UCLA
COMPUTATIONAL AND APPLIED MATHEMATICS

Hierarchical Algorithms and Architectures for Parallel Scientific Computing

Tony F. Chan

April, 1990

CAM Report 90-10

Department of Mathematics
University of California, Los Angeles
Los Angeles, CA. 90024-1555

Hierarchical Algorithms and Architectures for Parallel Scientific Computing

Tony F. Chan*

April 16, 1990

Abstract

There has been a recent emergence of many interesting and highly efficient hierarchical (multilevel) algorithms (e.g. multigrid, domain decomposition, wavelets, multilevel preconditioning, the fast multipole algorithms, etc.) for solving numerical problems in scientific computing. These algorithms derive their efficiency from using a hierarchical approach to capture the sharing of global information which is inherent in the physical processes being modelled. In addition to being computationally efficient, these algorithms also possess relatively high degrees of parallelism. I therefore argue that the architectures of parallel computers (especially massively parallel ones) should be designed to support hierarchical communication and synchronization needs of these algorithms. Hierarchical architectures are also more universal because they go beyond supporting a particular class of algorithms to supporting the underlying physical processes being modelled. Finally, I propose that algorithm designers take a critical look at traditional kernel algorithms and re-examine their cost-effectiveness in a massively parallel computing environment.

*Dept. of Mathematics, Univ. of Calif. at Los Angeles, CA 90024. The author has been supported in part by the Army Research Office under contract DAAL03-88-K-0085, the Department of Energy under contract DE-FG-03-87-ER-25037, and the National Science Foundation under contract NSF-DMS87-14612. Invited presentation to appear in the Proceedings of the 1990 ACM Int'l Conf. on Supercomputing, Amsterdam.

1 Physics, Algorithms and Architectures

There is a general agreement that parallelism is a cost-effective way (some may argue it is the *only* way) to provide the increasing demand in performance in computational power in many areas of science and engineering. There is much less agreement, however, on how to design effective parallel machines. It is well-known that it is not sufficient to simply compare the peak computing rates of various designs. A successful architecture must strike a balance between the computational rate and the communication and synchronization overheads, the memory hierarchy and its bandwidths. Most importantly, the architecture must support effective computational algorithms which can actually be used to solve the problems that scientists and engineers need to solve.

How should parallel machines be designed so that they can achieve this ultimate goal? If one looks around today on the architectures that have been proposed and built, one

finds a wide variety of approaches. Here is a partial list:

1. "General purpose" machines: designed for a general problem domain, e.g. artificial intelligence (e.g. the Connection Machine CM-1, although Thinking Machines Inc. has since then marketed the CM-2 as a numerical supercomputer as well) and scientific/numerical computing (e.g. Cray computers).

2. Data type machines: designed to work efficiently on certain data types, such as bit, scalar, vector and arrays.

3. Interconnection network machines: distributed multiprocessor computers classified by their underlying interconnection network, e.g. hypercubes (Intel iPSC, Ncube), tree, and meshes (e.g. the ICL DAP, the ILLIAC IV, the Goodyear MPP, and the Intel iPSC/2 with the Direct-Connect network interface).

4. Algorithm-specific machines: designed to run specific algorithms efficiently, e.g. the SAXPY 1M (for the BLAS Saxpy computation: $y = ax + y$) and many systolic arrays.

5. Problem-specific machines: designed for a specific physical or mathematical problem, e.g. the Navier-Stokes Computer [25] and the SUPRENUM multiprocessor system [30].

Given this large number of approaches, it is clear that there is no general agreement. Of course, the best design should depend on the ultimate intended use of the machine. A special purpose machine can be made more efficient than a general purpose machine but is more restrictive in its use. For example, machines intended for non-numerical applications need not even have floating point computing units. Machines designed for some AI applications probably need to handle tree searches efficiently.

For the purpose of this paper, we shall restrict our attention to one general problem area, namely scientific/numerical computing, by which I mean the field of computer simulation of mathematical models describing processes in mathematical physics (usually involving partial differential equations and/or integral equations). This includes computational fluid dynamics, computational chemistry, computational structural mechanics, semiconductor and circuit simulations, and oil reservoir simulations. Certainly, this is one of the largest classes of problems which has prompted the advent of parallel supercomputing and can also benefit the most from it.

