A SURVEY OF PARALLEL MULTIGRID ALGORITHMS

Tony F. Chan
University of California, Los Angeles
Research Institute for Advanced Computer Science, NASA Ames

Ray S. Tuminaro
Stanford University
Research Institute for Advanced Computer Science, NASA Ames

ABSTRACT

The multigrid algorithm is a fast and efficient (in fact provably optimal) method for solving a wide class of integral and partial differential equations. In addition, it is a natural choice for implementation on parallel computers because of the parallelism inherent in the algorithm. Over the past several years, there has been increasing research into parallel multigrid algorithms, ranging from purely theoretical studies to actual codes running on real parallel computers, some built with multigrid algorithms in mind. It is our goal in this paper to provide a brief but structured account of this field of research.

1. INTRODUCTION

The multigrid algorithm is a fast, efficient method for solving a wide class of integral and partial differential equations. This algorithm is now used in many areas of scientific computing (such as computational fluid dynamics and structural mechanics). The algorithm requires a series of problems be "solved" on a hierarchy of grids with different mesh sizes. For many problems, it is possible to prove that its execution time is asymptotically optimal. In addition to the strong asymptotic results, it has been demonstrated that a properly implemented multigrid code is competitive with other algorithms on meshes of a modest size.

The development of parallel computers leads one to search for parallel algorithms. Multigrid is a natural choice because of its success in serial computations and because of the parallelism inherent in the algorithm. Over the last several years, there has been increasing research into parallel multigrid methods, going from purely theoretical studies to actual codes running on parallel computers, to special computers built to facilitate running multigrid algorithms. While it is too early to write the final word on this subject, much experience has been gained and several ideas and trends have emerged. Although the literature is still scarce and not easily available, it is our purpose in this paper to provide a brief account of the state-of- the-art in this research area.

For the sake of brevity, we restrict our attention to multigrid algorithms on multiprocessor systems. For work on vector and pipeline computers, see [29] and [41]. Multigrid algorithms are also used in other parallel methods. For example, see Tang [44] for use of multigrid in parallel domain decomposition methods.

2. MULTIGRID ALGORITHMS

We give only a brief sketch of a typical multigrid algorithm applied to well-behaved linear elliptic partial differential equations (PDEs). More detailed descriptions of more general multigrid algorithms can be found in [5], [23], and [42]. An excellent introduction to multigrid can be found in [28].

Assume that a given elliptic partial differential equation is approximated by a discrete set of equations (finite differences or finite elements). The computational task (for which we use multigrid) is to solve the discrete set of equations. We denote these by

$$Au = b$$

where A is a matrix, b is a vector, and u is the vector of unknowns for which we seek the solution. It is useful when thinking about multigrid methods to consider the solution as a superposition of waves of different frequencies (say a Fourier decomposition). We will make reference to this frequency decomposition throughout the paper. A simple multigrid ("V" cycle) method consists of the following three steps. First, apply a relaxation scheme (i.e. an iterative method like Jacobi, or SOR) to an initial approximation. Typically, the difficulty with these relaxation schemes is that while they can efficiently reduce the error in a subspace of the frequency domain, they do poorly in another subspace. However, within multigrid the relaxation scheme must only reduce the errors associated with the high frequency domain. It is usually possible to find such a relaxation method when solving elliptic equations. Thus, after a few relaxation sweeps, the error in the current fine grid approximation is dominated by low frequency components. Since the error is smooth, it can be accurately approximated on a coarser grid. Thus step two is to form a correction equation (to solve for the error in the current approximation), project an approximation to this equation onto a coarser grid, and "solve" this system on the coarse grid. Finally, step three is to interpolate this coarse grid correction to the fine grid and add it to the fine grid approximation to get a new fine grid approximation. Since the coarse grid correction equation is only an approximation to the true correction equation, we must repeat the three steps until convergence.

It is important to notice that to solve the coarse grid equation one can use this same multigrid idea recursively. On the coarse grid we can perform relaxation to reduce the high frequency errors (which correspond to the middle frequencies on the fine grid). Project a correction equation on yet a coarser grid and continue.

We summarize this procedure with the following code segment.

```
proc multigrid(b,u,level,RelaxSweeps)
{
    if ( level = CoarsestLevel ) then $u = (A_{level})^{-1}b$
    else
        for k = 1 to RelaxSweeps do
            Relaxation(b,u,level)
        ComputeResidual(b,u,level,residual)
        ProjectResidual(level,residual,ProjRes)
        Multigrid(ProjRes,v,level+1,RelaxSweeps)
        Interpolate(level,v,correction)
        u = u + correction
    endif
}
```

There are many extensions and variations to the above algorithm. For example, there are "F" and "W" cycle algorithms which visit the coarser grids more often that the "V" cycle described above. Another variant is the Full Multigrid Algorithm (FMG), which systematically generates initial approximations to the solution on a given grid by recursively calling the multigrid procedure to obtain a solution on the next coarser grid and interpolating it up.

The convergence literature of multigrid methods is quite vast. The general idea is that high and middle frequency errors are reduced on the finest grid. This is due to the nature of the relaxation method. The lower frequency errors which are difficult to reduce on the fine grid are instead reduced on coarser grids (where they appear as higher frequency). We state without proof that under suitable conditions (usually obtainable for elliptic equations) it is possible to derive a multigrid method which converges in a constant number of iterations regardless of the mesh spacing, h. This property is quite extraordinary when one compares it with the O $(1/h)$ rate of convergence using SOR with optimal acceleration parameter on a two dimensional domain.

Though a rigorous analysis of the rapid convergence properties of the multigrid algorithm are beyond the scope of this article, we do present a simple data flow argument which can be used for obtaining lower bounds on convergence rates [12]. We assume that to solve an elliptic problem, each point within the interior of the domain must receive

(directly or indirectly) some information from all the boundaries of the domain. This is obvious when one considers that a change of the boundary conditions will change the solution over the entire domain. Consider the point SOR method applied to a standard 5 point discretization of Poisson's equation on a square. After k relaxation sweeps, each interior point receives information from all points that are within a distance k (where distance is measured by the minimum number of horizontal and vertical moves to adjacent points that must be performed to walk from one point to another). Thus only after $1/h$ iterations will all points in the interior receive some information from all the boundaries. Therefore, a lower bound on the rate of convergence for the SOR method is O $(1/h)$.

By contrast, within a typical multigrid algorithm each interior point receives information from all the boundaries in one iteration. Specifically, consider a multigrid iteration defined by performing one relaxation sweep on each grid level. Further define the coarse grid spacing on level, j, as $h_j = h_{j-1}/2$. Then if we use only a j-grid algorithm, each point receives information from all points that are a distance 2^{j-1} from it. This implies that if we coarsen down to a one point mesh, the update to any given point using the multigrid algorithm combines information from all the points inside the domain as well as the boundaries. Thus the required boundary information is received in one iteration. Therefore, a lower bound on its convergence rate is O (1) which is the actual convergence rate.

