A TUPLEWARE APPROACH TO DOMAIN DECOMPOSITION METHODS*

CRAIG C. DOUGLAS†

Abstract. Domain decomposition methods are highly parallel methods for solving elliptic partial differential equations. In many domain decomposition variants, the domain is partitioned into a number of (possibly overlapping) subdomains before computation begins. In contrast, the domain reduction method folds the domain into a number of smaller domains covering only a small portion of the entire domain. The solution over the entire domain is recovered by unfolding the solutions on the subdomains and summing them. The cost of the folding and unfolding is negligible.

Results are derived measuring how much time and space can be reduced in the problem discretization phase by using these methods. Storage can be reduced by a substantial factor (a factor of eight is constructed for an example in this paper), and discretization time can be reduced by a constant factor or more, depending on the time complexity formula for a given problem. The amount of storage required is substantially less than for traditional domain decomposition implementations, or ones based on standard iterative methods or multigrid (standard or nontelescoping).

A highly portable implementation using the Linda system for parallel or distributed programming is discussed. Linda adds a tuple data abstraction to a language, which is used to develop numerical software.

Key words. domain decomposition, multigrid, projection method, iterative methods, direct methods, Linda system

AMS(MOS) subject classifications. 65F50, 65F05, 65F10, 65W05

1. Introduction. In this paper, the solution to an elliptic boundary value problem is approximated using a combination of domain decomposition, multigrid, and projection method techniques (see [2], [4], [9], [12], [18], [19], [22], and [23]).

Consider problems of the following form.

$$\begin{cases} \mathcal{L}u = f & \text{in } \Omega \subset \mathbb{R}^d, \quad d > 0, \\ u = 0 & \text{on } \partial\Omega. \end{cases} \tag{1.1}$$

More complicated boundary conditions are discussed in [5], [13], and §4. That (1.1) is well posed and will be discretized by a finite element, volume, or difference scheme is assumed throughout this paper. Typically, $\mathcal{L}$ is a general, variable coefficient second or fourth order elliptic problem. Similar theory for nonlinear problems can be found in [1].

The domain reduction method uses properties of a partial differential equation to split the problem into several subproblems. The subdomains are constructed by folding the domain in particular ways, some of which are described in §§3–4. Each subproblem should be solved in parallel using the fastest known appropriate solution method.

The basic method was first derived in [14] as a parallel multilevel method for linear systems of equations where smoothing was unnecessary. The applicability to shared or distributed memory, coarse or fine grained parallel computers was discussed

* Based on IBM Research Report RC 15360, Yorktown Heights, NY, 1990. To appear in the second special issue of *Applied Numerical Mathematics* on domain decomposition methods (1991 or 1992).

† 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 Yale Station, New Haven, CT 06520. E-mail: *bells@watson.ibm.com*

informally in [15]. The elliptic partial differential equation case (with an emphasis on one and two dimensions) was analyzed in [16], and a technique for doubling the basic parallelism was introduced in [5]. This was extended to three dimensional problems in [13]. In the general case, or when the subproblems are solved only approximately, the method becomes an iterative method or can be used as a preconditioner. Bounds on the resulting convergence factors and condition numbers are given in [13].

In §2, the method is defined as it pertains to this paper in a very abstract manner. In §3, a simple example is worked out when $\Omega = (-1,1)^d$. By folding the domain in half in a particular manner, a simple 2^d way domain reduction is defined.

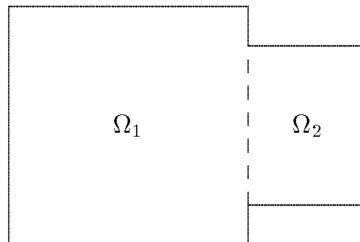
In §4, more complicated domain reductions are defined. Results are derived measuring how much time and space can be reduced (with respect to standard methods without reduction) in the problem discretization phase by using these methods. Storage can be reduced by a substantial factor, and discretization time can be reduced by a constant factor or more, depending on the time complexity formula for a given problem.

In §5, the Linda parallel programming environment is described briefly. Linda adds an abstract tuple data type to a language, which is used to design numerical software (tupleware) and to improve a domain decomposition method.

In §6, an implementation of the ideas of this paper is described. We show that we can easily and naturally solve (1.1) numerically by storing the coefficient matrices associated with various discretizations using much less space than required to store the solution on the entire domain. Given a fixed amount of usable storage on a given computer, this technique allows much larger problems to be solved. This is not an obvious result, and has not been noted in earlier papers on domain reduction methods.

There are hundreds of site licenses for Linda today, and it is available from a number of sources, some commercial and others academic. This is enough to warrant getting the numerical community interested in this style of parallel programming. It allows a quite different style of implementing sparse matrix software, and offers some advantages which are not immediately obvious. The reason why §6 works is that Linda provides a natural mechanism for implementing very complicated data structures easily, efficiently, and portably for the size problems numerical analysts are interested in.

The domain reduction method cannot be applied to all known problems directly. The techniques of §§2–4 would be difficult to apply easily to a problem defined on the following domain.



The obvious domain decomposition is the following.

If possible, a domain reduction technique should then be used to solve the problems that arise by using a standard domain decomposition method on each of Ω_1 and Ω_2 instead of a preconditioned conjugate direction method or multigrid. Note that the subproblems which naturally arise in the domain reduction method can be solved by parallel variants of preconditioned conjugate direction methods or multigrid.

2. Abstract Formulation. In this section, the domain reduction method is defined in three algorithms. The first two are special cases of the quite general third. Some relevant convergence results are stated with appropriate references.

A variational formulation of (1.1) is

$$\text{find } u \in V \text{ such that } a(u,v) \;=\; f(v),\ \forall v \in V. \tag{2.1}$$

Typically, $V = V(\Omega)$ is an appropriate Sobolev space, and

$$a(u,v) \;=\; \int_\Omega a \bigtriangledown u \bigtriangledown v \;+\; a_0 uvdx \quad \text{and} \quad f(v) \;=\; \int_\Omega fvdx.$$

The method may be formulated quite abstractly.

Algorithm 1. *To solve (2.1), subspaces $V_1, V_2, \ldots, V_n$ are found such that*

$$V \;=\; \bigoplus_{i=1}^{n} V_i \quad \textit{and} \quad V_i \perp V_j,\ i \neq j,$$

in the $a(\cdot,\cdot)$ inner product. Then the solution is

$$u \;=\; \sum_{i=1}^{n} u_i,$$

where u_i is the solution of the subproblem,

$$\textit{find } u_i \in V_i \textit{ such that } a(u_i,v) \;=\; f(v),\ \forall v \in V_i. \tag{2.2}$$

In reality, the cost of solving (2.2) by direct methods is often unreasonable. A more realistic definition of the method can be formulated.