Given this problem class, how should parallel machines be designed so that they can be used effectively to solve the large variety of computational problems that arise? How should the architecture interact with the algorithms and the physical problems? Should the architecture support specific algorithms? If so, which algorithms? How does one ensure that such a machine can adapt to new and more efficient algorithms that will emerge in the future? How should an algorithm designer strike a balance between designing algorithms for the physical problems versus designing for a specific computer architecture? These are some of the questions which I'd like to address in the rest of the paper.

My main thesis is that both algorithms and architectures should be designed to account for the fundamental physical processes in the mathematical models and that hierarchical algorithms and architectures are good ways of achieving this goal. Some of the

views in this paper have been expressed before in [6].

2 The Hierarchical Nature of Physical Processes

We shall first look at the fundamental nature of the physical processes which are implicitly described by the mathematical models of the physical laws. These are often in the form of time-dependent partial differential equations (PDEs) of the form

$$u_t = F(u(x,t))$$

where $u(x,t)$ is the value of a state variable (or a vector of variables) at a point x in a physical domain Ω and at time t. The PDE essentially describes the evolution of u in space and time when given initial and boundary conditions. In a numerical solution approach, the PDE is discretized on a computational grid, with differentiation replaced by discrete approximations (e.g. finite differences), and the discrete model is then simulated on the computer. Sometimes a full simulation in space-time is needed while sometimes only the *steady state* (or *equilibrium*) solution is needed (the steady state solution $u(x)$ satisfies the equation $F(u(x)) = 0$.

It has often been stated that PDEs are fundamentally local in nature: the evolution of a variable in a certain point in space depends only on the values of variables in a local neighborhood of the point. One argument is that the differentiation operator inherent in PDEs (and their approximations) are local by definition. Another reason is that the PDEs are really asymptotic continuous models of locally interacting discrete elements in the real physical world (e.g. molecules, charge particles, etc.).

While this point of view is based on a sound basis, it would be wrong to conclude that PDEs only describe processes that are local in nature. A prevalent feature of many physical systems, and one of the basic difficulties in their numerical simulation, is the presence of a large range of scales, both in space and time. Activities on a large scale is relatively global compared to those on a smaller scale. For example, in the modelling of atmospheric circulation, the time scale ranges from the (slow) speed of fronts to the (fast) speed of sound in the air, and the length scale ranges from the (large) size of continents to the (small) size of a tornado.

The presence of different scales plays an even more fundamental role in the modelling of steady state configurations. Mathematically, these are typically modelled by *elliptic* PDEs, which have infinite domains of dependence, i.e. the solution at any point depends on data at every other point in the computational domain. For example, increasing the load at a single point on the span of a bridge will change the steady state deflection of the bridge at every other location.

Thus, while PDEs may be defined via local operations, the solutions they describe are often global in nature. In fact, in many disciplines, the fundamental objective is to understand the interaction of the local and global scales (e.g. turbulence).

3 Algorithm Design Tradeoffs

A fundamental design principle for PDE algorithms is to capture efficiently the interaction of the global and local features of the physical processes described by the mathematical model. If only the global features are required, then one should minimize the

computational effort spent in computing on the small local scales.

The simplest algorithms are the local ones, exploiting the local nature of the discrete models. These have the advantages of simplicity, efficiency *per step* and highly parallel. But often they are slow because many local steps must be taken in order to account for the global interactions. On the other hand, global algorithms accounts for more of the global features of the solutions per step and can therefore take fewer steps but they are usually more complex, more costly per step and are less parallelizable.

The best algorithm for a particular problem depends on many factors, such as the desired accuracy of the final solution, the accuracy of the initial guess, whether the transient and/or the steady state is needed, and last but not least, the computer on which the algorithm is to be executed. The design process often involve delicate tradeoffs between the advantages and disadvantages of the local and global algorithms.

An example of this tradeoff is that between explicit and implicit algorithms for solving time dependent problems. Explicit methods are local in space and are highly parallelizable. In fact, many parallel computers are designed specifically with solving PDEs with explicit methods in mind, e.g. the ILLIAC IV, the ICL DAP. However, by having only local interactions, one limits the time scale to that of the smallest spatial scales represented on the computational grid. Consequently, the size of the time step is limited (by what is usually referred to as the CFL condition) and thus many time steps may have to be taken than is necessary for the desired accuracy of the solution. On the other hand, implicit algorithms involve sharing of global information, usually in the form of having to solve a system of equations per time step, but are less limited in the size of the time step and thus can take many fewer steps than an explicit method.