3. DESIGN AND ANALYSIS OF PARALLEL ALGORITHMS

In this section we present some criteria for designing and evaluating parallel algorithms. Though this task is complicated by a multitude of computer architectures, it is possible to describe some general principles that are valid on any parallel machine.

The main new task that faces the user of a parallel computer is how to decompose his problem so that all the processors will be as busy as possible. The ideal is to have each processor computing all the time. However, this ideal is rarely obtainable for a number of reasons. Communication overhead and load imbalancing are the two most prevalent and will be introduced in the next few paragraphs. A more detailed discussion of how these problems arise in multigrid will follow in later sections.

When utilizing a parallel computer, each processor typically has a separate subtask of the original problem. Due to the nature of most computational tasks, it is necessary for these subproblems to share information with each other. The degree to which this information sharing presents a problem is to some extent architecture dependent. Regardless of the architecture this sharing either directly or indirectly imposes delays (communication overhead) on the execution time of the program. In some systems (message passing) each processor has its own local memory and is interconnected to the others by some kind of network. To share information between two processors messages must be sent from one processor to another. Typically, the communication time varies depending on which two processors wish to communicate. Therefore, a given processor can communicate directly with those which it has a direct connection but must route messages through intermediate processors to communicate with others. Since this message passing represents overhead, the user should try and decompose his problem so that there is as little data sharing as possible. He should also map his problem so that most of the data sharing, occurs between processors which are close to each other. On other systems (shared memory) there is no direct penalty for sharing information between processors. All the processors share the same memory so that no explicit message passing is needed. However, in these systems there are delays associated with memory conflicts. These conflicts occur because the limited access to the memory that is allowed at a given time is exceeded by most parallel programs. To some extent these delays are beyond the control of the programmer.

Besides worrying about communication delays the user should consider load balancing when he partitions his problem among the processors. Specifically, a system is load balanced if all the processors have an equal load (amount of work) to perform. It is generally considered desirable to have the system as load balanced as possible. Typically, load imbalance implies that some processors are waiting idle for other processors to compute intermediate results that they need before they can continue. The more load balanced the problem is, the less time will be spent by processors being idle. Usually depending on the problem and the algorithm, this load balancing may or may not be easy.

To evaluate parallel algorithms we introduce two standard performance measures.

Let n be a measure of the problem size (for a PDE this might be the mesh size). Let p be the number of processors used to solve the problem and let $T(n,p)$ be the execution time for a problem of size n with p processors. The "speedup" in a parallel algorithm is the number of times faster that the parallel algorithm runs (with say p processors) than with just one processor. That is

$$S(n,p) = \frac{T(n,1)}{T(n,p)}.$$

Notice that the optimum speedup is $S(n,p) = p$. With this in mind we define the "efficiency", $E(n,p)$ of a parallel algorithm.

$$E(n,p) = S(n,p)/p.$$

Therefore the efficiency gives a measure of how well the entire machine is being used (with $E(n,p) = 1.0$ as the best).

It should be noted that the efficiency of an algorithm does not alone indicate the quality. What one is really interested in is a comparison of the execution time of the best serial algorithm against the execution time of the best parallel algorithm. Many times it is possible to identify an algorithm with a very high efficiency (for example using the Jacobi iterative method to solve a system of linear equations) that is poor because the corresponding serial algorithm is poor. In general, the best parallel algorithms (where by best we mean those which execute the fastest for a given problem) usually have good serial counterparts.

4. PARALLELIZING PDE ALGORITHMS

In this section we briefly discuss some techniques and considerations for writing one grid parallel PDE algorithms. We defer the discussion of multiple grid algorithms and of adaptive grids to later sections.

By far the most common way of assigning subtasks to each processor is by dividing the domain of interest into subdomains (one for each processor). Each processor is responsible for updating the variables associated with its subdomain only. There are two primary issues in this decomposition.

1. Choosing the size and shape of each subdomain.
2. Choosing which processor is assigned to each subdomain.

If the algorithm is to run on a shared memory machine, the processor assignment to subdomains is unimportant. However if an algorithm is to run on a message passing machine, the mapping of the domain to the architecture usually attempts to assign neighboring subdomains to processors that are directly connected to each other. The principle assumption is that the updates are local in nature (i.e., requires only information from neighbors). Many mappings which match adjacent subdomains to adjacent processors (or close to adjacent processors) for different kinds of machines have been studied. Some of these will be discussed in the next section in the context of the multigrid algorithm.

Determining the size and shape of the subdomains can be complicated if the PDE mesh is complicated. The hope is that by correctly choosing the size and shape of the subdomains, each processor will have an equal amount of work, and the amount of communication between processors will be kept at a minimum. The easiest situation is when the grid is uniform. In this case it is often assumed that the amount of computation is directly related to the size of the grid. Therefore, by dividing each of the subdomains into fairly equal sizes one feels reasonably assured that each processor has approximately the same amount of computation to perform. The situation is a little more difficult when the grid is not uniform or when there are local grids. These will be discussed later in the paper.

In addition to mapping the domain onto the architecture, the user must choose an algorithm such that the computations within the separate domains decouple as much as is possible. For example, consider solving a PDE on a square using the SOR method. Assume that we divide the square into subdomains by simply imposing a grid on top of the square. These subdomains are then assigned to different processors. In addition, assume that the PDE discretization corresponds to a five point stencil at the interior points. If we use a standard row-wise ordering (starting from the lower left corner) to perform the SOR updates, then the update at the point (i,j) must wait until the values at (i-1,j) and (i,j-1)

are both computed. This implies that a significant percentage of the processors will be idle waiting for relevant information to be computed. If, however, we use a red-black ordering of the points much more parallelism can be achieved. Specifically, we label all points (i,j) such that the sum of i and j is an odd number as red points and all other points are labeled as black points. Notice now that the updates for all the red points depend only on the values of the black points, and the updates for the black points depend only on the values of the red points. To implement a parallel red-black SOR solver (with one point per processor), we need only to repeat the following steps.

1. Simultaneously update all the black points.
2. Simultaneously communicate this information to the red points.
3. Update all the red points in parallel.
4. Simultaneously communicate this information to the black points.

Of course, to determine the value of each of the two algorithms one must also compare the rates of convergence as well. In general, the issues raised by the previous example are typical. In particular, often a small change to a standard numerical algorithm yields an algorithm which is still consistent with the problem and is much more parallelizable than the original algorithm. However, in comparing the two algorithms one must also assess the numerical properties of both algorithms. One class of such algorithms are the asynchronous methods [2]. The general idea is that there is no explicit synchronization for updating the solution values. Each processor simply applies the formulas to update its variables. In using the update formulas the current values of the variables are always taken. In the above example there would not be an explicit order of updating variables. Each processor updates its variables independently of the other processors.

Finally we mention that splitting the domain into subdomains is not the only way to parallelize a PDE code. One set of examples are the parallel FFT algorithms. Some other parallelization ideas with multigrid in mind can be found in [6].