Algorithm 2. *For an approximate solution u^m of (2.1), form the subproblems*

$$\textit{find } w_i \in V_i \;:\; a(w_i,v_i) \;=\; f(v_i) \;-\; a(u^m,v_i),\ \forall v_i \in V_i, \tag{2.3}$$

and find approximate solutions $\bar{w}_i \in V_i$ such that

$$\|\bar{w}_i - w_i\| \;\leq\; \varepsilon \|w_i\|, \tag{2.4}$$

where $\|\cdot\|$ is the energy norm and $\varepsilon < 1$ is a constant. Then set

$$u^{m+1} \;=\; u^m \;+\; \sum_{i=1}^{n} \bar{w}_i.$$

Note that step (2.4) can be realized by solving (2.3) by an iterative method with

convergence factor at worst ε in the energy norm, starting from an initial guess of zero. If more than one iterative method is used, then ε is the maximum of the convergence factors. If the subproblems are solved exactly, the method is related to the well-known Schwarz alternating method [27].

In implementation, full-rank restriction and prolongation operators are used to transfer information between the subspaces and V:

$$\mathcal{R}_i : V \rightarrow V_i \quad \text{and} \quad \mathcal{P}_i : V_i \rightarrow V.$$

Frequently, each prolongation operator is the adjoint of one of the restriction operators. Writing the problem (2.2) as $\mathcal{L}u = f$, the corresponding operators for the subproblems are then defined in the Galerkin sense:

$$\mathcal{L}_i \;=\; \mathcal{R}_i \mathcal{L} \mathcal{P}_i.$$

The operator $\mathcal{L}$ is discretized into a matrix $\mathcal{A}$. The subproblems, $\mathcal{A}_i$, are defined in the Galerkin sense, as well. Examples of several choices of restriction and prolongation operators, and the resulting $\mathcal{L}_i$ and $\mathcal{A}_i$, are given in [16].

The complete discrete domain reduction method is defined in a two level (multigrid-like) formulation.

Algorithm 3.

Smooth s times on u to get u^0
Do $j = 1, \cdots, m$ \{
  Compute residual $r^j = f - \mathcal{A}u^{j-1}$
  Solve in parallel, $i = 1, \cdots, n$:
    $\mathcal{A}_i c^i = \mathcal{R}_i r^j$
  Set $c = u^{j-1} + \sum_{i=1}^{n} \mathcal{P}_i c^i$
  Smooth s times on c to get u^j
  \}
Set $u = u^n$
\}

The parallel solve step in Algorithm 3 can be another execution of Algorithm 3, producing a tree construction of subproblems. More typically, it is a direct solver (cf., Algorithm 1) or an iterative solver (cf., Algorithm 2). Algorithm 3 is related to algorithms analyzed in [10], [11], [17], [20], and [24]–[26].

The convergence of Algorithms 1 and 3 is guaranteed by the following theorem.

Theorem 1. *Let $s = 0$ in Algorithm 3, $\Pi_i = \mathcal{P}_i \mathcal{R}_i$, and*

$$\sum_{i=1}^{n} \Pi_i \;=\; I.$$

Then Algorithms 1 and 3 converge to the exact solution if $\mathcal{A}$ commutes with each Π_i, $1 \leq i \leq n$.

An immediate corollary is the following.

Corollary 1. *Theorem 1 remains true for Algorithm 1 if the commutativity condition is replaced by $\Pi_i \mathcal{A}(\Pi_i - I) = 0$ for all i, $1 \leq i \leq n$.*

The proofs to Theorem 1 and Corollary 1 can be found in [14].

Suppose there is a set of internal interfaces

$$\{\Gamma_i\}_{i=1}^{\sigma},$$

where each $,_j$ divides a general domain Ω into two subdomains with equal volumes, and a set of $n = 2^\sigma$ restriction operators

$$\{\mathcal{R}_{\{\tau_i\}}\}, \quad i = 1, \cdots, \sigma, \tag{2.5}$$

where

$$\tau_i \;=\; \begin{cases} 1, & \text{when even functions are annihilated about } ,_i, \\ 0, & \text{when odd functions are annihilated about } ,_i. \end{cases} \tag{2.6}$$

Each $,_i$ divides Ω into two domains $\Omega_{i,1}$ and $\Omega_{i,2}$. Hence, (2.6) implies that for any $x \in \Omega_{i,1}$ and its reflection $y \in \Omega_{i,2}$ (using an unspecified operator which depends on $,_i$, $\mathcal{R}_{\{\tau_i\}}$, and the physical characteristics of Ω), that an even function g about $,_i$ (i.e., $g(x) = g(y)$) is annihilated if $\mathcal{R}_{\{\tau_i\}} g = 0$.

REMARK 1. *If Ω is a circle, then the $,_i$ would be curves which could be quite complicated geometrically (e.g., S shaped curves). The most simple case is when they are straight lines passing through the center of the circle, which is not necessarily at the origin.*

An operator $\mathcal{L}$ *preserves even/odd functions about* $,_k$ if $\mathcal{L}$ applied to any even (odd) function about $,_k$ is a even (odd) function about $,_k$. For example, the Δ operator preserves even and odd functions. Similarly, an operator $\mathcal{L}$ *reverses even/odd functions about* $,_k$ if $\mathcal{L}$ applied to any even (odd) function about $,_k$ is an odd (even) function about $,_k$. Knowing these properties allows the construction of subspaces V_i which are mutually orthogonal.

Each prolongation operator is the adjoint of the correct restriction operator

$$\mathcal{P}_{\{\tau_i\}} \;=\; \mathcal{R}_{\{\mu_i\}}, \tag{2.7}$$

where

$$\mu_i \;=\; \begin{cases} \tau_i, & \text{when } \mathcal{L} \text{ preserves even/odd functions about } ,_i, \\ 1 - \tau_i, & \text{when } \mathcal{L} \text{ reverses even/odd functions about } ,_i. \end{cases}$$

If we renumber the restriction and prolongation operators as

$$\{\mathcal{P}_i, \mathcal{R}_i\}_{i=1}^n,$$

then the following convergence result holds.

THEOREM 2. *Suppose $\mathcal{L}$ either preserves or reverses even and odd functions about each of the internal interfaces $,_i$, $i = 1, \cdots, \sigma$. Suppose the n restriction and prolongation operators are defined as in (2.5) and (2.7) with*

$$\sum_{i=1}^{d} \mathcal{R}_i^* \mathcal{R}_i = I$$

satisfied. Then Algorithm 3 converges to the exact solution in one iteration without smoothing if the subspace problems are solved exactly.

The proof to Theorem 2 can be found in [16].

The convergence of Algorithm 2 is guaranteed by the following theorem.