Many successful schemes can be viewed as striking a balanced tradeoff between these two extremes by modifying implicit schemes so that they are more efficiently computable per step but still share global information effectively. An example of this are the fractional step and approximate factorization schemes [3] which approximate spatial differential operators in multiple spatial dimensions by products of simpler one dimensional operators, typically in the form of having to solve narrowly banded linear systems which allow global sharing of information but are still efficiently solvable at least on conventional computers. Explicit methods were very popular in the early days of computing but implicit methods are seeing increasing use more recently.

If only the steady state solution is desired, then in principle it *could* be obtained by an explicit method which computes the transient solution from an initial state for a long enough period of time. But that is not the only way and often more efficient methods can be found to compute the steady state solution directly. For example, computing the transient (and probably highly oscillatory) motion of a bridge after a sudden load increase in order to obtain the steady state solution may not be the most effective method.

One alternative is to solve for the steady state solution directly by an iterative method [26]. Here the tradeoff between explicit (local) and implicit (global) algorithms is again present. The simplest methods are local relaxation methods (e.g. Jacobi, Gauss-Seidel, and SOR) which are purely explicit but are very slow in general. More implicit methods (e.g. ADI, block methods, incomplete LU preconditioned conjugate gradient methods) are faster but are more costly per step. The extreme is the direct solution of the discrete

equations by Gaussian elimination.

The underlying computing model also has a strong influence on the design of the algorithms. Many classical iterative methods (e.g. SOR, ILU) traverse the computational grid in a sequential manner, imposing a strict data-dependence which limits the degree of parallelism. Simpler and more local methods (e.g. Jacobi) are more parallelizable but are in general slower because they cannot account for the global nature of the solution. One attempt to rectify the situation is to re-order the unknowns (e.g. red/black ordering versus natural ordering) so that a higher degree of parallelism is available [14]. Another approach is to use only highly parallelizable computational kernels and try to combine them in a judicious way to construct faster convergent methods (e.g. polynomial preconditioning [26]). However, these approaches can only improve the situation by a limited degree because the resulting methods are still effectively local in nature[9].

Thus, many classical PDE algorithms can be viewed as making tradeoffs between the speed of convergence and the complexity (and degree of parallelism) per step, in order to satisfy the specific needs of a particular problem.

4 Hierarchical Algorithms

The above examples point out the need for algorithms which can capture the global features of the solutions and yet are still highly parallelizable. I would like to argue that hierarchical and multilevel methods are one such class of algorithms. Simply stated, these methods are good because they capture the global features of the solution efficiently, via local (and hence parallelizable) operations on each of a hierarchy of computational grid, each representing a different length scale.

In the last several years, there has been an emergence of a variety of such hierarchical algorithms. Here are some examples:

1. **Nested Dissection** [17]: This is a technique for ordering the unknowns in the computation grid in order to minimize the fill-in (and hence the computational work) when Gaussian elimination is applied. It is one of the older hierarchical numerical algorithms. The essential idea is to recursively separate the grid into two halves by a small separator set of grid points. By ordering the separator points last, the fill-ins are limited to those involving the separator set only and no fill-ins can occur between the two halves. Thus one can view the separator sets as providing an efficient (because they are chosen to include only a small number of points) and hierarchical global coupling between all the grid points.

2. **Multigrid Algorithms** [5]: These algorithms can be viewed as generalized relaxation methods for elliptic problems, in which the relaxation (or smoothing) is performed on a hierarchy of coarser grids in order to more efficiently annihilate the more global error components. They derive their efficiency by exploiting the different length scales of the solution on the appropriate grid level. These algorithms have seen increasing use in the last 15 years and the basic principles have been applied to many computing problems in science and engineering.

3. **Domain Decomposition Algorithms** [19, 8]: These are relatively new techniques for solv-

ing PDEs in which the computational domain is decomposed into a number of smaller (overlapping or non-overlapping subdomains. The solution is obtained by solving the PDE on the subdomains (and often also on a coarser grid in addition) and iteratively piecing the global solution together through the matching of the solution on the boundaries between the subdomains. The local features of the solution are accounted for by the subdomain solves and the global features are accounted for by the interface coupling and the coarse grid solution.