5. MULTIGRID - PARALLELISM WITHIN EACH GRID

From the comments of the previous section we can conclude that it should be possible to perform in parallel the operations associated with any given grid in the multigrid algorithm. In particular, if a proper relaxation scheme is chosen (Jacobi, Red-Black SOR), it is possible to perform the relaxation, interpolation, residual calculation, and the restriction on any given grid in parallel. This is accomplished by performing a domain partitioning and a corresponding mapping to the computer architecture. The communication cost depends crucially on how this mapping preserves the locality of nearest neighbor subdomains on the fine grid as well as the coarser grids. Moreover, each grid within the hierarchy of grids in the standard multigrid algorithm must be processed sequentially and, therefore, on the coarser grids many processors could be idle. The effect of the mapping and the load imbalance on the parallel efficiency of the algorithm will be addressed in this section.

5.1 Theoretical Complexity Results

Before evaluating the performance of some parallel multigrid algorithms, it is appropriate to consider some theoretical complexity results for solving PDE's in parallel and for executing the multigrid algorithm.

First we derive the optimal asymptotic time for solving a general elliptic partial differential equation discretized with a mesh containing N grid points. We assume that the N grid points are partitioned among P processors (that is, each processor contains $\frac{N}{P}$ points). A lower bound can be derived by considering the time it takes to determine the solution at just one grid point in the domain. The primary assumption is that to solve an elliptic partial differential equation requires some information from all the points in the interior. This can be justified by looking at the global nature of the Green's function formulation of the solution. That is, the solution at any point in the domain can be written as an integral over the whole domain. In matrix terminology, this corresponds to a dense inverse of the operator, which implies that the solution is a linear combination of the right-hand side (or the forcing function) at all points within the domain. Thus, consider the time it takes to collapse information from N points into one point. The best that can be done within each processor is $O(\frac{N}{P})$, since each point must be visited. The optimal time to combine the P pieces of information (one value per processor) into one number is $O(\log P)$ (using a tree visiting method). Thus a lower bound on the time for solving an elliptic partial differential equation is $O(\frac{N}{P} + \log P)$. This result appeared in [12] and is consistent with [49] where more details can be found.

Notice that the above arguments were presented without any considerations of computer architecture. The primary assumption was that even though the information effecting the solution at a point is strongest in the local vicinity of that point, to solve the elliptic problem entirely required global information. Thus an important feature for a fast parallel method is that it must allow global information to reach all points in the interior rapidly (preferably in O(1) time). Of course the above arguments neglect the architecture at hand. If one has an algorithm which globally mixes information, it is equally important that the parallel architecture allows this global information to be transmitted rapidly (preferably in O(log P) time). By modifying the above arguments we can derive a lower bound that includes the communications topology (for a message passing system) given by $O(\frac{N}{P} + \log P + \mathit{diameter}(\mathit{architecture}))$, where the diameter of the architecture is the maximum distance between any two processors.

Multigrid is one of the few global methods. An update to a given point in the interior is based on global information from around the domain. Let us derive the asymptotic execution time of a "V" cycle multigrid algorithm when communication delays are ignored. We assume that the next coarser grid is defined from the current grid by using an h to 2h coarsening. We also assume that we continue coarsening down to a one grid point mesh. Finally, we assume that each processor is given a subset of $\frac{N}{P}$ grid points that it must update and that the relaxation scheme is fully explicit (i.e. all points can be relaxed at once - such as the Jacobi method). This implies that all the operations in the multigrid algorithm (e.g. interpolation, restriction, residual calculation, relaxation) can proceed in parallel. Let us, for simplicity consider only two dimênsional problems. To perform the operations on each grid in the hierarchy will take time proportional to the maximum number of grid points assigned to any processor. At each coarser level we have 1/4 as many points per processor as the previous level until we reach a situation where we have one point per processor for $\log_4 P$ levels (at which point we reach the one point grid). Performing the appropriate operation counts, we have

$$\mathit{work} = C \sum_{0}^{\log_4 \frac{N}{P}} \frac{N}{4^i P} + \sum_{2}^{\log_4 P} C = O(\frac{4}{3}\frac{N}{P} + \log P).$$

Since a proper multigrid algorithm converges in a constant number of iterations, we can state that the multigrid algorithm is asymptotically optimal in parallel as its execution time is identical to the lower bound derived earlier.

5.2 Communication and Architectures

In this section we consider computer architectures, corresponding domain-processor mappings, and the communication needs for implementing parallel multigrid algorithms. The comments in this section hold primarily for message passing systems. Our fundamental goal is to look at the data flow in the multigrid method and to determine architectures and mapping which support this data flow at a minimal communication cost.

The fundamental difficulty which makes multigrid methods more difficult to parallelize than simple relaxation schemes is the hierarchy of grids. Most of the difficulty occurs when the current grid in the multigrid process contains one point per processor and the next grid must be mapped to the architecture. To simplify the explanation we start by discussing a two-grid method where there is one point per processor on the fine grid. Most of the proposed schemes start by partitioning the domain into subdomains and assigning these to different processors for the fine grid operations. For the coarse grid there are two different general processor assignment strategies that have been advocated. The first is that a coarse grid point is assigned to the same processor as its corresponding fine grid point was assigned to. This approach seems natural and straightforward, but there are

some difficulties. In particular, when the fine grid contains only one point per processor, many processors are idle on the coarse grid. This follows from the fact that there are far fewer coarser grid points than fine grid points. Unfortunately, this implies that two neighboring coarse grid points are not necessarily in neighboring processors even if the neighboring fine grid points were. For example, consider a two dimensional processor mesh where each processor can communicate directly with its nearest neighbors. Further, assume that each processor contains one grid point. Then to implement the relaxation, interpolation and restriction on grid level k (where k=0 is the fine grid) requires communication over paths of length 2^k. Clearly this can lead to large communication delays on very coarse grids.

A second approach is to shuffle the grid points into different processors during a transition from one level to the next. Often it is possible to maintain the property that neighboring points are assigned to neighboring processors. In this way it may be possible to maintain relatively cheap communication costs. Once again the extent to which this is a problem depends on the problem size and the machine size.

We now describe a few of the architectures that have been advocated and their corresponding multigrid algorithms.

One such topology is the Perfectly-Shuffled-Nearest-Neighbor (PSNN) array studied by Grosch [22]. The architecture is defined in the following way. There are an odd number, n, processors. Processor i is connected to processors $i+1 \mathit{mod}\ n$, $i-1 \mathit{mod}\ n$, and to processor $2i \mathit{mod}\ n$. A multigrid algorithm with an h to $2h$ coarsening can be easily implemented on this topology. Suppose we have a one dimensional problem with n grid points on the fine grid to be solved using an n processor system. We map grid point i to processor i. Then we can shuffle all the odd points which define the coarse grid into neighboring processors. Specifically, we assign i (where i is odd) to processors $2i \mathit{mod}\ n$ and the coarse grid points are contiguous. We omit the details and simply state that this architecture can be generalized to higher dimensions. The PSNN architecture has many nice features. In particular it seems relatively straight forward to implement (i.e., there are a fairly small number of interconnections to be made). Second, it is possible to maintain nearest neighbor connections on coarser grids of the multigrid algorithm.

