A New Parallel Domain Decomposition Method for the Adaptive Finite Element Solution of Elliptic Partial Differential Equations

Randolph E. Bank*and Peter K. Jimack†

Abstract

We present a new domain decomposition algorithm for the parallel finite element solution of elliptic partial differential equations. As with most parallel domain decomposition methods each processor is assigned one or more subdomains and an iteration is devised which allows the processors to solve their own subproblem(s) concurrently. The novel feature of this algorithm however is that each of these subproblems is defined over the entire domain — although the vast majority of the degrees of freedom for each subproblem are associated with a single subdomain (owned by the corresponding processor). This ensures that a global mechanism is contained within each of the subproblems tackled and so no separate coarse grid solve is required in order to achieve rapid convergence of the overall iteration. Furthermore, by following the paradigm introduced in [5], it is demonstrated that this domain decomposition solver may be coupled easily with a conventional mesh refinement code, thus allowing the accuracy, reliability and efficiency of mesh adaptivity to be utilized in a well load-balanced manner. Finally, numerical evidence is presented which suggests that this technique has significant potential, both in terms of the rapid convergence properties and the efficiency of the parallel implementation.

Key words. Partial differential equations, Parallel computing, Domain decomposition, Mesh adaptivity, Finite element method.

1 Introduction

Parallel algorithms for the efficient solution of elliptic partial differential equations (PDEs) have developed significantly over the past fifteen years or so. The majority of these algorithms fall into the general category of domain decomposition (DD) methods, about which there exists an extensive body of literature (see for example [27] or previous proceedings in this series). In such methods the domain must be divided into a number of subdomains (either overlapping or disjoint) and it is necessary to solve a sequence of smaller problems on these subdomains in order to determine the overall solution. The attraction of this approach for users of parallel computing systems comes when the sequence is such that some of these smaller problems may be solved concurrently.

One of the simplest parallel DD algorithms is the additive version of the Schwartz alternating method. Assuming that there is a one-to-one correspondence between processors and subdomains this technique only requires data communication between processors owning neighbouring subdomains. Its weakness however is that, as the underlying finite element (or finite difference) mesh is refined, its rate of convergence deteriorates significantly. (This may be viewed in terms of a corresponding preconditioned system whose condition number increases rapidly with the number of degrees of freedom in the mesh.)

*Dept. of Mathematics, University of California at San Diego, La Jolla, CA 92093, USA.

†School of Computer Studies, University of Leeds, Leeds LS2 9JT, UK.

One approach to overcoming this weakness is to add some coarse mesh correction procedure directly to the additive Schwartz method (e.g. [17, 35]). Such a mechanism for the global transport of information appears to be essential for the quality of any algorithm not to deteriorate as the underlying mesh is refined. In particular, it is also present in iterative substructuring methods such as [11, 12, 13, 14, 18, 20, 24, 33]. For all of these algorithms the condition number of the underlying preconditioned system increases only very slowly, if at all, as the mesh is refined. The price that is paid for this improvement however is that the additional communication overhead and load-balancing costs associated with the global mechanism are quite significant. This tends to make efficient parallel implementations for irregular meshes (due to the use of adaptivity or the geometric complexity of the domain for example) quite challenging.

A further development of the concept of a global or coarse mesh correction comes from the use of multilevel methods (e.g. [15, 37]). In these algorithms the coarse mesh is itself only a little coarser than the original mesh and is also partitioned by subdomain, so the coarse level correction problem may also be solved by a domain decomposition method. When the same two-level method is applied recursively to n levels using a nested sequence of meshes a (parallelizable) multilevel method is obtained. This is one approach to obtaining a global correction in a naturally parallel manner however it does tend to add further to the overall (global) communication overheads which now build up at each level. Practical parallel implementation on locally refined unstructured meshes is also far from straightforward.

In [36] Xu discusses how all of these DD techniques (and numerous others which are cited) relate to more general subspace correction ideas (with parallel multigrid algorithms also being considered in the same context). It may be observed that in all of the above approaches parallelism is achieved through each processor working on *its own* subdomain with an additional global correction introduced in some manner. The parallel algorithm that we introduce in this paper (motivated by the work of [5, 30, 31]) is rather different however since each processor works over the *entire* domain. The function spaces that each processor computes with are nevertheless very different from each other: each having the vast majority of their degrees of freedom in the particular subdomain owned by that processor. One way of viewing the proposed algorithm is therefore as a variation on the subspace correction approach. In Section 2, where this algorithm is introduced in detail, we take a more geometric view however and describe the method in terms of different finite element meshes and their corresponding stiffness matrices. An empirical study of the convergence properties of the algorithm is presented in Section 3. This is put forward as justification for the approach, along with some further variants which are also described. Finally, in Section 4, the parallel efficiency of our implementation is considered and a number of issues which warrant further investigation are discussed.

2 The parallel algorithm

The parallel technique that we introduce in this section is designed to utilize standard sequential adaptive mesh algorithms based upon local h-refinement, such as [4, 10, 16, 28, 34] (and many more), with only minimal modifications. The initial implementation described here is for linear problems in two space dimensions however we foresee no significant obstacles to extending the technique to nonlinear problems (as in [4, 23] for example) or to three space dimensions (as in [10, 34]). In order to simplify the description below it is convenient to make a small number of assumptions at this point. These are considered further at the end of the section.