4. **Multilevel Preconditioners** [23, 4, 2, 1]: These are preconditioning techniques to be used in conjunction with the conjugate gradient method for the iterative solution of elliptic linear systems. As mentioned earlier, traditional preconditioners are either not highly parallelizable or are very slow. This new class of preconditioners can be viewed as one cycle of a standard multigrid method without the smoothing operations. They use the multigrid principle to capture the different length scales of the solution but rely on the conjugate gradient method to deal with other convergence difficulties (e.g. large variations in the coefficients of the PDEs). They offer the efficiency of multigrid methods and the robustness of the conjugate gradient method.

5. **Adaptive Mesh Refinement Algorithms** [24]: These are algorithms for locally and adaptively refining a computational grid where local features demand a more refine resolution, e.g. the presence of steep gradients in the solution. The key idea is again to treat each length scale locally and efficiently on an appropriate grid. The global feature is accounted for by the coarse overall grid. A fundamental issue is the coupling between the refinement levels (i.e. the different length scales).

6. **Wavelet Basis** [28]: Mathematically, the wavelet basis is similar to the Fourier basis in that it forms an orthonormal basis for square-integrable functions. However, unlike the Fourier basis, the wavelet basis consists of functions defined on a hierarchy of nested grids: the basis functions on a particular grid level have almost compact support on that grid and is used to capture the essential features of a function in that length scale. An expansion of a function in the wavelet basis is less sensitive to local fluctuations as compared to a Fourier expansion. Wavelets is considered by many a major development in the field of image and signal processing. Its use for solving PDEs and its relationship with conventional multigrid algorithms are currently being explored [18, 21].

7. **The Fast Multipole Algorithms** [20]: This is a class of algorithms for efficiently computing N-body interactions under certain potential force fields. The main idea is to cluster the effect of a collection of particles (using the multipole expansions) for the purpose of accounting for the relatively weak global interactions with distant particles. More local interactions are computed directly. The idea is applied recursively to achieve optimal efficiency. These methods can be applied to solve PDEs through the boundary integral method.

All of the above hierarchical algorithms

have one feature in common: they capture the global processes in the mathematical models in an efficient way, resulting in near optimal (sequential) computational complexities.

5 Hierarchical Parallel Architectures

Ideally, a good computer architecture should support the best algorithms for solving a class of problems. Since I have argued that hierarchical algorithms are emerging as some of the most efficient for solving PDEs, I therefore take the position that parallel computer architectures for this class of problems should at least support the efficient implementation of hierarchical algorithms. To do this, the architecture must provide efficient ways for hierarchical communications and/or synchronization. The design of memory hierarchies must also take into account the quantity and bandwidth of hierarchical data access requirements. The additional cost of providing these facilities will be compensated by the efficient implementation of fast hierarchical algorithms. Ignoring these issues will result in architectures which will only support the less efficient (and more local) algorithms, possibly resulting in longer execution time for solving the physical problem. Moreover, having an architecture which takes into account the fundamental hierarchical processes of the physical problems to be simulated on it ensures that the architecture performs effectively for a much wider class of algorithms.

There have been many approaches employed to provide hierarchical communication/synchronization facilities in parallel computers. The simplest means to provide global communication is via a global data bus. However, this alone is often not sufficient because data congestion can occur. For distributed memory architectures, a variety of interconnection networks have been proposed to provide more efficient global communication. Many of these networks are hierarchical in nature, e.g. the hypercubes, the shuffle-exchange and other switching networks. Some are even designed specifically for supporting multilevel algorithm, such as the SUPRENUM computer [30]. The most popular is probably the hypercube network, which has been adopted by several commercial manufacturers, e.g. Intel iPSC, Ncube, CM-2, the NSC. The hypercube embeds the local nearest neighbor mesh network, but in addition, also supports hierarchical multilevel communications. By using the binary reflected Gray code of mapping data into processors, it is possible to preserve the locality of data on each grid level of a grid hierarchy [10]. Thus the hypercube is an ideal network for supporting hierarchical algorithms.

