PARALLEL MULTILEVEL AND MULTIGRID METHODS

CRAIG C. DOUGLAS*

1. Preliminaries. Multigrid methods originated earlier this century, in the *personnel* computing era. Someone who needed to compute an approximation to the solution of a partial differential equation during that era would fill a room with people. After using very simple mechanical calculators to compute parts of the approximation, these people would pass their parts to the other people in the room who needed them. Except for the very different time scales and approximate solution accuracy, this process is similar to computing on today's distributed memory parallel computers.

The most basic model problem for elliptic boundary value problems in multigrid has always been, effectively, the two dimensional Poisson equation on a square.

$$-\Delta u = f \text{ in } \Omega = (0,1)^2 \quad ; \quad u = 0 \text{ on } \partial\Omega. \tag{1}$$

A $N \times N$ uniform mesh with mesh spacing $h = 1/(N-1)$ is placed over Ω as shown in Fig. 1.

The grid points are x_{ij}, where $1 \le i, j \le N$. A central difference discretization is applied to (1) to get a set of linear equations

$$AU = F, \tag{2}$$

where $F_{(i-1)N+j} = h^2 f(x_{ij})$ and $A = [-I, T, -I]$ is the block tridiagonal matrix defined using the $N \times N$ matrices I, the identity matrix, and $T = [-1, 4, -1]$, a tridiagonal matrix.

In the early part of the twentieth century, partial differential equations were approximately solved by relaxation techniques. (In fact, many types of equations are solved in this way today.) Solving (2) by a Gauss-Seidel method could take the rest of the personnel computers' lives if N was large enough and the required accuracy was strict enough. Hence, new tricks were needed to reduce the computation time.

During the 1920's, and probably several decades earlier, two–level schemes were used by engineers. An auxiliary grid Ω_1 was used to generate an initial guess to the solution to (2) on Ω_2.

Let $GS(j, n) = n$ Gauss-Seidel sweeps on level j. Algorithm 1, a *one–way two–level* algorithm is defined by:

* Mathematical Sciences Department, IBM Research Division, Thomas J. Watson Research Center, P. O. Box 218, Yorktown Heights, NY 10598, and Department of Computer Science, Yale University, P. O. Box 2158, New Haven, CT 06520. E-mail: *bells@watson.ibm.com* or *douglas-craig@cs.yale.edu*.

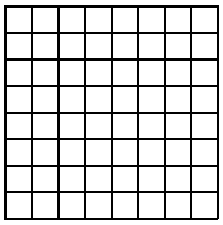

FIG. 1. *Uniformily meshed domain.*

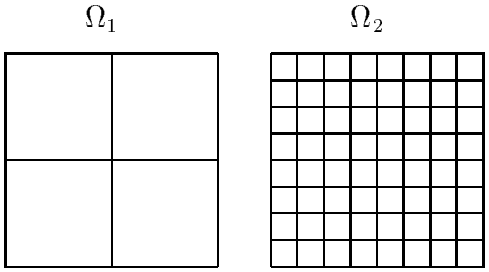

FIG. 2. *Meshes for the two–level schemes*

ALGORITHM 1.
(a) *Solve problem on* Ω_1 *by any means.*
(b) *Interpolate* U_1 *onto* Ω_2.
(c) *Do* $GS(2,\cdot)$ *until convergence.*

Bilinear interpolation was typically used in step (b). Let N_i be the number of grid points on grid Ω_i and define

$$\sigma_i \approx \frac{N_i}{N_{i-1}}, \quad i > 1.$$

Before digital computers were commonplace, σ_2 was typically between 5 and 10.

During the 1960's, Soviets developed what is now considered true multigrid. In this case, the auxiliary grid Ω_1 was used to correct approximations to the solution to (2) on Ω_2. Algorithm 2, a *two–level correction* algorithm is defined by:

ALGORITHM 2.
(a) $U_2 \leftarrow GS(2,n)$.
(b) $F_1 \leftarrow \sigma^2(F_2 - A_2U_2)$ *where* $\Omega_1 \cap \Omega_2$.
(c) $C_1 \leftarrow GS(1,m)$.
(d) $U_2 \leftarrow U_2 + Interpolate(C_1)$.
(e) *Repeat* (a)–(d) *until convergence.*

Projection of the style of step(b), known as *injection*, is in disfavor today; once again, bilinear interpolation has become typical. In this case, σ_2 is typically 4. (Note that this changes Ω_1 in the diagram.)