Assumption 2.1 *Some local error estimator is available which, given a triangular element, is able* *to return an estimate of the error on that element.*

Assumption 2.2 *The domain has been triangulated with a coarse mesh and this mesh has been partitioned into p sub-meshes (where p represents the number of processors in use) which each have an approximately equal total error (as defined by the above error estimator when applied to each element of the coarse mesh). Furthermore, each processor holds a complete copy of this coarse mesh and has a record of which subdomain (i.e. sub-mesh) each element of the coarse mesh belongs to.*

Assumption 2.3 *A sequential hierarchical refinement code is available for locally refining the coarse mesh based upon local error estimates. At this stage only piecewise linear approximations will be considered on the refined meshes and the location of each node will remain fixed once it has been created (i.e. neither p-refinement (e.g. [1, 2]) nor r-refinement (e.g. [4, 16]) will be considered).*

2.1 Introduction to the algorithm

In order to introduce the proposed algorithm it is simplest to consider the special case where $p = 2$. Based on our assumptions above each coarse element may be refined locally on each processor until the error in each leaf element satisfies some prescribed tolerance (or some maximum refinement depth is reached). If, on processor i, this tolerance is chosen to be extremely large for each element *not* belonging to subdomain i, then the final mesh on that processor will only be refined inside subdomain i or immediately outside it. (Typical h-refinement codes only permit a difference of at most one level of refinement between neighbouring elements in which case there may be a "safety layer" of refinement on the outer border of subdomain i on processor i.) In fact, it is actually convenient to extend this region of refinement on processor i by an extra "layer" of elements by only setting an artificially high tolerance on elements not in subdomain i and which do not border subdomain i. (Note that an element's parent may border subdomain i without the element itself being on the border. In this situation the element will be given a large error tolerance, even though its parent was not, so as to prevent further local refinement.) Figure 1 illustrates this for the case $p = 2$ on the domain $(0, 1) \times (0, 1)$ with one subdomain consisting of the region above the line $y = x$ and the other consisting of the region below this line.

We now define a *global fine mesh* as being the union from $i = 1$ to p of the fine mesh on subdomain i created by processor i. To avoid complications at this point we make the following assumption (which will also be considered further in Subsection 2.4 below).

Assumption 2.4 *The union from $i = 1$ to p of the fine mesh on subdomain i created by processor i is a conforming finite element mesh over the entire domain.*

When solving a linear elliptic PDE such as

$$-\underline{\nabla} \cdot (A\underline{\nabla} u) + \underline{b} \cdot \underline{\nabla} u + cu = f \qquad \text{on } \Omega \subset \Re^2 \tag{2.1}$$

(where A is symmetric and strictly positive-definite, $\underline{b}$ may be $\underline{0}$ and $c \geq 0$) subject to well-posed boundary conditions (e.g. $u|_{\partial\Omega} = 0$ for simplicity) using the Galerkin finite element method on the global fine mesh one obtains a system of n linear algebraic equations of the form

$$\int_\Omega \left(A\underline{\nabla}(\sum_{j=1}^{n} u_j\alpha_j) \right) \cdot \underline{\nabla}\alpha_i \; d\underline{x} + \int_\Omega \underline{b} \cdot \underline{\nabla}(\sum_{j=1}^{n} u_j\alpha_j)\alpha_i \; d\underline{x} + \int_\Omega c(\sum_{j=1}^{n} u_j\alpha_j)\alpha_i \; d\underline{x} = \int_\Omega f\alpha_i \; d\underline{x} \tag{2.2}$$

for $i = 1, ..., n$. Here n is the total number of vertices in the global fine mesh (excluding the Dirichlet boundary), α_j is the usual piecewise linear finite element basis function which has value 1 at vertex

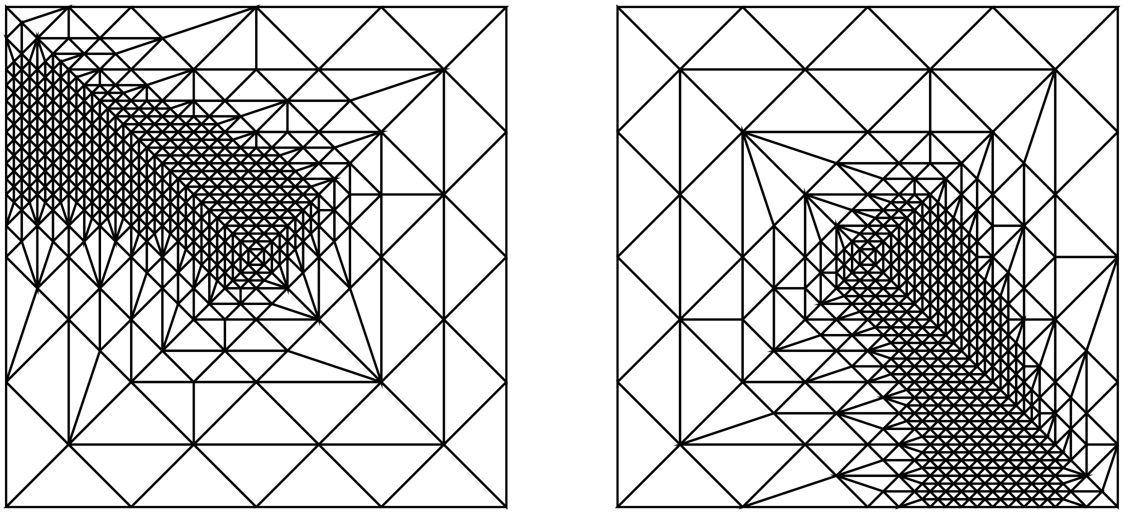

Figure 1: A typical pair of locally refined meshes for two subdomains. In this small example the coarse mesh contains 64 elements and at most three levels of refinement are permitted.