Another architecture considered by Chan and Schreiber [9] is in the form of a pyramid. One way to implement a multigrid process on this architecture is to assign to each level in the pyramid a different grid level. After the relaxation and residuals are computed on the fine grid, the points corresponding to the next coarsest grid level are shuffled to the next level of the pyramid. Once again this topology maintains nearest neighbor connections on all multigrid levels. Notice that while the communication overhead on this system is fairly minimal, one must process many multigrid problems at once to fully utilize the processors. Otherwise all pyramid levels except for one are idle at a given time. In their paper Chan and Schreiber derive asymptotic results for the speedup and efficiency of multigrid algorithms ("V" cycle, "W" cycle, and "FMG") on this architecture. Despite the presence of many idle processors, optimal asymptotic speedups can be obtained.

In Kolp and Miernedorff [30] they discuss multigrid implementations on two different computer topologies: nearest neighbor meshes and trees. We first discuss the nearest neighbor systems. In their paper they formalize the notation for arbitrary dimensional nearest neighbor systems applied to arbitrary dimensional problems. A b-dimensional nearest-neighbor system consists of a processor located on the cross points of a b-dimensional orthogonal grid. Each processor is coupled to those processors which are connected to it by a grid line. The easiest case is when the dimension of the machine is the same as the problem being solved. Here the domain is decomposed with a mesh that corresponds to the processor mesh. Neighboring domains are assigned to neighboring processors in the usual fashion. When the dimension of the domain and the processor system do not match, more complicated algorithms are given for performing a family of mappings which keep neighboring domains in neighboring or close to neighboring processors. The primary disadvantage with these nearest-neighbor networks is that the communication distances grow as we move to coarser and coarser grids. In fact, it is these growing communication needs which are the main restriction in these systems. Using their mappings they compute asymptotic speedup and efficiency results for the nearest-neighbor system. In addition, they determine algorithms which yield asymptotically optimal speedups. A tree structured topology is also considered in [30], together with a corresponding algorithm. The algorithm for mapping the subdomains essentially maps all the subdomains to the leaves of the tree in a way that will minimize nearest neighbor communication. Since all processing within the tree occurs at the leaves, many processors are not used in the computation. Computed asymptotic results are given for the tree architecture as well. In general, the path lengths by which nodes must communicate is not the primary restricting factor in performance. Instead, performance is limited by the overlapping of messages near the root of the tree (where the most traffic occurs).

Next, we consider the hypercube architecture which is probably the most studied of the ones presented in this section partly because many hypercube systems are commercially available. A p-dimension hypercube system contains 2^p processors. The interconnection topology is best described by assigning to each processor a unique p-digit binary number. There is a direct communication link between any two processors whose binary numbers differ in exactly one bit. Hypercube architectures have many favorable properties that are making them attractive in parallel computing circles. The primary advantage is that many different structures can be efficiently mapped to the hypercube graph. This includes rings, grids, and trees. In the context of multigrid there is a very nice property of the hypercube architecture that was pointed out by Chan and Saad in [10]. There is a mapping (specifically called the binary reflected gray code mapping) such that on the finer grids (grids where there is at least one point per processor) all communication for a basic multigrid algorithm is nearest neighbor. On the coarser grids (when there are fewer points per processor) all communication paths for the basic multigrid method are of length two. That is, even on the coarser grids the communication remains local. The main disadvantage with the hypercube is that it is probably the most difficult of the architectures mentioned in this section to build on a large scale (due to the large number of interconnections). However, despite these difficulties very large systems (up to 65000 processors) which are commercially available have been built. A few of the available hypercube computers include : Intel iPSC, N-cube, Connection Machine, Ametek's System 14, FPS's T-series and the Princeton/NASA Navier-Stokes Computer.

There has been much analysis as well as computer codes which have been written on hypercubes. We discuss in this section mainly those results which reflect on the communication aspects in the algorithm. Many studies have considered the size and shape of the general subdomains used in partitioning the region (see, for example [33], [34], [39], and [46]). The results differ somewhat depending on the machine but there are some general conclusions. For those machines that cannot perform parallel message transfers and where there is a large startup overhead cost in sending messages, it is usually preferable to divide the domain up into strips that go from one end of the domain to the other. The primary advantage of the strips is that only two sides of the region must communicate via messages. If, however, one has a system capable of parallel message transmission and where the message start up time is not too large, it is preferable to divide the domains up into boxes which are as square as possible.

In [14], Chan and Tuminaro developed timing models to predict the execution time and efficiency of a multigrid process as a function of hypercube communication parameters, grid size and cube size, taking into account communication costs. These models have been verified with computer codes on the Intel iPSC. Their general findings along with those of others [7] is that the greater the size of the problem as compared to the size of the hypercube, the better efficiency that can be obtained. Unfortunately, as the ratio of the size of the problem to the number of processors approaches one, the efficiency decreases. This effect is more noticeable for large processor systems than for smaller systems.

There have also been studies ([12], [39], and [46]) discussing the use of "V", "W", and "F" cycles in terms of efficiency on the parallel machine and overall execution time. An interesting point was raised by Naik and Ta'asan [39] concerning the comparison between the "V", "W" and "F" cycles. While it is true that the "V" cycle is better load balanced (it has better parallel efficiency) because it spends less time on the coarse grids, it is not necessarily the better method for solving a particular problem to a given accuracy because its convergence rate is often not as good as the "W" or the "F" cycles. In fact, in their numerical experiments (a driven cavity problem on a 128 by 128 grid), the "V" cycle is often the slowest, especially for small number of processors.

Other studies of implementation of multigrid algorithms on hypercubes include: Thole [46] where results of experiments on the Caltech Cosmic Cube are described, Thole [45]

where anisotropic problems are considered, Bassett [1] who applied a parallel multigrid method to a petroleum reservoir simulator on the Ametek System 14, Nosenchuck-Krist-Zang [40] who did a performance projection for using multigrid on a time dependent Navier-Stokes equation on the Princeton/NASA Navier-Stokes Computer, Van de Velde and Keller [48] who discuss a parallel implementation of a multigrid method with local grid refinements in conjunction with Schwarz's alternating procedure for a three-dimensional nonlinear elliptic problem. See also Hall-Salama-Lyzenga [24] and Cisneros [15].

Finally, we mention one additional architecture which is being built by the Suprenum project. This machine was designed with the intention that the multigrid algorithm would be one of its principle uses. The architecture uses a nearest neighbor mesh where each element in the nearest neighbor mesh is a cluster of processors. The intent in this design is that it represents a somewhat lower cost model than a hypercube but can still perform global communication operations relatively cheaply. More details on the architecture and multigrid implementations can be found in [3] and [26].