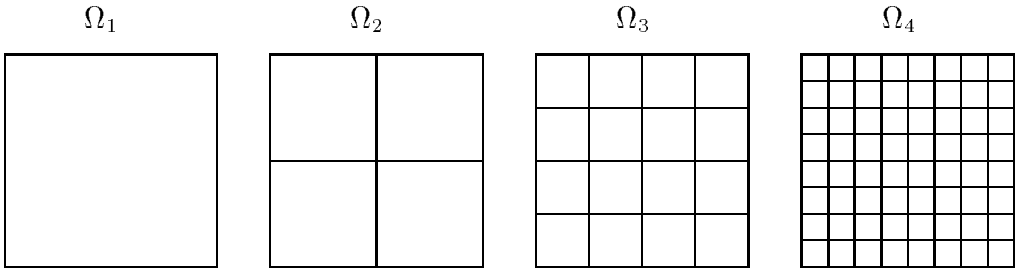

FIG. 3. *Multiple grid meshes*

Algorithm 2 is a notoriously slow algorithm for solving (2). The best trick for speeding it up is to use more than two grids. Define Ω_j, $j = 1, \cdots, k$ according to Fig. 3 and place uniform meshes Ω_j over Ω, with $h_j = \sigma^{k-j} h_k$, for some $\sigma \in \mathbb{R}$. Then solve

$$A_k U_k = F_k$$

using the U_j, $j < k$, in some prescribed manner.

Other tricks include

- Using different orders of interpolation (e.g., bicubics instead of bilinears) or projection to transfer information between meshes.
- Using better discretization methods (e.g., finite elements or volumes) for the partial differential equation.
- Solving the level 1 problem by some direct method.
- Using a more sophisticated smoother, such as conjugate residuals or CG-STAB.
- Starting the computation on level 1 instead of level k. (This is commonly referred to as a *nested iteration* in multigrid lingo.)

Thus, for a general linear differential equation on some domain Ω that has been discretized into a sequence of problems $j = 1, \cdots, k$,

$$A_j U_j = F_j, \quad U_j, F_j \in \mathcal{M}_j, \; A_j \in \mathcal{L}(\mathcal{M}_k). \tag{3}$$

The solution spaces $\mathcal{M}_j$ are typically function or vector spaces based on the discretization method (typically, finite differences, elements, or volumes) and $\mathcal{L}$ stipulates that A_j is a linear operator on $\mathcal{M}_j$. We assume that there exist mappings between the neighboring spaces:

$$R_j : \mathcal{M}_j \to \mathcal{M}_{j-1} \quad \text{and} \quad P_{j-1} : \mathcal{M}_{j-1} \to \mathcal{M}_j.$$

We also assume that there are mappings $Q_j : \mathcal{M}_j \to \mathcal{M}_{j-1}$ such that

$$A_{j-1} = Q_j A_j P_{j-1}.$$

This leads to Algorithm 3, a truly abstract correction algorithm of the following form:

ALGORITHM 3. $MG(j, \{\mu_\ell\}_{\ell=1}^{j}, x_j, F_j)$

(1) If $j = 1$, *then solve* $A_1 x_1 = F_1$ *exactly or by smoothing*

(2) If $j > 1$, *then repeat* $i = 1, \cdots, \mu_j$*:*

(2a) Smoothing: $x_j \leftarrow PreSmooth_j^{(i)}(x_j, F_j)$

(2b) Residual Correction:

$$x_k \leftarrow x_j + P_{j-1}\ MG(j-1, \{\mu_\ell\}_{\ell=1}^{j-1}, 0, R_j(F_j - A_j x_j))$$

(2c) Smoothing: $x_j \leftarrow PostSmooth_j^{(i)}(x_j, F_j)$

(3) Return x_j

This definition assumes that $\mu_1 = 1$. Common values for μ_j, $2 < j \leq k$, are 1 (*V cycle*) and 2 (*W cycle*).