j of the mesh, and u_j is the solution value at vertex j which is to be determined. It is conventional to express these equations in matrix form as the $n \times n$ linear system

$$K\underline{u} = \underline{f} \; , \tag{2.3}$$

where the stiffness matrix K and the load vector $\underline{f}$ have entries given by

$$\begin{aligned} K_{ij} &= \int_\Omega \left(\underline{\nabla}\alpha_i \cdot (A\underline{\nabla}\alpha_j) + \alpha_i(\underline{b} \cdot \underline{\nabla}\alpha_j) + c\alpha_i\alpha_j \right) \, d\underline{x} \; , & (2.4) \\ f_i &= \int_\Omega f\alpha_i \, d\underline{x} \; . & (2.5) \end{aligned}$$

It is this sparse system that we wish to solve in parallel.

Suppose the vector, $\underline{u}$, of unknown nodal values on the global fine mesh is written as $\underline{u}^T = (\underline{u}_1^T, \underline{u}_2^T, \underline{u}_s^T)$ where $\underline{u}_1$ is the vector of unknowns strictly inside subdomain 1, $\underline{u}_2$ is the vector of unknowns strictly inside subdomain 2 and $\underline{u}_s$ is the vector of unknowns shared by the two subdomains. The system (2.3) may be expressed in block matrix form as

$$\begin{bmatrix} A_1 & 0 & B_1 \\ 0 & A_2 & B_2 \\ C_1 & C_2 & A_s \end{bmatrix} \begin{bmatrix} \underline{u}_1 \\ \underline{u}_2 \\ \underline{u}_s \end{bmatrix} = \begin{bmatrix} \underline{f}_1 \\ \underline{f}_2 \\ \underline{f}_s \end{bmatrix} \tag{2.6}$$

with the obvious block structure. Note however that it is also possible to apply the Galerkin finite element method on the meshes generated by the two processors. On processor 1 this yields the system

$$\begin{bmatrix} A_1 & 0 & B_1 \\ 0 & \tilde{A}_2 & \tilde{B}_2 \\ C_1 & \tilde{C}_2 & A_s \end{bmatrix} \begin{bmatrix} \underline{u}_{1,1} \\ \underline{u}_{2,1} \\ \underline{u}_{s,1} \end{bmatrix} = \begin{bmatrix} \underline{f}_1 \\ \tilde{\underline{f}}_2 \\ \underline{f}_s \end{bmatrix} \tag{2.7}$$

whilst on processor 2 this yields the system

$$\begin{bmatrix} \tilde{A}_1 & 0 & \tilde{B}_1 \\ 0 & A_2 & B_2 \\ \tilde{C}_1 & C_2 & A_s \end{bmatrix} \begin{bmatrix} \underline{u}_{1,2} \\ \underline{u}_{2,2} \\ \underline{u}_{s,2} \end{bmatrix} = \begin{bmatrix} \tilde{\underline{f}}_1 \\ \underline{f}_2 \\ \underline{f}_s \end{bmatrix} . \tag{2.8}$$

Note that the matrix blocks with a tilde over them are smaller than the corresponding blocks in (2.6) since they correspond to the use of a coarser finite element mesh. The use of the additional layer of refined leaf elements around the subdomains ensures that the block A_s appears in (2.7) and (2.8) in its original form (i.e. as in (2.6)).

To obtain an approximate solution to the fine mesh problem (2.6) one may solve the two smaller problems (2.7) and (2.8) concurrently and then set

$$\begin{bmatrix} \underline{u}_1^{(1)} \\ \underline{u}_2^{(1)} \\ \underline{u}_s^{(1)} \end{bmatrix} = \begin{bmatrix} \underline{u}_{1,1} \\ \underline{u}_{2,2} \\ \frac{1}{2}(\underline{u}_{s,1} + \underline{u}_{s,2}) \end{bmatrix} . \tag{2.9}$$

Note that a communication between the two processors is required in order to evaluate the average $\frac{1}{2}(\underline{u}_{s,1} + \underline{u}_{s,2})$.

Having obtained this approximation to the solution the corresponding residual may now be calculated using the identity

$$\underline{r}^{(1)} = \begin{bmatrix} \underline{r}_1^{(1)} \\ \underline{r}_2^{(1)} \\ \underline{r}_s^{(1)} \end{bmatrix} = \begin{bmatrix} \underline{f}_1 \\ \underline{f}_2 \\ \underline{f}_s \end{bmatrix} - \begin{bmatrix} A_1 & 0 & B_1 \\ 0 & A_2 & B_2 \\ C_1 & C_2 & A_s \end{bmatrix} \begin{bmatrix} \underline{u}_1^{(1)} \\ \underline{u}_2^{(1)} \\ \underline{u}_s^{(1)} \end{bmatrix} . \tag{2.10}$$