5.3 Load Balancing

The primary objective in load balancing is to divide the workload equally among the processors. On uniform meshes, it is relatively easy to divide the domain into partitions with approximately equal number of mesh points. Unfortunately, a load imbalance can still occur because the multigrid algorithms use a hierarchy of grids which must be processed sequentially. This leads to fewer points than processors on some grids and results in many idle processors.

Assume we have one point per processor, our finest grid is a p by p mesh, and we continue coarsening down to a one point mesh. Then the percentage of utilized processors is

$$\sum_{i=0}^{\log p} \frac{1}{4^i} \approx \frac{4}{3\log p + 3}.$$

A similar analysis for a three-dimensional problem shows that the percentage of utilized processors is about $8/(7\log p + 7)$. Thus, utilization varies inversely with the logarithm of the total number of available processors.

Notice that while the communication difficulties are primarily related to message passing systems, this idle processor situation is common to all parallel processing systems. Many studies have been conducted to understand the extent to which these idle processors degrade the parallel performance. See, for example, [7], [14] and [32]. It is clear that the crucial factor is the number of grid points per processor on the finest grid. That is, if the size of the finest grid results in many grid points assigned to each processor, then the loss in efficiency of the overall method due to idle processors is insignificant. The exact point where the idle processors are considered significant depends on machine parameters, particulars of the multigrid algorithm (say "V" cycle or "W" cycle), and the amount of computation per point. It seems fairly safe to say that when solving modest grid sizes (around 128 by 128) on small processor systems (say less than 100 processors), it is not necessary to be overly concerned with the idle processor problem. However, for large parallel machines (like the Connection Machine [32]), the idle processor problem could be significant.

We next list a few straightforward approaches to handle this idle processor problem which make only very minor changes to the basic multigrid algorithm. The simplest idea is to treat the one point per processor grid as the coarsest grid and to solve it via repeated relaxation. This is without a doubt the easiest to program (significantly easier than programming the standard multigrid algorithm where processors are allowed to go idle). Unfortunately, if there are many processors this implies that one is iterating on a very large grid. Often the relaxation procedures on large grids are very slow; in fact, so slow that this relaxation can seriously dominate the execution time of the algorithm. An alternative is to consider solving the coarsest grid problem inexactly by using only a few relaxation iterations. However, this can hamper the overall convergence rate of the multigrid iteration and cause a loss of the h-independent convergence. Other schemes advocate collecting all the information in one node and using some sequential solution technique on the one node. One nice aspect of this approach is that it is also a fairly simple algorithm to code (once again easier to program than the logic for the idle processors need in standard multigrid). One can implement more complex coarse grid solution techniques as the coarse grid is essentially solved sequentially. Another nice feature of collecting the information into one node is that it avoids the coarse grid communication difficulties of the standard multigrid algorithm. The main problem with these schemes is that they sacrifice a lot of parallelism (especially on large processor systems) by assigning what may be a large task (solving the coarse grid equation) to only one of the processors.

In general, all of the approaches mentioned above do not usually perform as well as simply continuing the standard multigrid process letting processors go idle on coarser levels. We should point out that the logic necessary to implement processors becoming idle and then active again is far from trivial, and in general on a small processor system one might wish to use one of the simpler schemes mentioned above.

It should also be pointed out that if many multigrid problems are being solved, it is possible to compute them simultaneously in a way that reduces the inefficiencies of the coarser grid levels. One method described by Brandt [6] will be illustrated using the solution of two one dimensional problems to be solved by a two-grid method. One approach is to perform the fine grid operations for each problem using all the processors in turn. Then project the coarse grid equation for one problem on half the processors and the other coarse grid equation on the other processors. An even simplier scheme is just to split the machine into two pieces and solve the two pieces independently. On the fine levels, the processors will contain two points and on the coarse levels each processor will have one point.

6. CONCURRENCY AMONG HIERARCHY OF GRIDS

We now discuss some alternative multigrid-like algorithms which are designed to more effectively attack the idle processor problem. The basic idea in all of these algorithms is to make use of the idle processors to carry out concurrent iterations on more than one grid. In this sense, they are significantly different from standard multigrid algorithms; in particular, their convergence behavior is often not well understood. Though these algorithms show promise there is no accepted method which is clearly superior. For simplicity in the descriptions that follow, it is assumed that there is one point per processor on the finest grid of a one-dimensional mesh. We only consider how to extend the algorithm to the next coarser mesh as the generalization to yet coarser meshes follow logically.

Gannon and Rosendale [20] were among the first to advocate such a concurrent iteration approach. Their method, Concurrent Iteration (CI), enables simultaneous iterations on all grids. We briefly sketch the idea behind their algorithm. During any given iteration, there is a variable u_i associated with each grid, G_i. At the end of each iteration the current approximate solution is given by

$$\sum_{k=coarsest}^{finest} u_k$$

where in order to simplify the notation we have omitted the interpolation operator. It is understood that each u_i is interpolated to the finest grid before the summation takes place. In addition, at each iteration the current residual is given by a summation. That is

$$residual = \sum_{k=coarsest}^{finest} p_k .$$

Essentially in one iteration the following occurs:

1. The current residual is redistributed among all the grids using restriction and interpolation operators. The general effect of this redistribution is to move the lower frequency errors to coarser grids.
2. Relaxation iterations occur on all grids.
3. A new residual (p_k's) is computed.

One can see that the basic idea of the algorithm is that coarse grids are used to solve for the low frequency errors while finer grids solve for the higher frequency errors. In their paper Gannon and Van Rosendale illustrate a convergence proof (without the strong h independent convergence of traditional multigrid). In addition, Chan and Saad [10] have shown how to map this algorithm to a hypercube keeping the communication needs local. Unfortunately, there are still many idle processors in a hypercube implementation. The

problem stems from the fact that, for high dimensional problems, the total number of points in the concurrent iteration scheme is not close to a power of 2. Since hypercubes usually contain 2^p processors, this implies that many processors are unassigned.

A similar idea was used by Greenbaum [21] and Swisshelm, Johnson, and Kumar [43]. Before any smoothing is done on the fine grid, the residual is projected onto each of the coarser grids and used as the right-hand side for systems of linear equations on each grid. These problems are then approximately solved concurrently on each individual grid via relaxation and collected to obtain a final solution. Greenbaum further proposed a method for collecting the coarse grid solutions. Each solution is combined with those on finer grids in such a way as to make the residual on the finer grid orthogonal to the interpolated approximate solution. If the matrix A is symmetric positive definite, this minimizes the A-norm of the error in the combined approximation.

A more recent algorithm is due to Frederickson and McBryan [19]. The idea is essentially to do multiple coarse grid corrections instead of one. For the one-dimensional problem one does two coarse grid corrections; half the processors do one coarse grid correction (which will correspond to all the odd grid points on the fine grid) and the other half compute the other problem (all the even points on the fine grid). The hope is that by combining multiple coarse grid corrections one will improve the convergence rate of the multigrid algorithm. Frederickson and McBryan demonstrated this algorithm by applying it to the Poisson equation (in 2 dimensions). First, they specify the restriction and prolongation as local stencils with constant unknown coefficients. Using these it is possible to derive an error expression for the resulting multigrid iteration. This error expression can then be used to determine optimal coefficients for the restriction and prolongation operators. Their results (using these operators) indicate a significant improvement over the standard multigrid method.