THEOREM 3. *Let P_{V_i} be the orthogonal projection onto V_i, and λ_{max} and λ_{min} be the maximal and the minimal eigenvalues, respectively, of $\sum_{i=1}^n P_{V_i}$. Then the iterates produced by Algorithm 2 satisfy the error bound*

$$\|u^{m+1} - u\| \;\le\; (\max\{\lambda_{max} - 1, 1 - \lambda_{min}\} + \varepsilon\lambda_{max})\|u^m - u\|. \tag{2.8}$$

An immediate corollary is the following.

COROLLARY 2. *Using the same assumptions as in Theorem 3, if, in addition, $V_i \perp V_j$, for all i, j, then $\lambda_{max} = \lambda_{min} = 1$ and (2.8) reduces to*

$$\|u^{m+1} - u\| \leq \varepsilon \|u^m - u\|.$$

The proofs to Theorem 3 and Corollary 2 can be found in [13].

3. An Example. In this section, a simple example is described where the domain reduction method uses symmetries in (1.1) to split the problem into several subproblems. For $x = (x_1, \ldots, x_d) \in \Omega = (-1,1)^d$, define

$$\mathcal{R}_{i_1,\ldots,i_d} x = ((-1)^{i_1} x_1, \ldots, (-1)^{i_d} x_d), \quad (i_1, \ldots, i_d) \in \{0,1\}^d.$$

The subspaces $V_{i_1,\ldots,i_d} \subset V$ are defined by

$$V_{i_1,\ldots,i_d} = \{ u \in V : \mathcal{R}_{i_1,\ldots,i_d} u = u \}, \quad (i_1, \ldots, i_d) \in \{0,1\}^d.$$

It is immediate that the subspaces are mutually orthogonal in the $a(\cdot,\cdot)$ inner product as well as cover the entire solution space V. Therefore, (2.1) may be split into 2^d subproblems:

$$\text{(3.1)} \quad \text{find } u_{i_1,\ldots,i_d} \in V_{i_1,\ldots,i_d} \text{ such that } a(u_{i_1,\ldots,i_d}, v) = f(v), \ \forall v \in V_{i_1,\ldots,i_d}.$$

Then the solution of (2.1) is simply

$$u = \sum_{i_1,\ldots,i_d=0}^{1} u_{i_1,\ldots,i_d}.$$

Let $\tilde{\Omega} = (0,1)^d$. For a function $\tilde{v}$ on $\tilde{\Omega}$, define prolongation operators $\mathcal{P}_{i_1,\ldots,i_d}$, $(i_1, \ldots, i_d) \in \{0,1\}^d$, by

$$(\mathcal{P}_{i_1,\ldots,i_d} \tilde{v})(x) = (-1)^{i_1 i_1' + \cdots + i_d i_d'} \tilde{v}(\tilde{x}) \quad \text{for } x = \mathcal{R}_{i_1',\ldots,i_d'} \tilde{x}.$$

Then (3.1) can be formulated variationally as a problem on the subdomain $\tilde{\Omega}$ of the form

$$\text{(3.2)} \quad \text{find } \tilde{u}_{i_1,\ldots,i_d} \in \tilde{V}_{i_1,\ldots,i_d} \text{ such that } \tilde{a}(\tilde{u}_{i_1,\ldots,i_d}, \tilde{v}) = f(\tilde{v}), \ \forall \tilde{v} \in \tilde{V}_{i_1,\ldots,i_d}.$$

Specifically, $\tilde{V}_{i_1,\ldots,i_d}$ is related to the physical characteristics of $\tilde{\Omega}$ and the subspaces $V_{i_1,\ldots,i_d}$ by

$$\tilde{V}_{i_1,\ldots,i_d} = V_{i_1,\ldots,i_d} \cap V(\tilde{\Omega}) \quad \text{and} \quad V_{i_1,\ldots,i_d} = \mathcal{P}_{i_1,\ldots,i_d} \tilde{V}_{i_1,\ldots,i_d}.$$

The subspace linear functionals and bilinear forms are related to the ones on the full space V by

$$\tilde{f}(\tilde{v}) = f(\mathcal{P}_{i_1,\ldots,i_d} \tilde{v}), \quad \text{and} \quad \tilde{a}(\tilde{u}, \tilde{v}) = a(\mathcal{P}_{i_1,\ldots,i_d} \tilde{u}, \mathcal{P}_{i_1,\ldots,i_d} \tilde{v}).$$

Finally, the subspace solutions $u_{i_1,\ldots,i_d}$ can be represented in the full space V by

$$u_{i_1,\ldots,i_d} = \mathcal{P}_{i_1,\ldots,i_d} \tilde{u}_{i_1,\ldots,i_d}.$$

In reality, (3.2) is discretized on a uniform mesh, V is the space of finite element functions associated with the mesh (or the vector space associated with a finite difference discretization), and $V_{i_1,\dots,i_d}$ are then subspaces of it. The problems (3.2) can be then written as discretized boundary value problems on the reduced domain $\tilde{\Omega}$.

To visualize the domain reduction process, consider the following example, which illustrates that the method is applicable to a wider range of problems than just elliptic problems with constant coefficients. Let

$$(3.3) \quad \begin{cases} \mathcal{L}u = & -\sum_{i=1}^{d} (a_2(x)u_{x_i})_{x_i} - (a_1(x)u)_{x_i} + a_0(x)u & = f \quad \text{in } \Omega = (-1,1)^d, \\ & u & = 0 \quad \text{on } \partial\Omega, \end{cases}$$

where a_2 and a_0 are even functions about the axes, and a_1 is an odd function about the axes. For example,

$$a_2(x) \ = \ \prod_{j=1}^{d} \sin(\pi x_j), \qquad a_1(x) \ = \ \prod_{j=1}^{d} \cos(\pi x_j), \qquad \text{and} \qquad a_0(x) \ = \ \beta \in \mathbb{R}$$

are suitable choices.

By definition of $u_{i_1,\dots,i_d}$, (3.2) can be written as

$$\mathcal{L}\tilde{u}_{i_1,\dots,i_d} \ = \ \tilde{f}_{i_1,\dots,i_d} \qquad \textit{in } \tilde{\Omega}$$

with boundary conditions

$$(3.4) \qquad \begin{cases} \tilde{u}_{i_1,\dots,i_d} \ = \ 0 & \text{on the } d \text{ sides } x_1 \ = \ 1, \ \dots, \ \text{and } x_d \ = \ 1, \\ \tilde{u}_{i_1,\dots,i_d} \ = \ 0 & \text{on side } x_{i_k} \ = \ 0 \text{ if } i_k \ = \ 1, \ k \ = \ 1,\dots,d, \\ \frac{\partial}{\partial n}\tilde{u}_{i_1,\dots,i_d} \ = \ 0 & \text{on side } x_{i_k} \ = \ 0 \text{ if } i_k \ = \ 0, \ k \ = \ 1,\dots,d. \end{cases} .$$

Note that when (3.3) is discretized on a uniform mesh with a central difference scheme, 2^d systems of equations must be solved. By suitable choices of a_2, a_1, and a_0, the resulting systems can be nonsymmetric, indefinite.