The purpose of this article is not to analyze Algorithm 3 (see [4] for extensive analysis) but to discuss various methods for parallelizing it. Further, no mention is be made of nonlinear and/or nested iteration versions of Algorithm 3—the commentary applies to these cases verbatim.

There are two major themes in parallel multigrid today; telescoping and nontelescoping methods. Telescoping methods ($\sigma > 1$) include domain decomposition methods and a method which computes on all levels simultaneously. Nontelescoping methods use multiple coarse subspaces in which the sum of the unknowns on any level equals the sum on any other level.

2. Telescoping Parallelizations. Domain decomposition techniques offer a relatively easy path to parallelizing Algorithm 3. Each processor computes on the block of data per level that is assigned to it.

Because almost no work is required to get good convergence bounds, this is certainly the most attractive parallelizing technique from a theoretical standpoint. The method is merely a block iterative method used for smoothing, combined with standard multigrid. Hence, many standard multigrid convergence theorems apply verbatim.

Some communications and processor scheduling issues arise with this method. The amount of data communication involved and the location of the processors in a network will determine whether or not the problem is converted into an input–output problem rather than a computational one.

The factor σ_j (defined earlier) determines the telescoping of unknowns and thus plays a key part in processor scheduling. When the number of unknowns assigned to each processor falls below some threshold (which is a function of the problem, processor speed, and communications bandwidth and latency), some of the processors may be idle some of the time. To avoid this, an agglomeration of unknowns is typically performed and a certain number of processors become completely idle.

Having idle processors may seem like a waste of computer time on a parallel processor, but it actually is not necessarily the case. If the problem can be solved faster (as timed by a stopwatch) with some processors out of the computation some of time, then it is clearly a good approach, even if it is a bit wasteful. The principal reason for actually solving problems on parallel computers is that the results are perceived to be needed much sooner than just running many problems, one per processor.

As an aside, more and more parallel computing seems to be done on clusters of workstations (i.e., *distributed computing*) rather than on explicitly parallel machines. The workstation approach means that when a processor is out of the computation, its task scheduler can assign it to work on something else; thus, it may not actually be idle. This occurs naturally a parallel machine (e.g., Sequent), that has a reasonable operating system on each node of the machine.

During smoothing, information must be passed between processors along neighboring data regions. If a smoother like conjugate gradients is used, then information from dot products and matrix–vector multiplies will also clutter the communications' mechanism.

An asynchronous relaxation method would seem to be ideal in this environment. With this approach, each processor does its local relaxation method and uses the information obtained most recently from other processors. This is fast, and if the standard relaxation procedure converges, so does the asynchronous version, just not quite as quickly. Unfortunately, asynchronous relaxation methods have never caught on in the parallel multigrid community, mostly because the small number of smoothing iterations (2–6 iterations) makes it impossible to ignore changes in neighboring data.

Another approach to telescoping multigrid involves massively parallel computers (at least as many processors as unknowns on all of the meshes). Gannon and Van Rosendale [6] proposed what is referred to as a *concurrent multigrid* method. Typically, this means there should be $N \log_d N$ processors for a d dimensional problem.

The concept is that all operations should be performed simultaneously on all unknowns on all levels. An initial approximation of zero is assumed for the solution. Two sets of vectors, q_j and d_j, are used to hold information about right hand sides and data on the spectrum of levels in the sense that

$$F_k \approx \sum_{j=1}^{k} q_j + d_j.$$

A third set of vectors, x_j, contains the approximations to the solutions to each problem on each level. This information percolates to the finest level, k, to finally provide the approximate solution to the real problem. Algorithm 4, $GVRMG(k, \mu, F_k)$ is as follows:

ALGORITHM 4. $GVRMG(k, \mu, F_k)$

(1) *Initialize in parallel:*

$x_j = q_j = 0,\ 1 \le j < k\ ;\ q_k = F_k,\ x_k = 0.$

(2) *Repeat* $i = 1, \cdots, \mu$*:*

(2a) *Smoothing in parallel*

$x_j \leftarrow PreSmooth_j^{(i)}(x_j, q_j),\ 1 \le j \le k$

(2b) *Compute data corresponding to* x_j *in parallel:*

$d_j \leftarrow A_j x_j,\ 1 \le j \le k.$

(2c) *Compute residuals in parallel:*

$q_j \leftarrow q_j - d_j,\ 1 \le j \le k.$

(2d) *Project* q *onto coarser levels in parallel:*

$q_1 \leftarrow q_1 + R_2 q_2\ ;$

$q_k \leftarrow (I - P_{k-1} R_k) q_k\ ;$

$q_j \leftarrow (I - P_{j-1} R_j) q_j + R_j q_{j+1},\ 1 < j < k.$

(2e) *Inject* x *into finer levels in parallel:*

$x_1 \leftarrow 0\ ;\ x_k \leftarrow x_k + P_{k-1} x_{k-1}\ ;$

$x_j \leftarrow P_{j-1} x_{j-1},\ 1 < j < k.$

(2f) *Inject* d *into finer levels in parallel:*

$d_1 \leftarrow 0\ ;\ d_k \leftarrow d_k + P_{k-1} d_{k-1}\ ;$

$d_j \leftarrow P_{j-1} d_{j-1},\ 1 < j < k.$

(2g) *Put all the data back into* q *in parallel*

$q_j \leftarrow q_j + d_j,\ 1 \le j \le k.$

(3) *Return* x_k

This definition assumes that μ evenly divides twice the number of levels.

Algorithm 4 has a number of noteworthy aspects. First, communication between adjacent levels is twice the amount that would be expected. This is absolutely required in order for the consistency of the algorithm (without which it diverges).

Second, this algorithm limits what types of iterative methods that qualify as smoothers. The cost of one iteration on any level must be the same as the cost on any other level, assuming one processor per unknown. If the matrices A_j are similar enough to each other, Jacobi and conjugate gradients are fine, but Gauss-Seidel and SSOR are not. The latter two methods assume that the data is traversed in a particular order rather than all at once. Hence, the length of time to complete each iteration is dependent on the number of unknowns, violating the requirement of identical time per level.

Third, the number of levels that information must traverse to move from the finest level to the coarsest one and back is twice the usual number. Hence, all work estimates will be $O(2 \log N)$ (with the 2 being part of the constant) asymptotically in the number of levels, the usual multigrid method of estimating the problem complexity. This may seem high, but it is actually equivalent to the complexity of a standard V cycle.

3. Nontelescoping Parallelizations. Several competing methods have the same number of unknowns on each level. Each method has advantages and disadvantages.

The concept of using multiple subspaces to solve a problem whose solution lies in a particular space is hardly new. In fact, no one from the era in which it was invented is alive today. We will never know who really invented it, but we can be certain that it was introduced no later than in 1869.

Assume a rooted tree of problems ((3)) that are arbitrarily numbered. For a given problem k, it either has a set C_k of coarse space correction problems or it has none at all (i.e., $C_k = \emptyset$). When $C_k \neq \emptyset$, there are restriction and prolongation operators for each coarse space problem $\ell \in C_k$ such that

$$R_\ell : \mathcal{M}_k \mapsto \mathcal{M}_\ell \quad \text{and} \quad P_\ell : \mathcal{M}_\ell \mapsto \mathcal{M}_k.$$

We also assume that there are mappings

$$Q_\ell : \mathcal{M}_k \to \mathcal{M}_\ell \quad \text{such that} \quad A_\ell = Q_\ell A_k P_\ell.$$

These mappings are defined much as in the serial case.