For shared memory architectures, one approach is to use a hierarchical clusters of processors, linked together via a hierarchical network of data bus, to a hierarchical memory system. The best example of this is the Cedar system [22], but various features of this (e.g. hierarchical memory systems) have been used in the design of many other shared memory machines.

Of course, the most cost-effective design is technology dependent, and also depends on many other factors, such as the number of processors, etc. If the communication/synchronization technology employed is fast enough compared to computational rate of the arithmetic processors, then the topology of the architecture can be effectively hidden. For example, with more efficient communication technology, the Intel iPSC/2 abandons the hypercube network in its predecessor (the iPSC/1) altogether and rely

solely on a nearest neighbor mesh topology. In the extreme case, with negligible communication costs, the distinction between distributed memory architectures and shared memory architectures becomes fuzzy: data is there when you need it and is accessible at negligible cost. In fact, this framework has been used to design general purpose parallel programming models, e.g. the Linda system. However, especially for massively parallel machines, the communication needs of hierarchical algorithms cannot be completely hidden, and for an ideal optimally balanced machine, the hierarchical structure should be built-in.

On massively parallel architectures, hierarchical algorithms share a peculiar feature: on the coarser levels of the hierarchy, many processors not used by the algorithms may remain idle. Thus in terms of *processor utilization*, the hierarchical algorithms are not optimal. There have been some methods proposed for making better use of the idle processors. For example, for the multigrid solution of elliptic problems on massively parallel computers (see [11] for a brief survey), Frederickson-McBryan [15] combine solutions on multiple coarse grid problems to obtain better convergence rates. Gannon-van Rosendale [16] and Chan-Tuminaro [12] solve problems on each of the grid hierarchy simultaneously. However, the potential gain in the *problem solving efficiency*, while essentially free, is limited, because the unmodified hierarchical algorithms are often already optimal in computational complexity, *even allowing for the idle processor problem.* Some authors [13] have even argued that these new methods are no more efficient than a parallelization the standard multilevel methods, at least when the latter are running at top efficiency. However, these new methods may offer some improvement in more practical situations when it is more difficult to obtain optimal efficiency for the standard methods [31].

The following numerical results from [29] help to illustrate the points just made. In Table 1, we tabulate the results of solving a 2D Poisson problem on a 16K processor CM-2 with floating point hardware in single precision by the conjugate gradient method with several preconditioners. The table shows the number of iterations to achieve a certain accuracy, the total execution time in seconds and the Mflops rate.

Table 1: **PCG for 2D Poisson Problem**

precond.	grid	no. iter.	time	MFLOPS
CG(none)	256x256	401	1.23	406
RIC	256x256	40	57.1	1.4
Jacobi-2	256x256	197	1.15	337
LS-3	256x256	132	1.34	271
MGMF	256x256	26	1.02	47
CG(none)	1024x1024	1525	40.0	760
RIC	1024x1024	too	much	time
Jacobi-2	1024x1024	748	38.7	641
LS-3	1024x1024	503	41.5	534
MGMF	1024x1024	27	18.5	43

The methods tested are: CG(none), the conjugate gradient method with no preconditioning; RIC, the Relaxed Incomplete Cholesky method [26] with natural ordering; Jacobi-2 and LS-3, the Jacobi and least squares polynomial preconditioners with 2 and 3 terms respectively [26]; and MGMF, a multilevel preconditioner [23]. From the tables, it can be seen that the CG(none) always has the highest Mflops rate but never the minimum time (although it is almost the fastest for the smaller 256 by 256 problem), due to the large number of iterations needed, especially for the larger 1024 by 1024 prob-

solely on a nearest neighbor mesh topology. In the extreme case, with negligible communication costs, the distinction between distributed memory architectures and shared memory architectures becomes fuzzy: data is there when you need it and is accessible at negligible cost. In fact, this framework has been used to design general purpose parallel programming models, e.g. the Linda system. However, especially for massively parallel machines, the communication needs of hierarchical algorithms cannot be completely hidden, and for an ideal optimally balanced machine, the hierarchical structure should be built-in.

