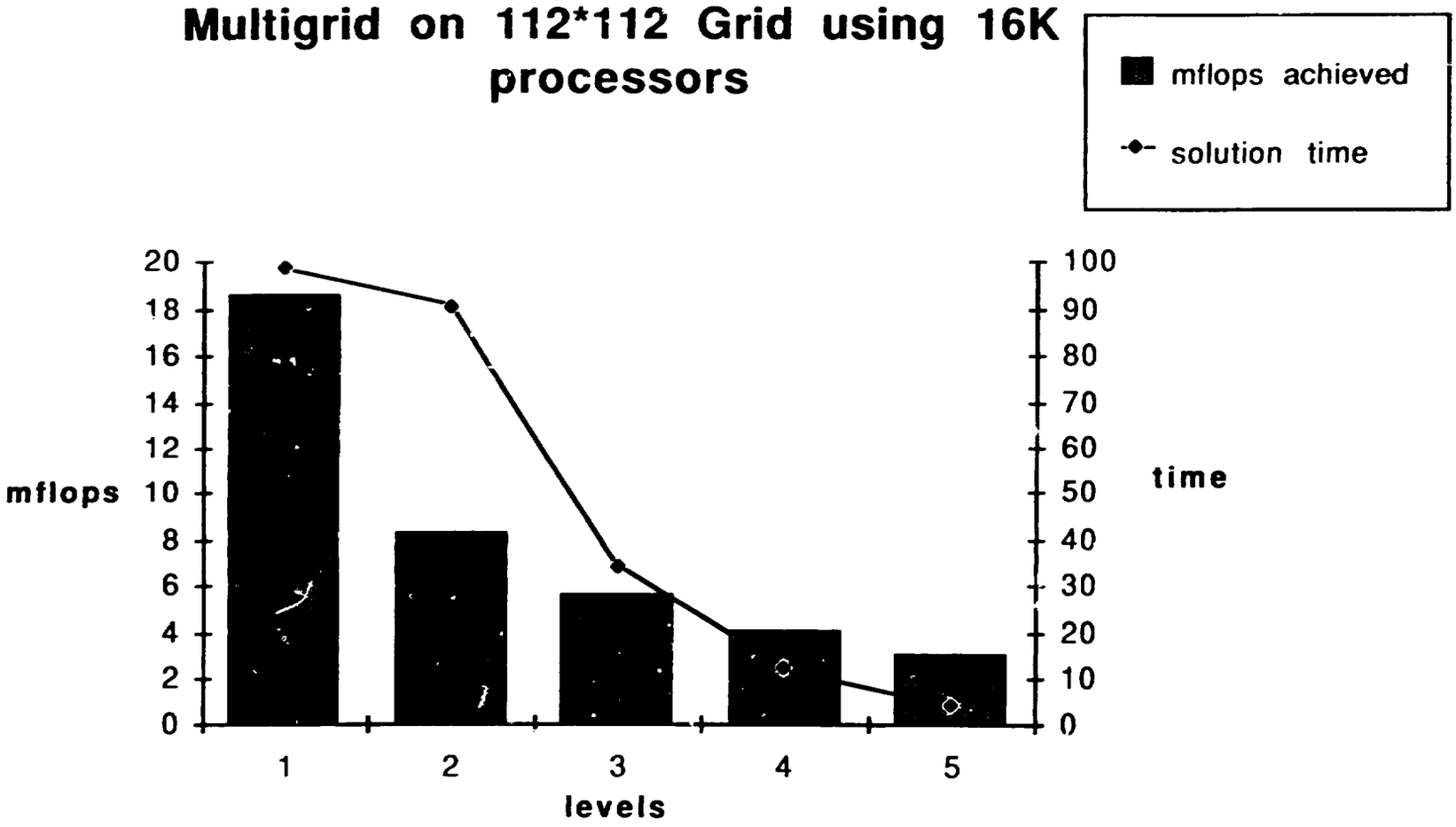
6.2. Multigrid Results

All multigrid solutions were obtained using V-cycles, with 3 modified Jacobi relaxations per grid level. The coarsest grid level was treated identically to the other levels. In Figure 3 we present measurements of multigrid performance on a 16k processor machine. All computations were on a 112×112 grid which allows for even sub-divisions down to 5 levels. The *curve* in Figure 3 shows the time required for the complete multigrid solution as a function of the number of levels used in solution. In standard multigrid fashion the solution time drops rapidly as the number of grid levels increases. One might conclude that multigrid is performing well on the CM. This is not in fact true as is borne out by the bar chart in Figure 3 which represents megaflops attained in multigrid solution as a function of the number of levels used. Here one sees that the maximum performance of over 19 Mflops when only one level is used (i.e. straight relaxation), drops rapidly to just over 3 Mflops when 5 levels are used.

To understand this behavior, we note that in the multi-level cases most processors are sitting idle much of the time, thus diminishing the ability to use available megaflops. Even on the highest level grid only 75% of the available processors are in use. On a fifth-level grid, only 49 of the available 16k processors are active. Yet the algorithm spends the same amount of *time* on such a grid as it does on the finest grid since the CM is an SIMD machine. It is actually remarkable that megaflops rates do not decay even more severely than indicated. Another feature of Figure 3 is the rather sharp drop in performance between level 1 and level 2. This is explained by the fact that in going from a 1-level problem (relaxation) to 2 levels a variety of overheads are incurred, including residual computations and coarse-to-fine grid transfers.

We conclude that standard multigrid as implemented here is not a particularly good algorithm for a massively parallel machine such as the Connection Machine[1].

1. This observation has prompted further research into parallel multigrid algorithms, and has resulted in a new class of multiple grid algorithms with much better megaflops rates, see[29].


Multigrid on 112*112 Grid using 16K processors
mflops achieved
solution time
mflops
time
20
18
16
14
12
10
8
6
4
2
0
100
90
80
70
60
50
40
30
20
10
0
1
2
3
4
5
levels