A multiple coarse space correction multigrid scheme is defined by algorithm 5, $PMG(j, \mu_j, C_j, x_j, F_j)$, as follows:

ALGORITHM 5. $PMG(j, \mu_j, C_j, x_j, F_j)$

(1) *If* $C_j = \emptyset$, *then solve* $A_j x_j = F_j$ *exactly or by smoothing*

(2) *If* $C_j \neq \emptyset$, *then repeat* $i = 1, \cdots, \mu_j$*:*

(2a) *Smoothing:* $x_j \leftarrow PreSmooth_j^{(i)}(x_j, F_j)$

(2b) *Residual Correction:*

$$x_j \leftarrow x_j + \sum_{\ell \in C_j} P_\ell \; PMG(\ell, \mu_\ell, C_\ell, 0, R_\ell(F_j - A_j x_j))$$

(2c) *Smoothing:* $x_j \leftarrow PostSmooth_j^{(i)}(x_j, F_j)$

(3) *Return* x_j

The performance of any variant of Algorithm 5 is dependent mainly on how small δ_j is:

$$\|(I - \sum_{\ell \in C_j} Q_\ell^{-1} R_\ell)u\| \leq \delta_j \|u\|, \quad u \in \mathcal{M}_j. \tag{4}$$

This is a measure of how many of the error components in $\mathcal{M}_j$ are not completely represented in the subspaces.

Ta'asan [8] introduced this method to the multigrid community when standard multigrid failed to converge for a class of problems with highly oscillatory solutions. He uses standard interpolation and projection methods and a Kaczmarz relaxation method in his examples.

Usig a different set of interpolation and projection methods Hackbusch [7] developed a variant of Ta'asan's method using standard smoothers. Ta'asan's and Hackbusch's methods are both referred to as *robust multigrid*, which adds confusion (and heated discussions) to the field.

Frederickson and McBryan [5], using an approach different from that of Gannon and Van Rosendale, also investigated ways to keep all of the processors busy on a massively parallel single instruction, multiple data (SIMD) machine. They used standard interpolation and projection methods and an elaborate smoother on each level. Unless great care is taken, this method computes the correction in one of the correction spaces while the corrections in the remaining spaces add up (pointwise) to zero. Their method is referred to as *parallel superconvergent multigrid*. A comparison of this method with that of Gannon and Van Rosendale might make an interesting student exercise.

The methods of Ta'asan, Hackbusch, and Frederickson–McBryan all use interleaved grids. For a problem on a square, this translates into the following, where the numbers refer to which subproblem the unknowns belong:

$$\begin{array}{cccccccc} 3 & 4 & 3 & 4 & 3 & 4 & 3 & 4 \\ 1 & 2 & 1 & 2 & 1 & 2 & 1 & 2 \\ 3 & 4 & 3 & 4 & 3 & 4 & 3 & 4 \\ 1 & 2 & 1 & 2 & 1 & 2 & 1 & 2 \\ 3 & 4 & 3 & 4 & 3 & 4 & 3 & 4 \\ 1 & 2 & 1 & 2 & 1 & 2 & 1 & 2 \end{array}$$

(The motivation is similar to that for multicolored orderings for standard iterative methods.) Each of these methods requires that all of the matrices associated with the spaces be generated, except in trivial cases, thus doubling the memory requirements expected for solving boundary value problems. In addition, the coarse space operators are more difficult to compute using Hackbusch's variant than with either of the other two methods. In general, these are all space–wasteful methods.

A fourth, very different approach that my colleagues and I developed, is referred to as either *constructive interference* or, more recently, *domain reduction* (many references can be found in[3]). Our original motivation for using a multiple coarse space parallel multigrid algorithm was to eliminate the smoothing step from standard multigrid algorithms. Smoothing takes most of the computational time but contributes almost nothing to the convergence rate, whereas coarse grid corrections take little time and reduce the error substantially. A general theory and simple examples were developed for multiple coarse space methods was using no smoothing on the fine grid and mutually orthogonal subspaces which covered all of the error components of the original space (thus, $\delta_j = 0$ in (4)). This leads to very efficient direct methods rather than the expected iterative ones.

A side benefit of this theory is that the fine grid problem and, if a trick is used, most of the coarse space matrices do not need to be generated. This method can use substantially less memory than a standard iterative or multigrid algorithm.

An additional note about domain reduction is that it leads naturally to more than 2^d subspaces for a d dimensional problem. An eight way decomposition of a problem on a square can be constructed, leading to problems defined on squares, rectangles,

and triangles. Both 60– and 64–way decompositions of a problem on a cube can be constructed with moderate difficulty. In theory, a 192–way decomposition of a problem on a cube is possible. The entire problem would be solved 2660 times faster, if each of the 192 subproblems were solved by sparse Gaussian elimination, than if the original problem is solved by the same method (the latter is not advised, however).