Since the subspaces are orthogonal, Corollary 2 applies. Hence, the correct way to monitor the error over the entire domain is to monitor the error of each of the subproblems: the largest subproblem error is the largest error on the entire domain. Running times on both shared and distributed memory parallel computers (Encore and Intel) are contained in [13].

4. Concrete Formulation. In this section, a more subtle approach to domain reduction is described. A careful examination of (3.4) shows that a purely Dirichlet problem has been converted into a collection of similar subproblems with Dirichlet and Neumann boundaries.

$$(4.1) \qquad \begin{cases} \mathcal{L}u \ = f & \text{in } \Omega \subset \mathbb{R}^d, \quad d \ > \ 0, \\ u \ = \ 0 & \text{on } \partial\Omega_D, \qquad \text{where } \partial\Omega \ = \ \partial\Omega_D \cup \partial\Omega_N, \\ & \qquad\qquad\qquad \text{and } \partial\Omega_D \cap \partial\Omega_N \ = \ \phi, \\ \frac{\partial}{\partial n}u \ = \ 0 & \text{on } \partial\Omega_N. \end{cases}$$

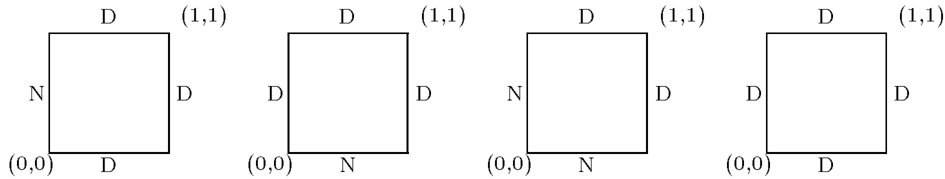
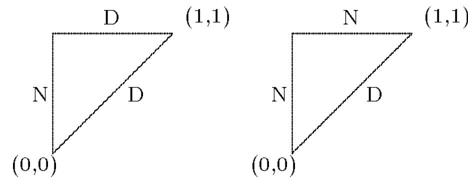


FIG. 1. *Four Way Reduction of the Square*

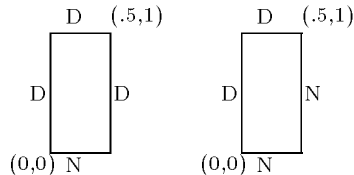
By folding the subproblem domains so that boundary conditions match, a higher degree of parallelism can be attained.

In §4.1, an eight way reduction of the square is constructed from the obvious four way reduction. In §4.2, 60 and 64 way reductions of the cube are sketched, based on the obvious eight way reduction. In §4.3, the effect on the amount of storage and time requirements during the discretization phase is analyzed.

4.1. Square Domain. In the case of a square, an eight way domain reduction is possible. Starting from $\Omega = (-1,1)^2$, four subproblems are solved on subdomains $(0,1)^2$ with Dirichlet (D) and Neumann (N) boundary conditions given in Fig. 1. The subdomain with two Neumann boundary sides can be folded across the diagonal to produce two new subproblems on triangles.



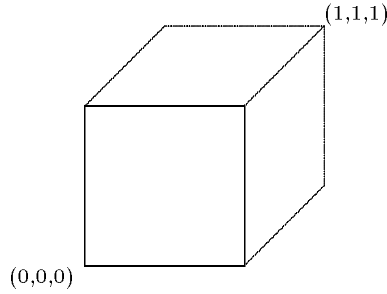
The subdomains with one Neumann boundary side can be folded across the middle of that boundary to produce two new subproblems on rectangles. For example,



Finally, the subdomain which is isomorphic to the original domain can be reduced onto two triangles, two rectangles, or recursively onto an eight way set of subproblems. More details are contained in [5].

4.2. Cube Domain. The three dimensional cube offers a real challenge. Starting from $\Omega = (-1,1)^3$, eight subproblems are solved on subdomains $(0,1)^3$ with

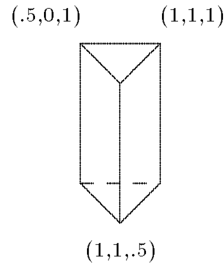
Dirichlet (D) and Neumann (N) boundary conditions given as in (3.4).



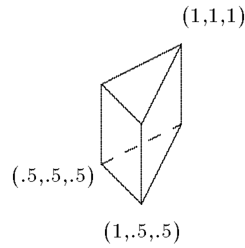
In this case, the subproblems have the following number of Neumann and Dirichlet Neumann boundary faces.

Number of subproblems	Number of faces Neumann	Dirichlet	Number of new subproblems
1	0	6	8
3	1	5	3×8
3	2	4	3×8
1	3	3	4 or 8
Total			60 or 64

The subdomain with no Neumann boundary face can be folded into the cube $(.5, 1)^3$ similarly to (3.4). The subdomains with exactly one Neumann boundary face can be folded into the following prism shaped domain.

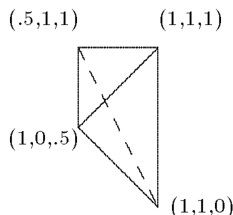


The subdomains with exactly one Neumann boundary face can be folded into the following wedge shaped domain.

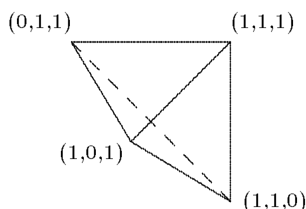


In each of these new subdomains, all of the angles are either 90° or 45°. The subdomain with exactly three Neumann boundary faces can be folded into the

following tetrahedra.



This is not a conveniently shaped domain for finite difference discretizations. A better shaped one is as follows.



Unfortunately, this domain has twice the volume of the previous one.

Normally, three dimensional grid generated problems should never be solved directly since the cost is approximately $cN^{5/3}$, $c \in \mathbb{R}$. Using a 64 way domain reduction, the cost of solving each subproblem directly is at most

$$c(N/64)^{5/3} \;=\; \frac{c}{1024}N^{5/3}, \quad c \in \mathbb{R}.$$

Using exactly 64 processors results in an elapsed time speedup of a factor of 1024.

Many domain decomposition implementations use iterative solvers to reduce the the error by some factor ε. If $k(N,\varepsilon)$ iterations are required to reduce the error in a problem with N unknowns by a factor of at least ε, then the cost is

$$ck(N/64,\varepsilon)(N/64), \quad c \in \mathbb{R}.$$

The function k is not necessarily linear in N, so the speedup may be much greater than just a factor of 64.

Using a multigrid solver reduces the cost to

$$c(N/64), \quad c \in \mathbb{R}.$$

A real question that must be addressed is why would anyone mix a domain decomposition method with multigrid? Under ordinary circumstances, a very simple complexity result proves this to be less efficient (timewise) than just using multigrid directly. This issue is addressed in §6.2.