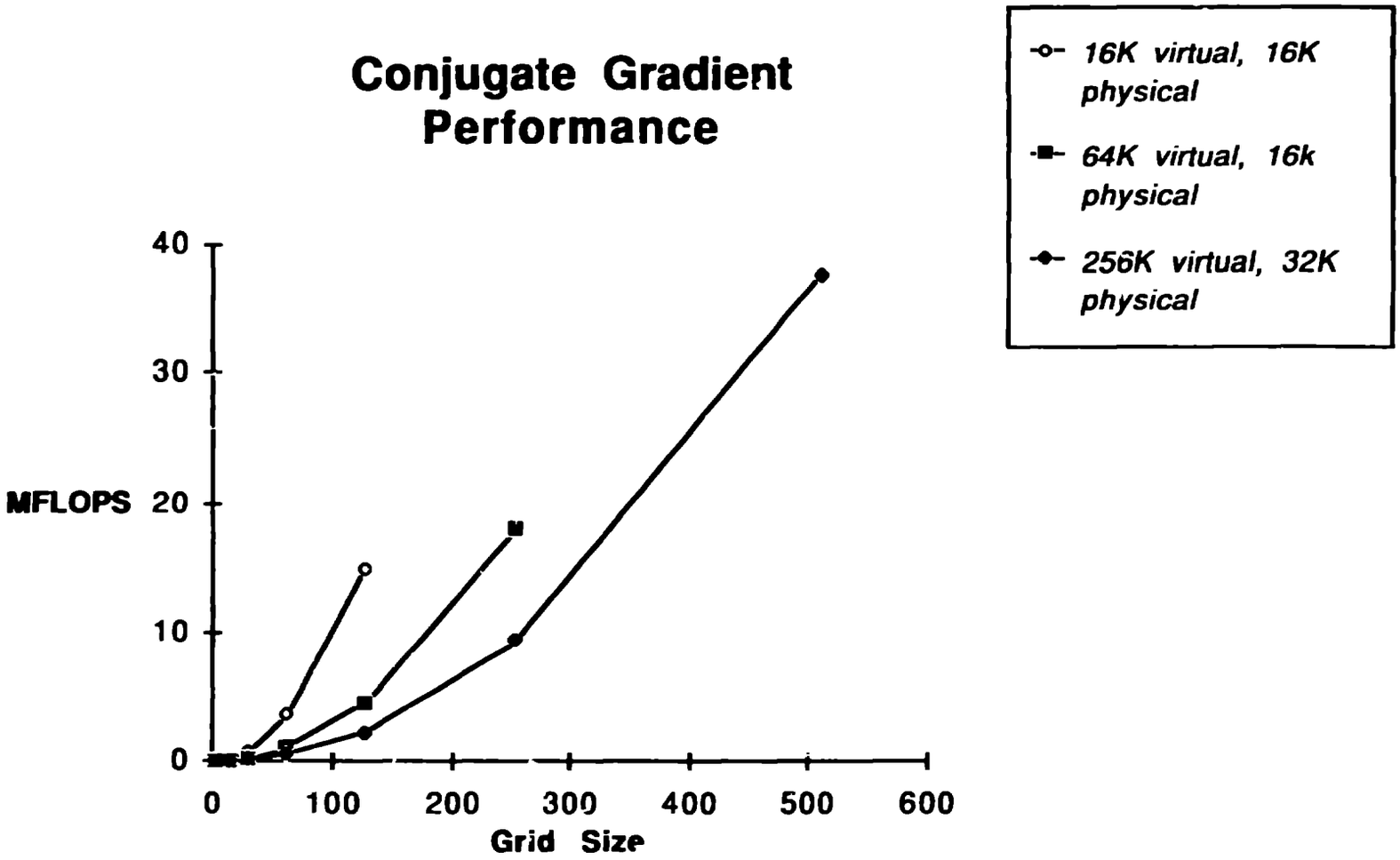
6.3. Conjugate Gradient Performance

Conjugate gradient performance is much more satisfactory than for multigrid because all processors may be kept busy most of the time, with the exception that during evaluation of the many inner products required by conjugate gradient most processors are inactive. This leads to a reduction in attained megaflops as compared to relaxation, but overall performance is still quite satisfactory, being in the region of 80 Mflops for a full 64k machine.

We present the results for conjugate gradient solution in Figure 4, where again we plot megaflops as a function of grid size. There are three curves, corresponding respectively to the cases (i) 16k virtual processors on 16k physical processors, (ii) 64k virtual processors on 16k physical processors and (iii) 256k virtual processors on 32k physical processors. Note in particular the relationship of the first two curves. The top point on these two curves both correspond to computations on 16k *physical* processors in which all processors are in use - in the first case computing with 16k grid points, and in the other case with 64k grid points. We see that higher megaflops are attained by using virtual processors. With virtual processors, much of the interprocessor communication is occurring within a single physical processor and thus external communication overhead is lower.

7. Acknowledgements

We wish to thank Thinking Machines Corporation, Cambridge, Mass., for providing access to the Connection Machine. We also benefited enormously from the uniquely stimulating environment at TMC, and acknowledge the help of many who were instrumental in providing hints to using the system.


Conjugate Gradient Performance
MFLOPS
40
30
20
10
0
0
100
200
300
400
500
600
Grid Size
16K virtual, 16K physical
64K virtual, 16k physical
256K virtual, 32K physical