To see how each of the variants operates, consider various projection operators in one dimension (where $x_0 = c_{N+1} = 0$):

- Linear projection: for $1 \leq i \leq N$,

$$y_i = x_{i-1} + 2x_i + x_{i+1}.$$

  Let one subspace consist of the odd numbered y_i's and another subspace for the even numbered y_i's. This is used by both Ta'asan and Frederickson–McBryan.
- Linear–linear orthogonal complement projection: for $1 \leq i \leq N$,

$$y_i = \begin{cases} x_{i-1} + 2x_i + x_{i+1} & i \; even, \\ \\ -x_{i-1} + 2x_i - x_{i+1} & i \; odd. \end{cases}$$

  The subspaces are defined as in the linear projection case. The values at odd numbered y_i's correspond to the orthogonal complement of the traditional space defined by linear projection. This is used by Hackbusch.
- Symmetric–antisymmetric projection: for $1 \leq i \leq N/2$,

$$y_i = \frac{x_i + x_{N-i+1}}{2} \quad and \quad \hat{y}_i = \frac{x_i - x_{N-i+1}}{2}.$$

  One subspace consists of y_i's and the other of $\hat{y}_i$'s. This defines a domain folding (or reduction) where even and odd functions are annihilated in exactly one subspace and exactly reproduced in the other. This is the method that my colleagues and I used.

The two dimensional definitions of the above are defined in the obvious manner using tensor products.

Unfortunately, these methods have not been directly compared on a nontrivial example problem. The closest is a simple problem from the literature (I apologize in advance to each originator of a method compared here):

$$\begin{cases} -10^5 u_{xx} - 10^{-5} u_{yy} &= f \text{ in } (0,1)^2, \\ \\ u &= 0 \text{ on } \partial(0,1)^2. \end{cases}$$

To make things comparable with known results, a uniform mesh is used, a standard (simple) discretization, the energy norm, one Jacobi iteration in the analysis for multigrid (MG) and robust multigrid, and a direct solve on the coarsest level(s). The contraction factors are:

Method	Contraction factor	
MG	0.97	(17×17 grid)
Robust MG	0.33	(h independent)
Domain reduction (iterative)	ϵ	(h independent)
Domain reduction (direct)	0.00	(h independent)
Parallel superconvergent	"small"	(h independent)

The solver in the domain reduction is either iterative (solving each problem to an accuracy of ϵ) or direct. If the smoother called for in parallel superconvergent can be constructed, then the contraction factor missing from the table will be very small, on the order of 0.05. Note that a line relaxation method, instead of point Jacobi, would make multigrid work well.

4. Implementing Parallel Multigrid. I have been distributing a public domain multilevel, aggregation-disaggregation code (MADPACK[2]) for some years. MADPACK is really a linear algebra package, rather than a package designed specifically for partial differential equations. The user specifies the domain or differential operator to MADPACK indirectly, not indirectly. Interpolations and projections are computed as matrix–vector multiplies. Several sparse matrix formats are allowed including a stencil format which is very efficient on regular meshes.

MADPACK has a sparse matrix–vector (and matrix transpose–vector) multiplier, a sparse direct solver, and three smoothers, namely, Gauss-Seidel, conjugate gradients, and Orthomin(1). The latter two are preconditioned by SSOR. It is quite compact.

As an experiment, I parallelized parts of it three years ago in the spirit of finding out how painlessly it could be done. Since I wanted to run my code on distributed memory (Intel iPSC2) and shared memory (Sequent Symmetry) parallel processors, as well as a network of workstations with a minimal amount of code changes, I started from the C version of MADPACK and adapted it to the Linda system[1]. (I refuse to program in any computer's assembly language, so why should I program my communications in exactly that sort of crude environment?)