A related method has been proposed by Douglas and Miranker [16] and can be viewed as a generalization of the previous method in that it allows for multiple coarse grid equations. A main difference is that they propose to eliminate the relaxation sweeps. In their paper they derive sufficient conditions for the restriction and prolongation operators that correspond to convergence in one iteration. They illustrate with some examples on the Poisson equation as well as on a variable coefficient problem favorable convergence rates.

In both of these methods the choice of the restriction and prolongation operators is critical to the convergence rate. In fact if the wrong operators are used one can get slower convergence (or even divergence). In principle these restriction and prolongation operators determine how to split the problem into multiple pieces and then recombine then back into one solution. If one views multigrid methods from a frequency point of view (where essentially different grids are solving the solution in a different frequency range), then these methods are very different from standard multigrid. However, if one views multigrid as a method which uses the different grids to solve the solution in different subspaces, then these methods are not so different. The different coarse grid equations are in effect solving the residual equation in different subspaces. More research is needed to determine principles which indicate how inexpensive operators should be chosen to achieve rapid convergence.

Chan and Tuminaro [12] proposed another algorithm to tackle the idle processor problem. The basic idea is similar in spirit to that of Gannon and Van Rosendale [20]. Fine grids are used to reduce high frequency errors and coarse grids are used to reduce low frequency errors, in the spirit of the standard multigrid algorithm. Once again consider only a one dimensional problem with one point per processor on the finest grid. In this algorithm the residual is formed as in standard multigrid algorithms. This residual is then split into two components (i.e. two systems are to be solved). One component contains mostly the middle and high frequencies while the other component contains the low frequencies. The splitting is accomplished using local smoothing techniques to form the low frequency component and simply subtracting this from the original residual to get the other component. The system containing the low frequency component on the right-hand side is projected onto the coarse grid and is solved by recursively calling the multigrid procedure. The other system is "solved" concurrently using additional relaxation sweeps on the fine grid. Chan and Tuminaro have demonstrated that it is possible to generate superior convergence rates using this splitting principle when compared to standard multigrid algorithms. Unfortunately the algorithm is a little more difficult to program than some of the other algorithms mentioned. In addition, when analyzing the total execution time one must contrast the better convergence rate with the additional time taken per iteration due to the splitting step. Under this light it does not always pay to perform this splitting. We would like to mention that Kuo [31] considered a similar idea based on using digital filtering principles in the design of inter-grid transfer operators in multigrid algorithms.

The concurrent multigrid algorithms presented so far are designed for hierarchy of grids each of which covers the whole domain. However, multigrid algorithms can be applied to more general hierarchy of grids. For example, in adaptive mesh refinement algorithms, locally refined grids are often created. Concurrent iteration can also be carried out in such situations. Hart and MacCormick [25] proposed such an algorithm based on the fast adaptive composite grid method (FAC) [36]. FAC uses overlapping global and local uniform grids for the adaptive solution of PDE's. The simplest case to consider is when there is one global mesh covering the entire domain and one local finer grid covering a subregion. Basically one cycle of a two-grid coarse-to-fine FAC algorithm consists of the following steps : Use the current approximation to compute the residual on the composite grid. Solve a correction equation on the coarse grid defined by using a projection of the residual on the right-hand side. Use this correction to update the approximation, compute the composite grid residual, and solve a new correction equation on the local fine grid. Finally, update the current approximation using the new correction. Unfortunately, the sequential nature in solving the grids creates load balancing problems. For example, if a fixed mapping of subdomains to processors is done, then processors assigned to domains not containing local grid points will be idle when the local grid is processed. To alleviate this problem Hart and MacCormick [25] propose an asynchronous fast adaptive composite grid method (AFAC) which allows the local grids and the global grids to be processed in parallel. The basic idea is to split the current residual into components by projecting it onto different spaces (appropriate for the different grids). Then one iteration of a simple AFAC algorithm is to solve on each grid a corresponding system of equations with the appropriate projected residual component as a right-hand side. Then the sum of the interpolated solutions to these systems is taken as a correction to the current approximation. We omit the details of the efficient parallelization of this algorithm and simply state that the convergence rates of the AFAC algorithm are comparable with the FAC algorithm.

7. LOAD BALANCING FOR IRREGULAR MESH PROBLEMS

For the most part, we have to this point assume fairly uniform grids, for which it is relatively easy to find domain-processor mappings that achieve good load balancing. In general, non-uniform grids present a problem: How to partition the domain to keep the processors load balanced? The answer to this question in some sense depends on whether it is known beforehand what the grid densities and communication needs are or whether they are determined dynamically.

McCormick and Quinlan [35] list a few attributes that they consider desirable in assigning processors to grid points. They include:

1. Each processor should be assigned to exactly one connected subregion.
2. Each subregion should be rectangular (or as close as possible).
3. Each processor should be able to determine the subregion it is assigned to as well as the processors assigned to neighboring subregions.
4. Assignments should exploit the communication topology of the machine.
5. Balanced on both arithmetic and communication loads.
6. Dynamically executable to account for changing environment.
7. The assignment scheme should be parallelizable.

We now present a few methods. Once again there is no method that is clearly considered better than the others. More research in this general area is certainly needed.

Simulated annealing is a method that has been increasingly used in load balancing algorithms [17]. The basic idea is to set up a mathematical analog between an optimization problem and a variational characterization of the equilibrium state of a physical system, the solution of which can then be found by physically motivated and rapidly converging (the annealing) algorithms. For the load balancing problem, the objective function

could be chosen to consist of two terms: one to reflect the load imbalance and the other the communication cost. The main advantage of this approach is that it finds the optimal solution to both the load balancing and the communication problem. However, due to the combinatorial nature of the optimization problem, this is usually obtained at a considerable cost, especially when the number of partitions is large.

A somewhat less expensive approach is a class of methods that we shall describe as Alternating Direction Decomposition. In these methods, the main objective is to obtain a simple and systematic mapping that will balance the load. The main idea is to sequentially partition the domain in alternating coordinate directions.

The simplest method in this class is the recursive binary decomposition (RBD) method [4]. This method proceeds by first subdividing the domain into two parts by a partition along one coordinate direction such that the load is approximately equal in each subdomain. Then one partitions each of the newly created partitions along an orthogonal coordinate direction in a similar fashion. This procedure would then be repeated recursively until the correct number of subregions is produced. Load balance is obviously guaranteed but communication cost is not necessarily minimized.

One potential problem with the RBD method is that the resulting partitions are not well organized for determining one's neighbors. To overcome this problem, McCormick and Quinlan [35] proposed a related method, which they called the multilevel load balancing (MLB) scheme. The idea is to partition into more than two subregions in each coordinate direction at a time and stop the recursion after each coordinate direction has been visited once. For example, in two dimensions the result of this algorithm is an interconnection of rectangles which are grouped into strips. The MLB method is designed to be used in conjunction with the AFAC algorithm mentioned in the previous section.