References

1. O. McBryan, E. Van de Velde, and P. Vianna, "Parallel Algorithms for Elliptic and Parabolic Equations," *Proceedings of the Conference on Parallel Computations in Heat Transfer and Fluid Flows*, University of Maryland, November 1984.
2. O. McBryan, "State of the Art of Multiprocessors in Scientific Computation," *Proceedings of European Weather Center Conference on Multiprocessors in Meteorological Models, Dec. 1984*, EWCMF, Reading, England, 1985.
3. O. McBryan and E. Van de Velde, "Parallel Algorithms for Elliptic Equation Solution on the HEP Computer," *Proceedings of the Conference on Parallel Processing using the Heterogeneous Element Processor, March 1985*, University of Oklahoma, March 1985.
4. O. McBryan and E. Van de Velde, "Parallel Algorithms for Elliptic Equations," *Commun. Pure and Appl. Math.*, vol. 38, pp. 769-795, 1985.
5. O. McBryan and E. Van de Velde, "Parallel Algorithms for Elliptic Equations," in *New Computing Environments: Parallel, Vector and Systolic*, ed. A. Wouk, SIAM, 1986.
6. O. McBryan and E. Van de Velde, "Elliptic Equation Algorithms on Parallel Computers," *Commun. in Applied Numerical Methods*, vol. 2, pp. 311-316, 1986.
7. O. McBryan and E. Van de Velde, "The Multigrid Method on Parallel Computers," in *Proceedings of 2nd European Multigrid Conference, Cologne, Oct. 1985*, ed. J. Linden, GMD Studie Nr. 110, GMD, July 1986.
8. O. McBryan and E. Van de Velde, "Hypercube Algorithms for Computational Fluid Dynamics," in *Hypercube Multiprocessors 1986*, ed. M. T. Heath, pp. 221-243, SIAM, Philadelphia, 1986.
9. O. McBryan and E. Van de Velde, "Architectural and Software Issues for Medium Scale Multiprocessor Systems," *Proceedings of the ARO workshop on Parallel Processing for Medium Scale Multiprocessors, Stanford University, Jan 1986*, SIAM, to appear.
10. O. McBryan and E. Van de Velde, "Hypercube Algorithms and Implementations," *Proceedings of the 2nd SIAM Conference on Parallel Processing for Scientific Computation, Norfolk, Nov. 1985*, SIAM, March 1987, to appear.
11. O. McBryan and E. Van de Velde, "Matrix and Vector Operations on Hypercube Parallel Processors," *Parallel Computing*, Elsevier, Jan 1987, to appear.

12. O. McBryan, "The Connection Machine: PDE Solution on 65536 Processors," Los Alamos National Laboratory Preprint, Aug 1986.

13. W. Daniel Hillis, *The Connection Machine*, MIT Press, Cambridge, Mass, 1985.

14. "Introduction to Data Level Parallelism," Thinking Machines Technical Report 86.14, Cambridge, Mass., April 1986.

15. G. L. Steele Jr., S. E. Fahlman, R. P. Gabriel, D. A. Moon, and D. L. Weinreb, *Common Lisp: The Language*, Digital Press, Burlington, Massachusetts, 1984.

16. *The Essential *LISP Manual, Release 1, Revision 7*, Thinking Machines Corporation, Cambridge, Mass., July 1986.

17. A. Brandt, "Multi-level adaptive solutions to boundary-value problems," *Math. Comp.*, vol. 31, pp. 333-390, 1977.

18. W. Hackbusch, "Convergence of multi-grid iterations applied to difference equations," *Math. Comp.*, vol. 34, pp. 425-440, 1980.

19. K. Stuben and U. Trottenberg, "On the construction of fast solvers for elliptic equations," *Computational Fluid Dynamics*, Rhode-Saint-Genese, 1982.

20. A. Brandt, "Multi-Grid Solvers on Parallel Computers," ICASE Technical Report 80-23, NASA Langley Research Center, Hampton Va., 1980.

21. O. McBryan, "Fluids, Discontinuities and Renormalization Group methods," *Physics 124A*, pp. 481-494, North-Holland Publishing Company, Amsterdam, 1984.

22. D. Gannon and J. van Rosendale, "Highly Parallel Multi-Grid Solvers for Elliptic PDEs: An Experimental Analysis," ICASE Technical Report 82-36, NASA Langley Research Center, Hampton Va., 1982.

23. 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.

24. 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.

25. 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.

26. 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.

27. 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.

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

29. P. Frederickson, O. McBryan, and Parallel Superconvergent Multiple Scale PDE Solvers, Los Alamos National Laboratory Preprint, to appear.