4.3. Discretization Time and Storage Reduction. In Fig. 1, all four boundary value problems are identical except along the left and bottom boundary lines. Suppose the same grid and discretization method is used on each subdomain. Four sparse systems of linear equations A_i are generated so that

$$A_i c^i \;=\; R_i r^j \tag{4.2}$$

can be solved in Algorithm 3. However, (4.2) can be rewritten as

$$(A_1 + C_i)c^i \;=\; R_i r^j,$$

where C_i is exceedingly sparse (see §6.2 for examples). In fact, C_1 has all zeros in this formulation. Hence, only A_1 and $\{C_i\}_{i=2}^4$ need to be generated, not $\{A_i\}_{i=1}^4$.

In the eight way reduction of the square, three or four sets of similar subproblems exist. By appropriate numbering of the subproblems, only

$$A_1,\ C_{11},\ A_2,\ C_{21},\ A_3,\ C_{31},\ C_{32},\ C_{33}$$

or

$$A_1,\ C_{11},\ A_2,\ C_{21},\ A_3,\ C_{31},\ A_4,\ C_{41}$$

need to be generated.

Similarly for the cube, the eight way reduction leads to generating only A_1 and $\{C_{1i}\}_{i=1}^7$. The 60 way reduction leads to generating $\{A_i, \{C_{ij}\}_{j=1}^7\}_{i=1}^7$, A_8, and $\{C_{8j}\}_{j=1}^3$. The 64 way reduction leads to generating $\{A_i, \{C_{ij}\}_{j=1}^7\}_{i=1}^8$.

Define $NZ(A)$ to be the number of nonzeros in a matrix A. Then the following theorem is immediate.

THEOREM 4. *Suppose $\{A_i, \{C_{ij}\}_{j=1}^{\tau_i}\}_{i=1}^{\sigma}$ must be generated. Further, suppose that there exists a constant $C \in \mathbb{R}$ such that*

$$A_i \in \mathbb{R}^{N\times N}, \quad NZ(A) \;=\; CN, \quad 1 \;\le\; i \;\le\; \sigma.$$

Assume there exists a function ψ and a constant $\bar{C} \in \mathbb{R}$ such that

$$NZ(C_{ij}) \;\le\; \bar{C}\psi(N).$$

If there are φ C_{ij}'s, then the amount of storage required is

$$S(N) \;\le\; \sigma CN \;+\; \varphi\bar{C}\psi(N).$$

Assuming the original problem on the full domain Ω was discretized similarly to the subproblems, approximately nCN storage would be required. The effective memory reduction ratio is

$$E_{MR}(N) \;=\; \frac{nCN}{S(N)}.$$

A goal in using domain reduction methods should be to make $\psi(N)/N \to 0$ as $N \to \infty$. Hence, we have shown when domain reduction is "optimal" in discretization space utilization.

COROLLARY 3. *Under the same assumptions as Theorem 4,*

$$\lim_{N\to\infty} E_{MR}(N) \;=\; n/\sigma \quad \textit{if and only if} \quad \lim_{N\to\infty} \psi(N)/N \;=\; 0.$$

Besides saving storage, this technique saves time. Let $T_G(N)$ represent the time required to generate any of the A_i. Let $\mathbb{P}_\sigma(x)$ be the set of all polynomials in x of order σ. Typically,

$$T_G(N) \;=\; \mathbb{P}_\sigma(N)\mathbb{P}_\tau(\log N), \quad \text{some } \sigma \text{ and } \tau.$$

For example, when $T_G(N) = CN$, linear speedup up to a factor of $E_{MR}(N)$ is possible. Similarly, when $T_G(N) = CN\log N$, polylog speedup is possible.

In many real problems, the cost of discretizing the boundary value problem takes far more computer time than the algorithm used to solve the problem. In these cases, reducing the discretization time is far more important than reducing the solution time. Of course, reducing both sets of times should be attempted.

5. Linda Approach to Computation. A number of semi-portable parallel programming environments now exist. Some of these systems mimic communication libraries supplied by some hardware manufacturer, others are complete languages. The Linda system is one of the few which is language independent in concept (see [3], [6], [8], [7], and [21]).

For each Linda implementation, there is a preprocessor and a communication library. The preprocessor adds a new data type (a *tuple*) and six operators which manipulate tuples or *processes*. As a result, Linda can be interpreted as a global database system, rather than as a language. The data is maintained in *tuple space*, which can be distributed across a number of processors. The processors may be part of a tightly coupled system (e.g., a shared or distributed memory machine), or a loosely coupled system (e.g., where the processors are connected via a network). The extreme case is when the data is on machines in different locations, say, NASA Houston and a space shuttle.

Data flow is analyzed both at compile time and during execution. On distributed memory systems, efforts are made to prefetch tuples based on this analysis. There can be a penalty for using this type of system. When transferring small tuples (e.g., a few words of memory) between processors, the extra overhead of Linda can be as high as 40%. However, when transferring large tuples (e.g., several thousand words of memory), the penalty is less than 1%.

The actual Linda operators are as follows.

eval	Evaluate a function in parallel.
in	Input destructively a tuple from tuple space (wait for one if necessary).
inp	Input destructively a tuple (if it exists) from tuple space.
out	Output a tuple to tuple space.
rd	Input nondestructively a tuple from tuple space (wait for one if necessary).
rdp	Input nondestructively a tuple (if it exists) from tuple space.

The destructive operators remove a tuple from tuple space. The nondestructive operators just copy a tuple out of tuple space. The operators ending in p return a success or failure value, depending on if a tuple matches the pattern or not. In either case, the routines return immediately.

A tuple can have any number of entries, and each entry can be almost anything. Tuples with mixed data types (i.e., character strings, arrays, reals, integers, and constants) are allowed. Reading or inputting is complicated by a matching feature: some arguments can be formal (i.e., completely specified) and some can be informal (anything in tuple space can match these). Informal arguments are preceded by a question mark. For example,

$$\text{in}(4,\ \text{``}arff\text{''},\ ?\ x : n)$$

would input *any* tuple which has as its first entry an integer 4, the character string (of length 4) *arff* as its second entry, and an array of the same specific data type as x. This array would be stored in x and its length in n.

6. Tupleware. A software package has been produced using the Linda system for solving large sparse linear systems of equations. While the work presented here utilizes features of Linda, it should be emphasized that the results carry over to other programming environments. Implementation of the domain reduction method using

this package reduces the memory requirements by an integer factor over conventional packages.

In §§6.1–6.2, a number of constants are used of the form

TW_$\cdots$ or TWUP_$\cdots$

They are character strings or integer valued objects, and are defined inside the package. Their exact definitions, however, are not germane to this paper.