This may also be achieved in parallel since

$$\underline{r}_1^{(1)} = \underline{f}_1 - A_1 \underline{u}_1^{(1)} - B_1 \underline{u}_s^{(1)} \tag{2.11}$$

for which all necessary data is present on processor 1, and

$$\underline{r}_2^{(1)} = \underline{f}_2 - A_2 \underline{u}_2^{(1)} - B_2 \underline{u}_s^{(1)} \tag{2.12}$$

for which all necessary data is present on processor 2. Also,

$$\underline{r}_s^{(1)} = \underline{f}_s - \sum_{i=1}^{2} \left(C_i \underline{u}_i^{(1)} - A_{s(i)} \underline{u}_s^{(1)} \right) \tag{2.13}$$

where $A_{s(i)}$ is the contribution to A_s which is obtained by restricting integration to subdomain i only. (This may be assembled at no extra computational cost at the same time that A_s is being assembled on processor i.) Note that although each term in the sum may be computed concurrently by each of the processors a further communication is required to add these two contributions. In general this residual will be non-zero and so it is necessary to form a fixed point iteration based upon solution of the error equation:

$$K\underline{e}^{(k)} = \underline{r}^{(k)} \tag{2.14}$$

$$\underline{u}^{(k+1)} = \underline{u}^{(k)} + \underline{e}^{(k)} . \tag{2.15}$$

The algorithm for this fixed point iteration is shown in Figure 2. Note that it is necessary to restrict $\underline{r}_1^{(k)}$ to the coarse mesh covering subdomain 1 on processor 2 and $\underline{r}_2^{(k)}$ to the coarse mesh covering subdomain 2 on processor 1. This is done using the rectangular matrices M_1 and M_2 respectively, which make use of the hierarchical data structure which is present by Assumption 2.3. These matrices are discussed further over the next two subsections.

1/. Initialise: $k = 0$

$$\underline{u}_1^{(0)} = \underline{0}; \quad \underline{u}_2^{(0)} = \underline{0}; \quad \underline{u}_s^{(0)} = \underline{0}$$

$$\underline{r}_1^{(0)} = \underline{f}_1; \quad \underline{r}_2^{(0)} = \underline{f}_2; \quad \underline{r}_s^{(0)} = \underline{f}_s$$

2/. Repeat

2.1 $$\begin{bmatrix} A_1 & 0 & B_1 \\ 0 & \tilde{A}_2 & \tilde{B}_2 \\ C_1 & \tilde{C}_2 & A_s \end{bmatrix} \begin{bmatrix} \underline{z}_{1,1} \\ \underline{z}_{2,1} \\ \underline{z}_{s,1} \end{bmatrix} = \begin{bmatrix} \underline{r}_1^{(k)} \\ M_2 \underline{r}_2^{(k)} \\ \underline{r}_s^{(k)} \end{bmatrix}$$

2.2 $$\begin{bmatrix} \tilde{A}_1 & 0 & \tilde{B}_1 \\ 0 & A_2 & B_2 \\ \tilde{C}_1 & C_2 & A_s \end{bmatrix} \begin{bmatrix} \underline{z}_{1,2} \\ \underline{z}_{2,2} \\ \underline{z}_{s,2} \end{bmatrix} = \begin{bmatrix} M_1 \underline{r}_1^{(k)} \\ \underline{r}_2^{(k)} \\ \underline{r}_s^{(k)} \end{bmatrix}$$

2.3 $$\begin{bmatrix} \underline{z}_1 \\ \underline{z}_2 \\ \underline{z}_s \end{bmatrix} = \begin{bmatrix} \underline{z}_{1,1} \\ \underline{z}_{2,2} \\ \frac{1}{2}(\underline{z}_{s,1} + \underline{z}_{s,2}) \end{bmatrix}$$

2.4 $$\begin{bmatrix} \underline{u}_1^{(k+1)} \\ \underline{u}_2^{(k+1)} \\ \underline{u}_s^{(k+1)} \end{bmatrix} = \begin{bmatrix} \underline{u}_1^{(k)} \\ \underline{u}_2^{(k)} \\ \underline{u}_s^{(k)} \end{bmatrix} + \begin{bmatrix} \underline{z}_1 \\ \underline{z}_2 \\ \underline{z}_s \end{bmatrix}$$

2.5 $$\begin{bmatrix} \underline{r}_1^{(k+1)} \\ \underline{r}_2^{(k+1)} \\ \underline{r}_s^{(k+1)} \end{bmatrix} = \begin{bmatrix} \underline{f}_1 \\ \underline{f}_2 \\ \underline{f}_s \end{bmatrix} - \begin{bmatrix} A_1 & 0 & B_1 \\ 0 & A_2 & B_2 \\ C_1 & C_2 & A_s \end{bmatrix} \begin{bmatrix} \underline{u}_1^{(k+1)} \\ \underline{u}_2^{(k+1)} \\ \underline{u}_s^{(k+1)} \end{bmatrix}$$

2.6 $k \mathrel{+}= 1$

Until $\|\underline{r}^{(k)}\| \leq TOL$

Figure 2: An algebraic description of the 2 subdomain version of the fixed point iteration for the solution of (2.3) (which may be partitioned as (2.6)).