First, I realized that the obvious approach of storing matrices by column strips and vectors by row strips meant that the sparse matrix–vector multiplies were trivial to implement in parallel. I then replaced the three smoothers by two—diagonally preconditioned conjugate gradient and Orthomin(1)—which added a parallel dot product routine. Then I load balanced the operating processors by maintaining the same number of unknowns per processor independent of the level (effectively agglomerating at each level). Finally, I made absolutely sure that if I was doing a direct solve on the coarsest level, that I had only one processor involved. (The point of this exercise was to learn something about implementing something substantial in parallel and get something running quickly, not to produce a product quality code.)

The good news is that it was reasonably efficient in all three machine environments. The first implementation (on the Sequent) took a long time; the second (on the Intel iPSC2) , which unfortunately required porting to the distributed memory environments, took an afternoon. With twenty minutes more work, I also had a simple three

dimensional domain reduction example running (with 99% parallel efficiency) that produced publishable results. Assuming there was really enough data associated with each active processor to keep them all active computationally, the parallel efficiency could be kept in the 75%–99% range for most problems tried, even accounting for idle processors.

The bad news is that the differential equation front end to MADPACK had to determine how to break natural data objects like vectors and sparse matrices into strips. This was not nearly as painful as had been expected, but introduced many subtle bugs into the process that had to be found and fixed.

There are a number of parallel multigrid codes in existence besides this one. In particular, the Suprenum project produced a collection of interesting codes. To the best of my knowledge, there are no public domain parallel multigrid codes for nontrivial problems, nor any available for anonymous ftp on the Internet[2].

5. Conclusions. Parallel multilevel methods, which originated earlier this century in the personnel computing era, were the natural precursors to standard multilevel methods. Parallel multilevel methods are also the natural successors, in the age of advancing computers, to standard multigrid methods. What makes six parallel multilevel methods practical and impractical was discussed in the context of the three algorithms that encapsulate them. Because no one has carefully compared all of these methods, on a collection of common problems, on a variety of machine architectures, it is difficult to determine the conditions in which a particular method is really the right or wrong choice.

Using high level programming tools makes implementing these methods much easier, although nontrivial. The use of low level tools, such as explicit message passing methods, has been dismissed as a waste of human time. Whether or not high level tools are used, parallel multilevel methods scale well assuming there is enough data to make using a parallel computer worthwhile.

REFERENCES

[1] N. Carriero and D. Gelernter, *Linda in context*, Comm. ACM, 32 (1989), pp. 444–458.

[2] C. C. Douglas, *Mgnet Digests and Code Repository*. Monthly digests subscribed to by sending a message to mgnet-requests@cs.yale.edu and an anonymous ftp site (casper.cs.yale.edu) for codes and papers on multigrid and related topics.

[3] ——, *A tupleware approach to domain decomposition methods*, Appl. Numer. Math., 8 (1991), pp. 353–373.

[4] C. C. Douglas and J. Douglas, *Towards a general convergence theory under few assumptions for abstract multilevel algorithms*, in Preliminary version of the Proceedings of the Fifth Copper Mountain Conference on Multigrid Methods, S. F. McCormick and J. Mandel, eds., Computational Mathematics Group, University of Colorado at Denver, Denver, Colorado, 1991. To appear in SIAM J. Numer. Anal. as *A unified convergence theory for abstract multigrid or multilevel algorithms, serial and parallel*, 29 (1992) or 30 (1993).

[5] P. Frederickson and O. McBryan, *Parallel superconvergent multigrid*, in Multigrid Methods: Theory, Applications, and Supercomputing, S. F. McCormick, ed., Marcel Dekker, New York, 1988, pp. 195–210.

[6] D. Gannon and J. V. Rosendale, *On the structure of parallelism in a highly concurrent PDE solver*, J. Par. and Dist. Comp., 3 (1986), pp. 106–135.

[7] W. Hackbusch, *A new approach to robust multi–grid solvers*, in ICIAM'87: Proceedings of the First International Conference on Industrial and Applied Mathematics, Society for Industrial and Applied Mathematics, Philadelphia, 1988, pp. 114–126.

[8] S. Ta'asan, *Multigrid Methods for Highly Oscillatory Problems*, PhD thesis, Weizmann Institute of Science, Rehovot, Israel, 1984.