On massively parallel architectures, hierarchical algorithms share a peculiar feature: on the coarser levels of the hierarchy, many processors not used by the algorithms may remain idle. Thus in terms of *processor utilization*, the hierarchical algorithms are not optimal. There have been some methods proposed for making better use of the idle processors. For example, for the multigrid solution of elliptic problems on massively parallel computers (see [11] for a brief survey), Frederickson-McBryan [15] combine solutions on multiple coarse grid problems to obtain better convergence rates. Gannon-van Rosendale [16] and Chan-Tuminaro [12] solve problems on each of the grid hierarchy simultaneously. However, the potential gain in the *problem solving efficiency*, while essentially free, is limited, because the unmodified hierarchical algorithms are often already optimal in computational complexity, *even allowing for the idle processor problem.* Some authors [13] have even argued that these new methods are no more efficient than a parallelization the standard multilevel methods, at least when the latter are running at top efficiency. However, these new methods may offer some improvement in more practical situations when it is more difficult to obtain optimal efficiency for the standard methods [31].

The following numerical results from [29] help to illustrate the points just made. In Table 1, we tabulate the results of solving a 2D Poisson problem on a 16K processor CM-2 with floating point hardware in single precision by the conjugate gradient method with several preconditioners. The table shows the number of iterations to achieve a certain accuracy, the total execution time in seconds and the Mflops rate.

Table 1: **PCG for 2D Poisson Problem**

precond.	grid	no. iter.	time	MFLOPS
CG(none)	256x256	401	1.23	406
RIC	256x256	40	57.1	1.4
Jacobi-2	256x256	197	1.15	337
LS-3	256x256	132	1.34	271
MGMF	256x256	26	1.02	47
CG(none)	1024x1024	1525	40.0	760
RIC	1024x1024	too	much	time
Jacobi-2	1024x1024	748	38.7	641
LS-3	1024x1024	503	41.5	534
MGMF	1024x1024	27	18.5	43

The methods tested are: CG(none), the conjugate gradient method with no preconditioning; RIC, the Relaxed Incomplete Cholesky method [26] with natural ordering; Jacobi-2 and LS-3, the Jacobi and least squares polynomial preconditioners with 2 and 3 terms respectively [26]; and MGMF, a multilevel preconditioner [23]. From the tables, it can be seen that the CG(none) always has the highest Mflops rate but never the minimum time (although it is almost the fastest for the smaller 256 by 256 problem), due to the large number of iterations needed, especially for the larger 1024 by 1024 prob-

lem. On the other hand, the RIC method requires many fewer iterations but has limited parallelism, resulting in the worst execution time and Mflops rate. The polynomial preconditioners run at respectable Mflops rates but the number of iterations is still large, due to the lack of global coupling in the methods. Their run times are generally comparable to CG(none) for the problems tested. The multilevel preconditioner has the smallest run time of all, which is the result of a small number of iterations and being reasonably parallel (medium Mflops rate). These results help to illustrate that the hierarchical algorithms achieve a nice balance between degree of parallelism and computational efficiency.

6 Where Are the (Truly) Parallel Algorithms?

Before closing, I have a final remark on the design of parallel numerical algorithms in general. The traditional approach to algorithm design is to reduce (or approximate) a complex problem to (by) a sequence of simpler kernel problems. Examples of kernel problems are: basic linear algebra computations such as matrix-vector products and vector inner products, the solution of tridiagonal and triangular systems of equations, the fast Fourier transform, the computation of finite differences, etc. Thus, in Gaussian elimination, the solution of a general linear system is reduced to the solution of two triangular systems. In ADI methods, a discrete elliptic operator in multi-dimensions is approximated by a sum of one dimensional operators (often in the form of tridiagonal matrices).

Underlying this approach is the implicit assumption that these kernel problems can be solved efficiently. However, this depends on the computing model in which the problem is to be solved. Traditional kernel problems have been developed over many years and are primarily based on hand-computation and sequential computers. The advent of vector computers have required modifications in the specification of some kernel algorithms. I believe that, for massively parallel computers, we may need to re-examine the choice of kernels more fundamentally.

The solution of tridiagonal (and more generally narrowly banded) systems provides a good example. On sequential computers, tridiagonal systems can be solved in optimal time (proportional to the number of unknowns) and therefore is one of the simplest and most efficient way of providing global coupling in mathematical models. This fact alone has resulted in its wide-spread use in many algorithms in scientific computing. However, on a vector computer, it is difficult to solve tridiagonal systems using vector operations with the maximal vector length. One of the most efficient methods is cyclic reduction, which uses vectors with lengths ranging from $O(n)$ to $O(1)$, where n is the number of unknowns. Fortunately, many problems require solving many independent tridiagonal systems and this modified kernel can be vectorized efficiently. The situation is even worse on massively parallel computers, where tridiagonal systems are not so easy to solve any more. In fact, there is no known method for solving tridiagonal systems in time less than $\log n$, no matter how many processors one has. Thus on these computers, tridiagonal systems are really no easier to solve than say computing the FFT, a much more complicated kernel in the sequential case.