2.2 Generalization of the algorithm

Having introduced the fixed point iteration for $p = 2$ it is now possible to generalize this to arbitrary choices of p. The mesh generation and matrix assembly on each processor are unaltered, and we will again work with Assumption 2.4 for the time-being. Hence we may define a global fine mesh as before and the corresponding Galerkin finite element equations, (2.3), may be partitioned as

$$\begin{bmatrix} A_i & 0 & B_i \\ 0 & \bar{A}_i & \bar{B}_i \\ C_i & \bar{C}_i & A_{i,s} \end{bmatrix} \begin{bmatrix} \underline{u}_i \\ \underline{\bar{u}}_i \\ \underline{u}_{i,s} \end{bmatrix} = \begin{bmatrix} \underline{f}_i \\ \underline{\bar{f}}_i \\ \underline{f}_{i,s} \end{bmatrix} \tag{2.16}$$

for any choice of $i \in \{1, ..., p\}$. Here $\underline{u}_i$ is a vector of fine mesh nodal values inside subdomain i, $\underline{\bar{u}}_i$ is a vector of fine mesh nodal values outside subdomain i and $\underline{u}_{i,s}$ are the remaining fine mesh nodal values, on the interface of subdomain i. The rest of the partition into blocks follows from this.

As with the case $p = 2$ it is only possible to fully assemble the finite element equations for the meshes actually generated on each processor. For processor i these may be written as

$$\begin{bmatrix} A_i & 0 & B_i \\ 0 & \tilde{A}_i & \tilde{B}_i \\ C_i & \tilde{C}_i & A_{i,s} \end{bmatrix} \begin{bmatrix} \underline{u}_i \\ \tilde{\underline{u}}_i \\ \underline{u}_{i,s} \end{bmatrix} = \begin{bmatrix} \underline{f}_i \\ \tilde{\underline{f}}_i \\ \underline{f}_{i,s} \end{bmatrix}, \tag{2.17}$$

where the tilde above a block again indicates that it is smaller than the corresponding block in (2.16) due to the use of a coarser finite element mesh. Now, by introducing the restriction operator M_i, from the part of the global fine mesh outside subdomain i to the coarser mesh covering the same region on processor i, the overall fixed point iteration shown in Figure 2 may be generalized to p processors. This is done in Figure 3. Note that in this figure $A_{i,s(i)}$ and $\underline{f}_{i,s(i)}$ represent the contributions to $A_{i,s}$ and $\underline{f}_{i,s}$ respectively obtained by restricting integration to subdomain i only. Since the latter terms are typically assembled from each element in turn there is no additional computational overhead associated with accumulating the partial assemblies $A_{i,s(i)}$ and $\underline{f}_{i,s(i)}$ as well.

2.3 Parallel implementation issues

The algorithm of Figure 3 has been developed with a distributed memory programming model in mind and a straightforward parallel implementation may be obtained with calls to only a small number of communication subprograms (using the Message Passing Interface (MPI), [29], for example). In this subsection we discuss the main features of such an implementation, making frequent reference to the steps enumerated in Figure 3 and Assumptions 2.1 to 2.4 (which are discussed and justified in Subsection 2.4 below).

Step 1 is the parallel mesh generation phase. By Assumption 2.1 there exists some error indicator upon which to base the local refinement of each mesh (which may be undertaken independently on each processor by Assumption 2.3), and by Assumption 2.2 we may reasonably expect that the meshes generated will all have a similar number of elements (since the error per element will be approximately equal to some fixed target value and the total error per subdomain is approximately equal). If Assumption 2.4 is also valid then no inter-processor communication at all is required in the parallel generation of these matching, load-balanced meshes. Again we emphasize that, as in Figure 1, although each mesh covers the entire domain the vast majority of the nodes and elements are located in and around subdomain i.

The assembly of the finite element equations (Step 2) may clearly be completed in parallel without the need for inter-processor communication since each processor works only with its own mesh. In practice the stiffness matrix should be stored using a sparse data structure (as in [7] for example), with only a small amount of additional memory required for the separate storage of $A_{i,s(i)}$ and $\underline{f}_{i,s(i)}$. In our implementation we also compute a sparse incomplete LU factorization of the stiffness matrix on each processor at this stage, to be used as a preconditioner in the iterative solution of the system at Step 4.1. This too may be completed independently on each processor using standard sequential algorithms (e.g. [8, 9]).

The initialization of Step 3 also requires no inter-processor communication. It should be noted that initialization of $\tilde{\underline{u}}_i^{(0)}$ and $\tilde{\underline{r}}_i^{(0)}$ is not strictly necessary in the version of the algorithm given in Figure 3 since they are not used in subsequent steps. A slight modification of the algorithm however would be to use $\tilde{\underline{r}}_i^{(0)}$ instead of $M_i\bar{\underline{r}}_i^{(0)}$ at the first pass of Step 4.1 in order to avoid some global communication (see below). This corresponds to solving (2.17) on each processor at the first pass of Step 4.1.

1/. Generate the mesh for processor i

2/. Assemble the FE equations (2.17) (saving $A_{i,s(i)}$ and $\underline{f}_{i,s(i)}$ separately for later use)

3/. Initialise: $k = 0$

$$\underline{u}_i^{(0)} = \underline{0}; \quad \underline{\tilde{u}}_i^{(0)} = \underline{0}; \quad \underline{u}_{i,s}^{(0)} = \underline{0}$$

$$\underline{r}_i^{(0)} = \underline{f}_i; \quad \underline{\tilde{r}}_i^{(0)} = \underline{\tilde{f}}_i; \quad \underline{r}_{i,s}^{(0)} = \underline{f}_{i,s}$$

4/. Repeat

4.1
$$\begin{bmatrix} A_i & 0 & B_i \\ 0 & \tilde{A}_i & \tilde{B}_i \\ C_i & \tilde{C}_i & A_{i,s} \end{bmatrix} \begin{bmatrix} \underline{z}_i \\ \underline{\tilde{z}}_i \\ \underline{z}_{i,s} \end{bmatrix} = \begin{bmatrix} \underline{r}_i^{(k)} \\ M_i \underline{\bar{r}}_i^{(k)} \\ \underline{r}_{i,s}^{(k)} \end{bmatrix}$$