An even simpler approach is the method of scattered decomposition [38]. The procedure is to map a small region of the domain onto the processors using a regular nearest neighbor mesh mapping and cover the whole domain with repeated copies of this processor mesh. The motivation is that if the granularity of the nonuniformity of the mesh is not smaller than that of the small region, then on the average each processor will get a similar load (e.g. number of mesh points.) The main disadvantage of this method is the higher communication cost due to the finer granularity of the mapping. The main attraction is its simplicity and general applicability - for example problems with local grids.

Finally, we discuss a partitioning due to Mierendorff [37] that specifically considers local grids. Suppose our problem consists of two grids : one covering the entire domain and the other covering the upper right quarter. The idea is to partition the entire domain not covered by the local grid over the entire machine. One should then partition the local grid over the entire machine. They show some examples of how this can be done maintaining nearest neighbor connections on nearest-neighbor machines. Using this total partition each processor is assigned a region of the local grid and of the entire domain. Thus, even when just the local grid is being processed, no processor is idle. We have indicated how the algorithm proceeds for one local grid. When there are many local grids one must split the grid sequence into a series of subsequences. Within each subsequence one fixed partitioning is used. However from one subsequence to the next there is a new partitioning and data is shuffled. The unfortunate difficulty with this scheme is that each processor is now assigned a series of little subdomains instead of just having one. This corresponds to more complex programming and logic.

8. IMPLEMENTATIONS

Many of the algorithms that have been described in this paper have been implemented on a variety of parallel computers. To date the authors are aware of multigrid implementations on the Caltech Cosmic Cube [33], [46], the Intel iPSC [7], [14], [18], [33], [39], the Ametek System 14 [1], the Connection Machine [32], and the now-extinct Denelcor HEP [34].

Other research efforts consider the software aspects of programming multigrid (and other numerical methods) on parallel machines. As was mentioned earlier, McBryan and Van de Velde [33] describe a machine dependent library of subroutines which they use to implement their multigrid algorithm. The library is such that the code which uses the library never needs to make explicit reference to machine dependent features. Therefore, another machine which contains the same library could run the same multigrid code.

Finally, we mention that while most of the codes discussed in this paper are used to solve model problems, more sophisticated problems are starting to be solved by parallel multigrid methods. For example, Tylavsky [47] has implemented the FLO52 code on an Intel iPSC hypercube. FLO52 is a well known and highly used code for fluid dynamics that was developed by Jameson [27]. The heart of the code contains a fairly sophisticated multigrid solver. Other codes are sure to appear.

9. ACKNOWLEDGEMENT

This work is in part supported by Cooperative Agreement No. NCC2-387 between NASA and the University Space Research Association (USRA) and by the Department Of Energy under contract DE-FG03-87ER25037.

10. REFERENCES

[1] Bassett, M., *An Implementation of Multigrid on a Hypercube Multiprocessor*, Proceedings of the First Conference on Hypercube Multiprocessors, Knoxville, TN, August 1985, M. Heath (ed), SIAM, Philadelphia, 1986, pp. 211 - 220.

[2] Baudet, G., *Asynchronous Iterative Methods for Multiprocessors*, Journal of ACM, Vol. 25, No. 2, April 1978, pp. 226-244.

[3] Behr, P., Giloi, W., and Muhlenbein, H., *Suprenum: The German Supercomputer Architecture - Rationale and Concepts*, Proceedings of the 1986 International Conference on Parallel Processing, K. Hwang, S. Jacobs, E. Swatzlauder (eds), IEEE Computer Society Press, 1987, pp. 567-575.

[4] Berger, M., and Bokhari, S., *A Partitioning Strategy for Non-Uniform Problems on Multiprocessors*, ICASE Report 85-55, NASA Langley Research Center, Hampton, VA, Nov. 1985.

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

[6] Brandt, A., *Multigrid Solvers on Parallel Computers*, Elliptic Problem Solvers, M. Schultz (ed), Academic Press, NY, 1981, pp. 39-84.

[7] Briggs, B., Hart, L., McCormick, S., and Quinlan, D., *Multigrid Methods on a Hypercube*, Proceedings of the Third Copper Mountain Conference on Multigrid Methods, S. McCormick (ed), Marcel Dekker, NY, 1987, (to appear).

[8] Chan, T. and Saied, F., *A Comparison of Elliptic Solvers for General Two-Dimensional Regions,* SIAM J. Sci. Stat. Comput., Vol. 6, No. 3, July 1985, pp. 742-760.

[9] Chan, T. and Schreiber, R., *Parallel Networks For Multi-grid Algorithms: Architecture and Complexity,* SIAM J. Sci. Stat. Comput., Vol. 6, No. 3, July 1985, pp. 698-711.

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

[11] Chan, T., Saad, Y., and Schultz, M., *Solving Elliptic Partial Differential Equations on Hypercubes,* Proceedings of the First Conference on Hypercube Multiprocessors, Knoxville, TN, August 1985, M. Heath (ed), SIAM, Philadelphia, 1986, pp. 196-210.

[12] Chan, T. and Tuminaro, R., *Design and Implementation of Parallel Multigrid Algorithms*, Proceedings of the Third Copper Mountain Conference on Multigrid Methods, S. McCormick (ed), Marcel Dekker, NY, 1987 (to appear).

[13] Chan, T. and Tuminaro, R., *Designing Parallel Multigrid Algorithms*, RIACS Technical Report 87.21, NASA Ames Research Center, Moffett Field, CA, Aug. 1987.

[14] Chan, T. and Tuminaro, R., *Multigrid Algorithms on Hypercube Processors,* Proceedings of the Second Conference on Hypercube Multiprocessors, Knoxville, TN, August 1986, M. Heath (ed), SIAM (to appear).

[15] Cisneros, A., *Irregular Regions and Multigrid Methods on the Hypercube*, Caltech

Concurrent Computation Project Memo, Pasadena, CA, 1987 (to appear).

[16] Douglas, C. and Miranker, W., *Constructive Interference in Parallel Algorithms*, SIAM Journal on Numerical Analysis, (to appear), also available as IBM Research Report RC 11742, 1986.

[17] Fox, G., *Load Balancing and Sparse Matrix Vector Multiplication on the Hypercube*, Caltech Concurrent Computation Project Memo 327, Pasadena, CA, 1986.

[18] Frederickson, P. and Benson, M., *Fast Pseudo-Inverse Algorithms on Hypercubes*, Proceedings of the Third Copper Mountain Conference on Multigrid Methods, S. McCormick (ed), Marcel Dekker, NY (to appear).

[19] Frederickson, P. and McBryan, O., *Parallel Superconvergent Multigrid*, Proceedings of the Third Copper Mountain Conference on Multigrid Methods, S. McCormick (ed), Marcel Dekker, NY, 1987 (to appear).