A similar situation holds for the solution of triangular systems: solving them efficiently on massively parallel computers require so-

phisticated and complicated algorithms [26]. A naive implementation can easily take as much computational time as the factorization process to obtain the triangular factors. Alternative methods for factorizing a matrix into more parallelizable factors have been proposed [27].

Should we re-examine the choice of kernels in traditional algorithms and abandon kernels that are not highly parallelizable and favor more parallelizable kernels which may have higher sequential computational complexity? Should we search for and use more hierarchical kernel algorithms? As an example, the standard implementation of a fast Poisson solver on a 2D rectangular domain uses FFTs in one dimension followed by solving tridiagonal systems in the other dimension. Mathematically, the second phase can also use FFTs but tridiagonal systems are usually preferred because they can be solved faster. On a parallel computer, should we use a FFT-FFT approach instead? There is evidence that a FFT-FFT approach may have the added benefit of producing more accurate results [7].

We can even go further and re-consider the discretization procedures used to construct the discrete models from the PDEs. Traditional methods can be classified as local (e.g. finite differences, finite elements, etc.) and global (e.g. spectral methods, boundary integral methods, etc.). The usual tradeoffs exist: the local methods are simpler but require many more degrees of freedom than the global ones. This again is a reflection of the existence of both local and global features in the PDE. Why not use hierarchical discretizations? Some research along this direction using wavelet basis has just begun [18]. These are exciting developments.

Hierarchical algorithms are not as easy as local algorithms to implement and this probably has caused some resistance in their acceptance into mainstream scientific computing. However, this situation may change with the advent of massively parallel computing. In a way, parallelism has forced us to think about the fundamental assumptions of algorithm design.

Acknowledgement: The author thanks Mr. Charles Tong for obtaining the data in Table 1 and for comments on the paper.

References

[1] O. Axelsson, *An algebraic framework for multilevel methods,* Report 8820, Department of Mathematics, Catholic University, The Netherlands, 1988.

[2] R. E. Bank, T. F. Dupont and H. Yserentant, *The hierarchical basis multigrid method,* Numer. Math. 52, pp. 427-458, 1988.

[3] R.M. Beam and R.F. Warming, *An Implicit Factored Scheme for the Compressible Navier-Stokes Equations,* AIAA J, 16, 1978, 393-402.

[4] J. H. Bramble, J. E. Pasciak and J. Xu, *Parallel multilevel preconditioners,* To appear in Math. Comp.

[5] A. Brandt, *Multi-level Adaptive Solutions to Boundary-Value Problems,* Math. Comp., Vol. 31, pp. 333-390, 1977.

[6] T.F. Chan, *The Physics of the Parallel Machines,* in "Opportunities and Constraints of Parallel Computing", J.L.C. Sanz (ed.), Springer Verlag, pp. 15-20, 1989.

[7] T.F. Chan and D. Foulser, *Effectively Well-Conditioned Linear Systems*, SIAM J. Sci. Stat. Comp., Vol. 9, No. 6, Nov. 1988, pp. 963-969.

[8] T.F. Chan, R. Glowinski, J. Periaux and O.B. Widlund, (eds.), *Domain Decomposition Methods*, SIAM, Philadelphia, 1989.

[9] T. F. Chan, Jay C.C. Kuo and C. H. Tong, *Parallel elliptic preconditioners: Fourier analysis and performance on the Connection Machine,* Computer Physics Communications 53, pp. 237-252, 1989.

[10] T.F. Chan and Y. Saad, *Multigrid Algorithms on the Hypercube Multiprocessor*, IEEE Trans. Comp., Vol. C-35, No. 11, Nov. 1986, pp.969-977.

[11] T.F. Chan and R. Tuminaro, *A Survey of Parallel Multigrid Algorithms*, in "Parallel Computations and Their Impact on Mechanics", AMD Vol. 86, A.K. Noor (ed.), The American Society of Mech. Engineers, New York, pp.155-170, 1988.