6.1. Package Design. A layered design is used in the package: there are serial, ugly parallel, and clean parallel routines. An ugly parallel routine runs on an individual processor and communicates with other processors running the same routine. A clean parallel routine is called serially by the user and interfaces with a set of ugly parallel routines. The ugly parallel routines are faster than the clean routines, but for large enough problems, there is no noticeable difference.

A legitimate question to ask is why such an approach was considered. The following (C related) pseudo code uses p processors to do a conjugate gradient iteration using an ugly parallel routine.

for ($i = 0$; $i < p$; $i++$)
    eval(twup_conjugate_gradients($keycg, \cdots, i, p$));

There can be a quite large penalty in this approach. First, p idle processors must be found, then a (memory) working set must be set up for each processor, and finally the routine is started on each processor. On an Encore or Sequent, setting up a working set uses a UNIX *fork* operation, which copies *all* of the current working set (potentially many megabytes of memory). In fact, this can take longer than executing the routine, if caution is not exercised.

A more efficient technique is to start the p processors early, give them something to do after a while, and then have them wait for more work. The clean interface provides this capability. To initiate p processors, a key (e.g., *keyc*) is designated, and the clean interface is called.

twcl_interface(*keyc*, TW_STARTUP, p);

The twcl_interface routine starts p processors that wait for appropriate tuples to appear in tuple space describing work to do.

The conjugate gradient example will now be reconsidered. A tuple is first put in tuple space with the parameters required by the twup_conjugate_gradients routine.

out($keyc, keycg, \cdots, p$);

Then the clean interface routine is called again.

twcl_interface(*keyc*, TWUP_CONJUGATE_GRADIENTS, p);

Information about the iteration (e.g., number of iterations or the factor that the residual norm was reduced by) can be removed from tuple space upon return.

in(TW_CL_INTERFACE_RET, *keyc*,
    TWUP_CONJUGATE_GRADIENTS, ? *ret*:);

Other operations can now be performed using the p processors by appropriate calls to twcl_interface without paying the start up penalty. To quit using the p processors, the clean interface is called one last time.

twcl_interface(*keyc*, TW_QUIT, p);

A more complicated example, and more relevant to this paper, is to assign q processors to each of the n subproblems. A start-up call to twcl_interface is made for each of the n subproblems (with a different key, $keyc_i$, for each), and all qn processors compute in parallel. Domain decomposition routines are easily produced using this technique.

For a constant s, vectors x, y, and z, and a sparse matrix A, routines exist to compute $x^T y$, $z = y + sx$, and sparse matrix-vector multiplication (actually, $y = Ax$, $y = A^T x$, $y = y \pm Ax$, or $y = y \pm A^T x$). Routines also exist to analyze a matrix, build diag(A), redistribute a vector across processors, and print distributed objects. These routines are used to implement conjugate gradients, conjugate gradients squared (see [28]), multigrid, and some simple domain reduction methods.

6.2. Memory Usage. The naive approach to storing a matrix in a distributed manner is to divide it into a collection of blocks. This is also done here, but the blocks are allowed to be subdivided into overlapping subblocks.

A matrix A is stored hierarchically in tuple space. All tuples are keyed using a character string. To access a subblock b on processor p, a description is first read.

rd($keya$, p, b, ? D:)

D is a structure with the following information.

$Col_indices$	Pointer to an array of column indices.
Col_coeffs	Pointer to an array of coefficients.
$block_col$	Where the first column in this subblock resides in this block.
$block_row$	Where the first row in this subblock resides in this block.
$cols$	The number of rows in this subblock.
len_anal_info	Either 0 or the number of processors.
len_col_coeffs	Length of the coefficients array.
$len_col_indices$	Length of the column index array.
$row_entries$	The number of rows in this block with nonzeros.
$rows$	The number of rows in this block.
$storage_type$	Symmetric or nonsymmetric storage scheme.
x_piece	Which piece of a distributed vector x to use in computing Ax.
$matrix_name$	A character string (or *key*) identifying the coefficients.

Getting the coefficients and column indices requires a second access to tuple space.

rd($matrix_name$, ? $Col_indices$:, ? Col_coeffs:)

Certain routines need some global information about A. This requires a third access to tuple space.

rd(TW_ANALYZE_NAME, $keya$, ? $InfoBlk$:, ? $InfoRow$:,
? $InfoCol$:, ? $InfoTrn$:)

The length of each of the $Info$ arrays is equal to the number of processors. The information tuple contains the following information.

$InfoBlk$	The number of subblocks associated with a processor.
$InfoCol$	The first column of A associated with a processor.
$InfoRow$	The first row of A associated with a processor.
$InfoTrn$	The number of vectors transferred by the matrix–vector multiplier to this processor when computing $A^T x$.

All of these vectors are computed in a simple routine in the package.

Certain matrix names are reserved, namely, ones for the identity matrix I and

Row	Column index	Coefficient
$r1$	$-r1$	$a_{r1,r1}$
	$c1$	$a_{r1,c1}$
	$c2$	$a_{r1,c2}$
	$\vdots$	$\vdots$
	ci	$a_{r1,ci}$
$r2$	$-r2$	$a_{r2,r2}$
	$c1$	$a_{r2,c1}$
	$c2$	$a_{r2,c2}$
	$\vdots$	$\vdots$
	cj	$a_{r2,cj}$
$r3$	$-r3$	$a_{r3,r3}$
	$\vdots$	$\vdots$
$r\sigma$	$-r\sigma$	$a_{r\sigma,r\sigma}$
	$c1$	$a_{r\sigma,c1}$
	$c2$	$a_{r\sigma,c2}$
	$\vdots$	$\vdots$
	ck	$a_{r\sigma,ck}$
End	0	

FIG. 2. *Format for column index and coefficient arrays*

$-I$. In these special cases, the coefficients are assumed rather than stored in tuple space. Extra code handles these cases.

The coefficients and column indices are stored in a tuple as arrays, as is shown in Fig. 2. Rows and columns are one based (rather than zero based which would be natural in some computer languages). For example, the matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 2 \\ 3 & 4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 6 \end{bmatrix}$$

would be stored as

Column index	Coefficient
-1	1
5	2
-2	4
1	3
-4	0
1	5
5	6

Unlike many sparse matrix packages available today, no array of pointers into the column index array is required. Further, for an $N \times N$ matrix with M rows of all zeros, information is stored only for the $N - M$ rows which have nonzeros, not

for all N rows. This format does not guarantee, however, that no zeros are stored. For example, any diagonal element $a_{ii} = 0$ is stored. This format really assumes that the matrices will be used row by row sequentially rather than in a random fashion. While this does not completely eliminate sparse Gaussian elimination as an interesting algorithm for this package, it certainly hinders it greatly.