4.2 Let each entry of $\underline{z}_{i,s}$ be averaged over all corresponding entries found on neighbouring processors

4.3 $\underline{u}_i^{(k+1)} = \underline{u}_i^{(k)} + \underline{z}_i; \quad \underline{u}_{i,s}^{(k+1)} = \underline{u}_{i,s}^{(k)} + \underline{z}_{i,s}$

4.4 $\underline{r}_i^{(k+1)} = \underline{f}_i - A_i \underline{u}_i^{(k+1)} - B_i \underline{u}_{i,s}^{(k+1)}; \quad \underline{r}_{i,s}^{(k+1)} = \underline{f}_{i,s(i)} - C_i \underline{u}_i^{(k+1)} - A_{i,s(i)} \underline{u}_{i,s}^{(k+1)}$

4.5 Let each entry of $\underline{r}_{i,s}^{(k+1)}$ be summed over all corresponding entries found on neighbouring processors

4.6 $k \mathrel{+}= 1$

Until $\|\underline{r}^{(k)}\| \leq TOL$

Figure 3: An algebraic description of the algorithm to be followed by processor i (from 1 to p) in the p subdomain version of the fixed point iteration for the solution of (2.3) (which may be partitioned as (2.16)).

Step 4.1 is the most complex step of the algorithm from a parallel programming point of view. Obtaining the right-hand side of the linear system requires global (all-to-all) communication between the processors and consequently needs to be implemented as efficiently as possible. Due to the importance of this implementation we postpone a detailed description until the end of this discussion of the overall algorithm. Once the right-hand side of each linear system has been obtained on each processor however the systems may clearly be solved independently using standard sequential algorithms. Since the stiffness matrices are sparse it is appropriate to use an iterative solver for these equations (e.g. [3]), possibly with an ILU factorization (computed at Step 2) as a preconditioner, and so the issue of convergence needs to be considered. Experiments suggest that a fairly large convergence tolerance is optimal (see Section 4 for further details).

The next step, 4.2, also requires inter-processor communication, but only locally between processors owning neighbouring subdomains (i.e. subdomains which share one or more vertex). Each processor sends to each of its neighbours a list of those entries of $\underline{z}_{i,s}$ which correspond to vertices shared with that neighbour. Such a list is also received from each neighbour and each item of data is added to the corresponding entry in $\underline{z}_{i,s}$. To obtain an average, each entry in $\underline{z}_{i,s}$ is divided by a positive integer, $m_{i,s}(j)$ say, where $m_{i,s}(j)$ is equal to the number of subdomains which share the vertex corresponding to the j^{th} entry of $\underline{z}_{i,s}$ on processor i. Note that $m_{i,s}(j)$ can be computed after Step 1 of the algorithm without any inter-processor communication since, for 2-d meshes, $m_{i,s}(j) = 2$ for any node on the boundary of subdomain i which is not in the original coarse mesh and, by Assumption 2.2, the value of $m_{i,s}(j)$ may easily be determined for any node present in the original coarse mesh.

Steps 4.3 and 4.4 are clearly local operations on each processor and Step 4.5 may be completed using the same local communication routines required for Step 4.2. (It should be noted that this neighbour-to-neighbour communication pattern occurs in most distributed memory parallel finite element codes (e.g. [20, 24]) and so is well-understood.)

The final step of the algorithm in Figure 3 is the test for convergence. This too requires a (small) global communication, as well as a synchronization. In our implementation we use the 2-norm of the residual and make use of the MPI_Allreduce subprogram. Each processor first accumulates

$$\|\underline{r}_i\|_2^2 + \sum_{j=1}^{\dim(\underline{r}_{i,s})} \frac{(\underline{r}_{i,s})_j^2}{m_{i,s}(j)} \tag{2.18}$$

(where $(\underline{r}_{i,s})_j$ is the j^{th} entry of $\underline{r}_{i,s}$ for $j = 1$ to $\dim(\underline{r}_{i,s})$), and then the global sum is taken in order to determine the square of the 2-norm of the residual on the global fine mesh.

For the rest of this section we return to the formation of the right-hand side vector in Step 4.1 of the algorithm. Once the issues associated with the accumulation of this vector on each processor have been satisfactorily resolved the detailed discussion of the parallel implementation issues will be complete.

In order to form $M_i\bar{\underline{r}}_i^{(k)}$, processor i needs to know the restriction of $(\underline{r}_j^{(k)}, \underline{r}_{j,s}^{(k)})$ to the part of mesh i which covers subdomain j (for $j = 1$ to p but $j \neq i$). By calculating this restriction on processor j and then sending it on to processor i the length of the message that must be sent from j to i is equal to the number of vertices in mesh i which lie in subdomain j or on its boundary. Note that as the global fine mesh is refined this length does not increase very significantly (if at all) since it depends mainly on the size of the coarse starting mesh in subdomain j (plus a small amount of additional refinement on part of the edge of subdomain j if j is a neighbour of i). Hence the major programming issue that must be resolved is that of how processor j is able to calculate the restriction of $(\underline{r}_j^{(k)}, \underline{r}_{j,s}^{(k)})$ to the part of mesh i which covers subdomain j. Once this is done, assembly of the right-hand side of the system in Step 4.1 may be completed on processor i as soon as it has received a contribution to the restriction $M_i\bar{\underline{r}}_i^{(k)}$ from each of the other processors. (Note that processor j must use the value of $\underline{r}_{j,s}^{(k)}$ calculated at Step 4.4, not 4.5, when performing this restriction since nodes on the interface with other processors will be counted more than once.)