[20] Gannon, D. and van Rosendale, J., *On the Structure of Parallelism in a Highly Concurrent PDE Solver*, Journal of Parallel and Distributed Computing, Vol. 3, 1986, pp. 106-135.

[21] Greenbaum, A., *A Multigrid Method for Multiprocessors*, Proceedings of the Second Copper Mountain Conference on Multigrid Methods, S. McCormick (ed), Appl. Math. and Computation, Vol. 19, 1986, pp. 75-88.

[22] Grosch, C., *Performance Analysis of Poisson Solvers on Array Computers*, Report TR 79-3, Old Dominion Univ., Norfolk, VA, 1979.

[23] Hackbusch, W., *Multi-grid Methods and Applications*, Springer-Verlag, Berlin, 1985.

[24] Hall, J., Salama, M., and Lyzenga, G., *Solution of Large Matrix Equations: Multigrid Method and Parallel Processing*, Caltech Concurrent Computation Project Memo 260, Pasadena, CA, 1987.

[25] Hart, L. and McCormick, S., *Asynchronous Multilevel Adaptive Methods for Solving Partial Differential Equations on Multiprocessors*, Proceedings of the Third Copper Mountain Conference on Multigrid Methods, S. McCormick (ed), Marcel Dekker, NY, 1987 (to appear).

[26] Hoppe, H. and Muhlenbein, H., *Parallel Adaptive Full-Multigrid Methods on Message-Based Multiprocessors*,Parallel Computing, Vol. 3 , 1986, pp. 269-287.

[27] Jameson, A., *Solution of the Euler Equations for Two Dimensional Transonic Flow by a Multigrid Method*, Princeton MAE Rept. 1613, June 1983.

[28] Jespersen, D., *Multigrid Methods for Partial Differential Equations*, Studies in Numerical Analysis, G. Golub (ed), MAA Studies in Mathematics, Vol. 24, 1984, pp. 270-318.

[29] Johnson, G. and Swisshelm, J., *Multigrid for Parallel-Processing Supercomputers*, Proceedings of the Third Copper Mountain Conference on Multigrid Methods, S. McCormick (ed), Marcel Dekker, NY, 1987 (to appear).

[30] Kolp, O. and Mierendorff, H., *Efficient Multigrid Algorithms for Locally Constrained Parallel Systems*, Proceedings of the Second Copper Mountain Conference on Multigrid Methods, S. McCormick (ed), Appl. Math. and Computation, Vol. 19, 1986, pp. 169-200.

[31] Kuo, C., *Parallel Algorithms and Architectures for Solving Elliptic Partial Differential Equations*, M.I.T. Masters Thesis, January 1985.

[32] McBryan, O., *The Connection Machine: PDE Solution on 65536 Processors*, Parallel Computing (to appear).

[33] McBryan, O. and Van de Velde, E., *Matrix and Vector Operations on Hypercube Parallel Processors*, Parallel Computing, Vol. 5, Nos. 1 & 2, July 1987, pp. 117-125.

[34] McBryan, O. and Van de Velde, E., *The Multigrid Method on Parallel Processors*, Multigrid Methods II, Hackbusch, W. and Trottenberg, U. (eds.), Proceedings, Cologne, Oct. 1985, Lecture Notes in Mathematics No. 1228, Springer-Verlag, Berlin, 1986, pp. 232-260.

[35] McCormick, S. and Quinlan, D., *Multilevel Load Balancing for Multiprocessors - An Outline*, Proceedings of the Third Copper Mountain Conference on Multigrid Methods, S. McCormick (ed), Marcel Dekker, NY, 1987 (to appear).

[36] McCormick, S., and Thomas, J., *The Fast Adaptive Composite Grid (FAC) Method for Elliptic Equations*, Mathematics of Computation, Vol. 46, No. 174, April 1986, pp. 439-456.

[37] Mierendorff, H., *Parallelization of Multigrid Methods with Local Refinements For a Class of Non-Shared Memory Systems*, Proceedings of the Third Copper Mountain Conference on Multigrid Methods, S. McCormick (ed), Marcel Dekker, NY, 1987 (to appear).

[38] Morrison, R. and Otto, S., *The Scattered Decomposition for Finite Elements*, Caltech Concurrent Computing Project Memo 286, Caltech, Pasadena, CA, May 1985.

[39] Naik, V. and Ta'asan, S., *Performance Studies of the Multigrid Algorithms Implemented on Hypercube Multiprocessor Systems*, ICASE Report No. 87-5, NASA Langley Research Center, Hampton, VA, Jan. 1987.

[40] Nosenchuck, D., Krist, S., and Zang, T., *On Multigrid Methods for the Navier-Stokes Computer*, Proceedings of the Third Copper Mountain Conference on Multigrid Methods, S. McCormick (ed), Marcel Dekker, NY, 1987 (to appear).

[41] Ortega, J., and Voigt, R., *Solution of Partial Differential Equations on Vector and Parallel Computers,* SIAM Review, Vol. 27, No. 2, June 1985, pp. 149-240.

[42] Stuben, K. and Trottenberg, U., *Multigrid Methods: Fundamental Algorithms, Model Problem Analysis and Applications.* Multigrid Methods, W. Hackbusch and U. Trottenberg (eds), Proceedings, Koln-Porz, Nov. 1981, Lecture Notes in Math, No. 960, Springer-Verlag, Berlin, 1982, pp. 1-176.

[43] Swisshelm, J., Johnson, G., and Kumar, S., *Parallel Computation of Euler and Navier-Stokes Flows*, Proceedings of the Second Copper Mountain Conference on Multigrid Methods, S. McCormick (ed), Appl. Math. and Computation, Vol. 19, 1986, pp. 321-331.

[44] Tang, W. P., *Schwarz Splitting and Template Operators*, Stanford University, PhD Thesis, Stanford, CA, 1987.

[45] Thole, C., *A Short Note on Parallel Multigrid Algorithms for 3D-Problems*, GMD-Studien Report, GMD, St-Augustin, West Germany, 1987 (to appear).

[46] Thole, C., *Experiments with Multigrid on the Caltech Hypercube*, GMD-Studien Nr. 103, GMD, St-Augustin, West Germany, Nov. 1985.

[47] Tylavsky D., *Assessment of Inherent Parallelism in Explicit CFD Codes*, NASA Ames Research Report, Moffett Field, CA, 1987.

[48] Van de Velde, E., and Keller, H., *The Design of a Parallel Multigrid Algorithm*, Proceedings of the Second International Conference on Supercomputing, Vol. II, L. Kartashev and S. Kartashev (eds), 1987, pp. 76-83.

[49] Worley, P., *Problem Dissection and the Implications for Parallel Computation,* PhD Thesis, Stanford University (to appear).

reprinted from

Parallel Computations and Their Impact on Mechanics — AMD-Vol. 86
Editor: A. K. Noor
(Book No. G00379)

published by

THE AMERICAN SOCIETY OF MECHANICAL ENGINEERS
345 East 47th Street, New York, N.Y. 10017
Printed in U.S.A.

37-16