Many packages which are designed for solving variable coefficient boundary value problems use a significant amount of storage when solving even simple problems (e.g., a Poisson or Helmholtz equation). The overlapping subblock technique overcomes this flaw.

First, consider a constant coefficient Helmholtz equation.

$$\begin{cases} -\triangle u + Su = f & \text{in } (-1,1)^2, \\ u = 0 & \text{on } \partial(-1,1)^2, \end{cases}$$

where $S \in \mathbb{R}$. Assume these problems are discretized by either central finite differences on a uniform mesh or by finite elements on a uniform triangulation using C^0 piecewise linear polynomials and the usual nodal basis functions. In either case, the matrix A is block tridiagonal [29].

$$A = [\,-I,\ T,\ -I\,] \in \mathbb{R}^{N\times N}, \quad N = M^2,$$

where I is an $M \times M$ identity matrix and T is an $M \times M$ tridiagonal matrix.

$$T = [\,-1, s, -1\,], \quad s = 4 + Sh^2,$$

where h is the mesh spacing.

As an extreme, A can be stored on a single processor as follows.

M	$M \times M$ blocks of diag(T).
2	$(N-M)\times(N-M)$ negative identity matrices, one starting at $(1, M+1)$, the other at $(M+1, 1)$.
2	$(N-1)\times(N-1)$ negative identity matrices, one starting at $(1,2)$, the other at $(2,1)$.

Define

S_I	The number of bytes per integer (typically, 4 or 8, but not 2).
S_R	The number of bytes per real (typically, 8).
S_C	The number of bytes to store a key (typically, 40).
S_A	The number of bytes per description (typically $12S_I + S_C$).

Then the amount of storage required is

$$(M+4)S_A + (M+1)S_I + MS_R.$$

While this storage technique saves a very large amount of storage, it saves even more when $S_A < (M+1)S_I + MS_R$. On multiple processors, the $-I$ subblocks would be split in the obvious manner.

Now consider a variable coefficient Helmholtz equation. It would be stored as follows.

1	$N \times N$ block of diag(T)
2	$(N-M)\times(N-M)$ negative identity matrices, one starting at $(1, M+1)$, the other at $(M+1, 1)$.
2	$(N-1)\times(N-1)$ negative identity matrices, one starting at $(1,2)$, the other at $(2,1)$.

Then the amount of storage required is

$$5S_A \ + \ (N+1)S_I \ + \ NS_R.$$

On multiple processors, the subblock associated with diag(T) would be split as well as the $-I$ subblocks.

Consider the two and three dimensional boundary value problems on the square and cube discussed in §4. The operators are no longer assumed to be simple boundary value problems. Assume these problems are discretized by either central differences on a uniform mesh or by finite elements on a uniform triangulation.

A four way reduction of the square leads to four discrete problems on the same subdomain. The only difference between the subproblems is the boundary conditions. Hence, only one of these needs to be discretized, and the rest are recovered by adding a correction (see §4.3). This is also true of the eight way reduction of the cube. Hence, for an n (4 or 8) way reduction on the square or cube, we have

$$A_i \ = \ A_1 + C_i, \quad i = 1, \cdots, n,$$

where C_i is extremely sparse.

By choosing a zero initial guess, $s = 0$, and $m = 1$ in Algorithm 3, we avoid having to generate the discrete system A on Ω. Suppose the uniform grid on the domain Ω has N points, of which M are along the boundary. Then the amount of storage needed to store each of the coefficient and column index arrays is approximately

$$\frac{C}{n}N + \frac{3}{4}M \qquad \text{instead of} \qquad CN.$$

To put this in perspective, the following table gives the total amount of integer and real storage required to solve an n way reduction.

n	N	M	C	$(C/n)N + .75M$	CN	$CN/[(C/n)N + .75M]$
4	126^2	500	3	12,282	47,628	3.8779
4	126^2	500	5	20,220	79,380	3.9258
4	126^2	500	9	36,096	142,884	3.9584
8	62^3	22,328	4	135,910	953,312	7.0143
8	62^3	22,328	7	225,283	1,668,296	7.4053
8	62^3	22,328	27	821,103	6,434,856	7.8368

To convert these numbers into bytes, take a number k in the table, and compute $(S_I + S_R)k$.

The amount of storage needed to store the matrices used in a standard domain decomposition method with minimal overlap (or a standard iterative method) is given by the CN column above. The amount of storage required by a standard multigrid is $((n+1)/n)CN$ and for one of the nontelescoping multigrid methods (see [17] and [20]) is $CN \log_n N$.

n	N	C	Multigrid	Nontelescoping multigrids
4	126^2	3	63,504	638,644
4	126^2	5	105,840	1,064,408
4	126^2	9	190,512	1,915,934
8	62^3	4	1,089,499	2,748,129
8	62^3	7	1,906,624	4,809,227
8	62^3	27	7,354,121	18,549,875

The large difference in memory requirements can determine whether or not a given problem can be solved on some parallel machines. In fact, even on a serial machine the space savings are so large that this makes using the domain reduction method an interesting prospect (c.f., the end of §4.2).

The eight way reduction of the square increases memory requirements somewhat. If there are four sets of similar subproblems, then

$$\frac{C}{2}N \; + \; M$$

storage is required. The following table gives the total amount of integer and real storage required to solve an 8 way reduction using four similar subproblems.

n	N	M	C	$(C/2)N+M$	$CN/[(C/2)N+M]$
8	126^2	500	3	24,314	1.9589
8	126^2	500	5	40,190	1.9751
8	126^2	500	9	71,942	1.9861

If there are three sets of similar subproblems, then the storage requirement depends on whether there are three correction matrices for the triangular or rectangular subdomain problems. Since less storage is required for correcting the subproblems on rectangles, it makes more sense to correct those problems. Thus,

$$\frac{3C}{8}N \; + \; M$$

storage is required. The following table gives the total amount of integer and real storage required to solve an 8 way reduction using three similar subproblems.

n	N	M	C	$(3C/8)N+M$	$CN/[(3C/8)N+M]$
8	126^2	500	3	18,360	2.5940
8	126^2	500	5	30,267	2.6226
8	126^2	500	9	54,081	2.6420

7. Conclusions. The domain reduction method offers an interesting mathematical preconditioning technique for solving real problems. When it is directly applicable to a problem, it offers a way of producing a direct method which is embarrassingly parallel. When the cost of solving the subproblems by a direct method is prohibitive, iterative methods should be used, similar to domain decomposition methods.

Unlike standard domain decomposition methods, there is no question about how much overlapping of subdomains, how to communicate with neighboring subdomains, or how many iterations of communication are required. On typical problems, only one iteration of the domain reduction method (using iterative solvers) is required, with the worst error on any subdomain being the worst error on the entire domain. Typically, this means that for any problem that a standard domain decomposition method and domain reduction method can both solve, the domain reduction method always takes less time and storage to solve the problem to the same accuracy.