For processor j to restrict $(\underline{r}_j^{(k)}, \underline{r}_{j,s}^{(k)})$ to the part of mesh i which covers subdomain j this part of mesh i must be communicated to processor j in a pre-processing step. This may occur any time after the completion Step 1 of the algorithm (and simultaneously with Step 2 if desired). Once j has received this mesh from i it must identify which of the vertices of its own mesh correspond to the vertices received (by Assumptions 2.3 and 2.4 each vertex received must match the location of a vertex that has been generated on processor j). A naive way of implementing this process for each vertex received would be to search through each vertex in mesh j which lies in subdomain j, or on its boundary, until the location matches that of the received vertex. This would be very inefficient as the global fine mesh is refined however since one would be searching through nearly all of the nodes on mesh j. By including a *node level* field in the hierarchical mesh data structure however this search may be trimmed significantly since a node generated at level m of the hierarchical refinement need only be compared against nodes of the same level on processor j. Further efficiency gains may be obtained for the larger values of m by searching through the nodes on j which are closest to the boundary with i first. Once this matching process is complete for each vertex received from processor i, the hierarchical data structure on processor j may be used to calculate the restriction of $(\underline{r}_j^{(k)}, \underline{r}_{j,s}^{(k)})$ onto mesh i at each iteration.

2.4 Discussion

We conclude this section with a brief discussion of the underlying Assumptions (2.1 to 2.4) that have been made in order to justify the parallel algorithm that we have introduced. A numerical study of the performance of the algorithm is postponed until Sections 3 and 4, where it is respectively demonstrated that the technique appears to show good conditioning properties when the global finite element mesh is uniformly refined and that an efficient parallel implementation may be achieved.

The main motivations for the DD method proposed here come from the *full domain parallel multigrid* approach of [30, 31] and the parallel adaptive meshing paradigm of [5]. This latter approach to undertaking parallel adaptive finite element computations addresses the load-balancing problem in a new way, requiring less communication than existing techniques, and also allows existing adaptive PDE codes, such as [4, 16, 34], to run in a parallel environment with only a small amount of recoding. There are three main components.

1. The solution of the small finite element system that is obtained by approximating the problem of interest on a coarse initial mesh, followed by the use of *a posteriori* error estimates to partition the mesh. This partition is undertaken such that each subregion has about the same total approximate error, and so the size of these subregions could vary considerably in terms of numbers of coarse elements or grid points.

2. Each processor is provided the complete coarse mesh and instructed to sequentially solve the *entire* problem, with the stipulation that its adaptive refinement should be limited largely to its own subdomain. The target number of elements and grid points for each problem is the same.

3. A final mesh is made up of the union of the refined subdomains provided by each processor. This mesh is regularized and a final solution computed, using a parallel domain decomposition or multigrid technique.

This approach has a number of interesting features, such as the reduction of the load-balancing problem to the numerical solution of a small elliptic problem on a single processor for example (but see also Section 4 below), and is justified in detail in [5]. In particular however the new parallel DD method outlined in this section is ideally suited for the computation of the final, global, solution required by component 3 above. Moreover, by following this approach each of the first three Assumptions (2.1 to 2.3) will be automatically satisfied.

In order to satisfy Assumption 2.4 it is proposed in [5] that a conforming global fine mesh be made from each of the subdomains through the use of local communication between processors owning neighbouring subdomains, followed by a small amount of additional local refinement where necessary. An alternative to this comes from the observation that, in the algorithm used in this paper, the small overlap in the regions that are refined by each processor means that it is quite unusual for Assumption 2.4 not to be satisfied automatically (given a reliable error indicator and in the absence of r-refinement). Hence, only if a match cannot be found for a boundary node when completing the pre-processing required for the computations of the right-hand sides in Step 4.1, should any further mesh modification be undertaken. This will save unnecessary neighbour-to-neighbour communications. If one wishes to make use of local node movement (r-refinement) to improve mesh quality (as in [4, 16] for example) then this may be undertaken in parallel after the pre-processing for Step 4.1 has been completed. We have yet to investigate such an approach in detail however.

3 Convergence and acceleration

The parallel DD solver introduced in the previous section is designed to make use of standard sequential algorithms and software as much as possible. This includes code for adaptive meshing, finite element assembly, solution of sparse systems and *a posteriori* error estimation. In this section we justify the algorithm both through an empirical study of its convergence properties and by considering convergence acceleration using Krylov subspace techniques. Some related mathematical theory is also discussed as a means of further justification.

3.1 Convergence of the algorithm

We consider the algorithm outlined in Figure 3 when applied to two simple test problems.