[12] T.F. Chan and R. Tuminaro, *Design and Implementation of Parallel Multigrid Algorithms*, in "Multigrid Methods: Theory, Applications and Supercomputing", S.F. McCormick (ed.), Lecture Notes in Pure and App. Math., Vol. 110, Dekker, New York and Basel, pp. 101-115, 1988.

[13] N.H. Decker, *On the Parallel Efficiency of the Frederickson-McBryan Multigrid*, Report 90-17, ICASE, NASA Langley, February 1990.

[14] I.S. Duff and G.A. Meurant, *The Effect of Ordering On Preconditioned Conjugate Gradients*, BIT (29) 1989, pp. 635-657.

[15] P. Frederickson and O. McBryan, *Parallel Superconvergent Multigrid*, in "Multigrid Methods: Theory, Applications and Supercomputing", S.F. McCormick (ed.), Lecture Notes in Pure and App. Math., Vol. 110, Dekker, New York and Basel, pp. 195-210, 1988.

[16] D. Gannon and J. van Rosendale, *On the Structure of Parallelism in a Highly Concurrent PDE Solver*, J. of Par. and Distr. Comp., Vol. 3, 1986, pp. 106-135.

[17] A. George, *Nested Dissection of a Regular Finite Element Mesh*, SIAM J. Numer. Anal., 10 (1973), pp. 345-363.

[18] R. Glowinski, W. Lawton, M. Ravachol and E. Tenenbaum, *Wavelet Solution of Linear and Nonlinear Elliptic, Parabolic and Hyperbolic Problems in One Space Dimension*, Technical report, AWARE Inc., Cambridge, Mass., 1990.

[19] R. Glowinski, G.H. Golub, G.A. Meurant and J. Periaux, *Domain Decomposition Methods for Partial Differential Equations*, SIAM, Philadelphia, 1988.

[20] L. Greengard, *The Rapid Evaluation of Potential Fields in Particle Systems*, ACM Distinguished Dissertation Series, MIT Press, 1988.

[21] V. Henson, *Wavelets and Multigrid*, talk presented at the 1st Copper Mountain Conference on Iterative Methods, April, 1990.

[22] D. Kuck, E. Davidson, D. Lawrie and A. Sameh, *Parallel Supercomputing Today and the Cedar Approach*, Science 231, pp. 967-974.

[23] C.-C. J. Kuo, T. F. Chan and Charles Tong, *Multilevel Filtering Elliptic Preconditioners,* UCLA CAM Report 89-23, August 1989. To appear in SIAM J. Matrix Analysis.

[24] S. McCormick, *Multilevel Adaptive Methods for Partial Differential Equations*, SIAM , Philadelphia, 1989.

[25] D.M. Nosenchuck, S.E. Krist and T.A. Zang, *On Multigrid Methods for the Navier-Stokes Computer*, in "Multigrid Methods: Theory, Applications and Supercomputing", S.F. McCormick (ed.), Lecture Notes in Pure and App. Math., Vol. 110, Dekker, New York and Basel, pp. 491-516, 1988.

[26] J.M. Ortega, *Introduction to Parallel and Vector Solution of Linear Systems*, Plenum, New York, 1988.

[27] J. Shanehchi and D. Evans, *Further Analysis of the QIF Method*, Int. J. Comput. Math., 11, 143-154.

[28] G. Strang, *Wavelets and Dilation Equations: a Brief Introduction*, SIAM Review, 31 (1989), pp. 614-627.

[29] C.H. Tong, *Parallel Implementation of the Multilevel Filtering Preconditioners on the Connection Machine*, talk presented at the 1st Copper Mountain Conference on Iterative Methods, April, 1990. UCLA CAM Report to appear.

[30] U. Trottenberg, *SUPRENUM - an MIMD Multiprocessor System for Multilevel Scientific c Computing*, Proc. CONPAR 86, Lecture Notes in Computer Science 237, Springer Verlag, Heidelberg, 1986, pp. 48-52.

[31] R. Tuminaro, *A Highly Parallel Multigrid-like Method for the Solution of the Euler Equations*, Tech. Report, Sandia Lab., Albuquerque, April 1990.