Storage can be reduced by a significant constant factor, and discretization time can be reduced by a constant factor or more, depending on the time complexity formula for a given problem. The amount of storage required is substantially less than for traditional domain decomposition implementations, or ones based on standard iterative methods including multigrid.

Finally, a highly portable implementation using the Linda system for parallel or distributed programming exists. Using Linda greatly reduced the coding time and directly led to the rapid development of the tricks used in the discretization discussions of this paper.

Acknowledgements. I would like to thank Robert Bjornson, Nicholas Carriero, David Gelernter, and Roger Smith (all of the Computer Science Department at Yale University). The first three people provided help in using the Linda system. Roger asked me why I was doing tupleware, and after an explanation to him, I developed the reduced storage technique based on the overlapped subblock codes. Finally, I would like to thank Steve McCormick at the University of Colorado at Denver for offering some good suggestions while he acted as a relentless referee.

REFERENCES

[1] E. L. Allgower, K. Böhmer, and M. Zhen, *A generalized equilbranching lemma with applications in* $\mathbf{D}_4 \times \mathbf{Z}_2$ *symmetric elliptic problems: part 1*, Tech. Rep. 9, Philipps-Universität Marburg, Marburg, Germany, 1990.

[2] R. E. Bank and C. C. Douglas, *Sharp estimates for multigrid rates of convergence with general smoothing and acceleration*, SIAM J. Numer. Anal., 22 (1985), pp. 617–633.

[3] R. Bjornson, N. Carriero, and D. Gelernter, *The implementation and performance of hypercube Linda*, Tech. Rep. 690, Department of Computer Science, Yale University, New Haven, 1989.

[4] A. Brandt, *Multi–level adaptive solution to boundary–value problems*, Math. Comp., 31 (1977), pp. 333–390.

[5] F. Brezzi, C. C. Douglas, and L. D. Marini, *A parallel domain reduction method*, Numer. Meth. for PDE, 5 (1989), pp. 195–202.

[6] N. Carriero, *Implementation of tuple space machines*, PhD thesis, Yale University, December 1987. Also, Computer Science Department, Yale University, Technical Report 567.

[7] N. Carriero and D. Gelernter, *How to write parallel programs: a guide to the perplexed*, Tech. Rep. 628, Department of Computer Science, Yale University, New Haven, 1989. To appear in ACM Comp. Surveys.

[8] ———, *Linda in context*, Comm. ACM, 32 (1989), pp. 444–458.

[9] T. Chan, R. Glowinski, G. A. Meurant, J. Périaux, and O. Widlund, eds., *Domain Decomposition Methods for Partial Differential Equations II*, Philadelphia, 1989, Society for Industrial and Applied Mathematics.

[10] H. C. Chen, *The* SAS *domain decomposition method for structural analysis*, PhD thesis, University of Illinois at Urbana–Champaign, Urbana, Illinois, 1988.

[11] H. C. Chen and A. H. Sameh, *A matrix decomposition method for orthotropic elasticity problems*, SIAM J. Matrix Anal. Appl., 10 (1989), pp. 39–64.

[12] C. C. Douglas, *Multi–grid algorithms with applications to elliptic boundary–value problems*, SIAM J. Numer. Anal., 21 (1984), pp. 236–254.

[13] C. C. Douglas and J. Mandel, *The domain reduction method: high way reduction in three dimensions and convergence with inexact solvers*, in Fourth Copper Mountain Conference on Multigrid Methods, J. Mandel, S. F. McCormick, J. E. Dendy, C. Farhat, G. Lonsdale, S. V. Parter, J. W. Ruge, and K. Stüben, eds., Society for Industrial and Applied Mathematics, Philadelphia, 1989, pp. 149–160.

[14] C. C. Douglas and W. L. Miranker, *Constructive interference in parallel algorithms*, SIAM J. Numer. Anal., 25 (1988), pp. 376–398.

[15] ———, *Some nontelescoping parallel algorithms based on serial multigrid/aggregation/disaggregation techniques*, in Multigrid Methods: Theory, Applications, and Supercomputing, S. F. McCormick, ed., Marcel Dekker, New York, 1988, pp. 167–176.

[16] C. C. Douglas and B. F. Smith, *Using symmetries and antisymmetries to analyze a parallel multigrid algorithm*, SIAM J. Numer. Anal., 26 (1989), pp. 1439–1461.

[17] 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.

[18] R. Glowinski, G. H. Golub, G. A. Meurant, and J. Périaux, eds., *On the Schwarz alternating method I*, Philadelphia, 1988, Society for Industrial and Applied Mathematics.

[19] W. Hackbusch, *Multigrid Methods and Applications*, Springer–Verlag, Berlin, 1985.

[20] ———, *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.

[21] J. S. Leichter, *Shared tuple memories, shared memories, buses, and LAN's – Linda implementations across the spectrum of connectivity*, PhD thesis, Yale University, New Haven, CT, 1989. Computer Science Department Technical Report 714.

[22] P. L. Lions, *On the Schwarz alternating method I*, in Domain Decomposition Methods for Partial Differential Equations, R. Glowinski, G. H. Golub, G. A. Meurant, and J. Périaux, eds., Society for Industrial and Applied Mathematics, Philadelphia, 1988, pp. 1–42.

[23] ———, *On the Schwarz alternating method II*, in Domain Decomposition Methods for Partial Differential Equations II, T. Chan, R. Glowinski, G. A. Meurant, J. Périaux, and O. Widlund, eds., Society for Industrial and Applied Mathematics, Philadelphia, 1989, pp. 47–70.

[24] J. Mandel and S. F. McCormick, *Iterative solution of elliptic equations with refinement: the model multi–level case*, in Domain Decomposition Methods for Partial Differential Equations II, T. Chan, R. Glowinski, G. A. Meurant, J. Périaux, and O. Widlund, eds., Society for Industrial and Applied Mathematics, Philadelphia, 1989, pp. 93–102.

[25] S. F. McCormick, *Fast adaptive composite grid (FAC) methods*, in Defect Correction Methods, K. Böhmer and H. J. Stetter, eds., Springer–Verlag, Vienna, 1984, pp. 115–121.

[26] S. F. McCormick and J. Thomas, *The fast adaptive composite grid (FAC) method for elliptic equations*, Math. Comp., 46 (1986), pp. 439–456.

[27] H. A. Schwarz, *Über einige Abbildungsaufgaben*, Ges. Math. Abh., 11 (1869), pp. 65–83.

[28] P. Sonneveld, *CGS, a fast Lanczos–type solver for nonsymmetric linear systems*, SIAM J. Sci. Stat. Comp., 10 (1989), pp. 36–52.

[29] G. Strang and G. J. Fix, *An Analysis of the Finite Element Method*, Prentice–Hall, New York, 1973.