Problem 1

$$-\underline{\nabla} \cdot (\underline{\nabla} u) = f \ .$$

Problem 2

$$-\underline{\nabla} \cdot (\underline{\nabla} u) + \left[\begin{array}{c} 1 \\ 1 \end{array} \right] \cdot \underline{\nabla} u = f \ .$$

In each case the domain $\Omega = (0,1) \times (0,1)$ and Dirichlet boundary conditions are applied throughout $\partial\Omega$. These are chosen, along with the source term f, so that the exact solution is given by

$$u(\underline{x}) = (x_1 - \frac{1}{2})^2 (x_2 - \frac{1}{2})^2 \ . \tag{3.1}$$

Table 1 shows the performance of the algorithm when $TOL = 10^{-6} \|\underline{r}^{(0)}\|_2$ and the systems encountered at step 4.1 in Figure 3 are solved exactly at each iteration. For each calculation a coarse mesh of just 64 elements has been used and the global fine meshes contain between 1024 and 1048576 elements (representing between 2 and 7 levels of uniform refinement respectively of the coarse mesh). Between 2 and 32 subdomains have been considered (corresponding to the use of between 2 and 32 processors in a parallel implementation — although discussion of parallel performance is postponed until Section 4).

It is apparent from the results contained in Table 1 that the fixed point iteration proposed converges very rapidly and (for these examples at least) in a manner which is independent of the size (h say) of the fine mesh. Moreover, the algorithm performs just as well on the non-self-adjoint

Elements/Procs.	Problem 1					Problem 2				
	2	4	8	16	32	2	4	8	16	32
1024	3	4	4	4	4	3	4	4	4	5
4096	3	4	4	5	4	3	4	5	5	5
16384	3	4	4	5	5	3	4	5	5	5
65536	3	3	4	5	5	3	4	5	5	5
262144	3	3	4	5	5	3	4	5	5	5
1048576	3	3	4	5	5	3	4	5	5	5

Table 1: The performance of the proposed algorithm on two test problems: figures quoted represent the number of iterations required to reduce the initial residual by a factor of 10^6.

problem (Problem 2) as on the self-adjoint one; at least in this case, where the convection term does not dominate. It would also appear that the rate of convergence of the algorithm is only very weakly dependent upon p, if at all. This observation, along with the apparent independence from the mesh size, h, leads one to suspect connections between this approach and that of optimal additive Schwartz algorithms (see, for example, [17, 22, 35, 36]). This connection is discussed in Subsection 3.3 below.

3.2 Acceleration of convergence

The use of Krylov subspace methods ([3, 21]) to accelerate the convergence of fixed point iterations, such as that proposed in Figure 3, is quite standard. In this subsection we propose using a parallel GMRES ([32]) algorithm to solve (2.3) (partitioned as in (2.17) on each processor still) using steps 4.1 to 4.2 of Figure 3 as a preconditioner. This requires only very minor modification to the parallel code described in the previous section: and, in particular, the data structures and partition of the data are identical. (Furthermore, the code for the parallel matrix-vector products (step 4.4) and the parallel inner products (2.18) in the fixed point algorithm is directly re-used in the generalization to preconditioned GMRES.)

The performance of the fixed point algorithm is very good when it is applied to Problems 1 and 2 (see Table 1). Hence, the application of the corresponding preconditioned GMRES algorithm to these two problems gives results that are very similar: with iteration counts being either one fewer or exactly the same for each computation. A more demanding test problem is the following anisotropic diffusion equation.

Problem 3

$$-\underline{\nabla} \cdot \left(\begin{bmatrix} \mathit{100} & \mathit{0} \\ \mathit{0} & \mathit{1} \end{bmatrix} \underline{\nabla} u \right) = f \; .$$

Once more the domain is $\Omega = (0,1) \times (0,1)$, Dirichlet boundary conditions are applied throughout $\partial\Omega$ and f is chosen so as to give the exact solution (3.1).

Table 2 shows the number of iterations required for both the original and the accelerated versions of the algorithm to solve this example on various meshes with different numbers of subdomains. Again, a convergence tolerance of $10^{-6}\|\underline{r}^{(0)}\|_2$ is used and the subproblems at step 4.1 in Figure 3 are solved exactly at each iteration. The cost of a single iteration is almost identical for both versions of the algorithm.

Elements/Procs.	Original					Precon. GMRES				
	2	4	8	16	32	2	4	8	16	32
1024	5	6	11	11	11	5	6	7	8	9
4096	7	7	10	11	12	5	6	8	9	10
16384	8	8	11	16	14	6	7	9	11	11
65536	9	8	12	24	17	6	7	9	12	12
262144	10	9	14	32	20	6	7	10	12	13
1048576	10	9	14	39	23	6	8	10	13	14

Table 2: The performance of the original and the preconditioned GMRES versions of the proposed algorithm on Problem 3: figures quoted represent the number of iterations required to reduce the initial residual by a factor of 10^6.

It is now less apparent that the fixed point algorithm has a performance (in terms of the iteration count) that is independent of the fine mesh size h. For small values of p (2, 4 and 8) the number of iterations appears to have stopped growing between the 6th and 7th refinements, however this is not yet apparent when $p = 16$ or 32. Also, for this anisotropic example, the performance of the fixed point algorithm appears to be much more sensitive to the number of subdomains than for the isotropic diffusion example (Problem 1). Figure 4 shows the partition of the coarse mesh into 16 and 32 subdomains respectively and, for this particular example, it appears that the latter is a better partition than the former. Other partitions into 16 subdomains are possible of course and it is to be expected that the rate of convergence when solving highly anisotropic problems such as this will depend upon the precise decomposition that is used. This fact has been noted in the context of more conventional domain decomposition solvers by Keyes *et al.*, [26], for example who suggest that in order to achieve petaflops computing "partitioning... must adapt to coefficients (grid spacing and flow magnitude and direction) for convergence rate improvement". This important issue of how to obtain the most appropriate partitions of a mesh is beyond the scope of this paper however.

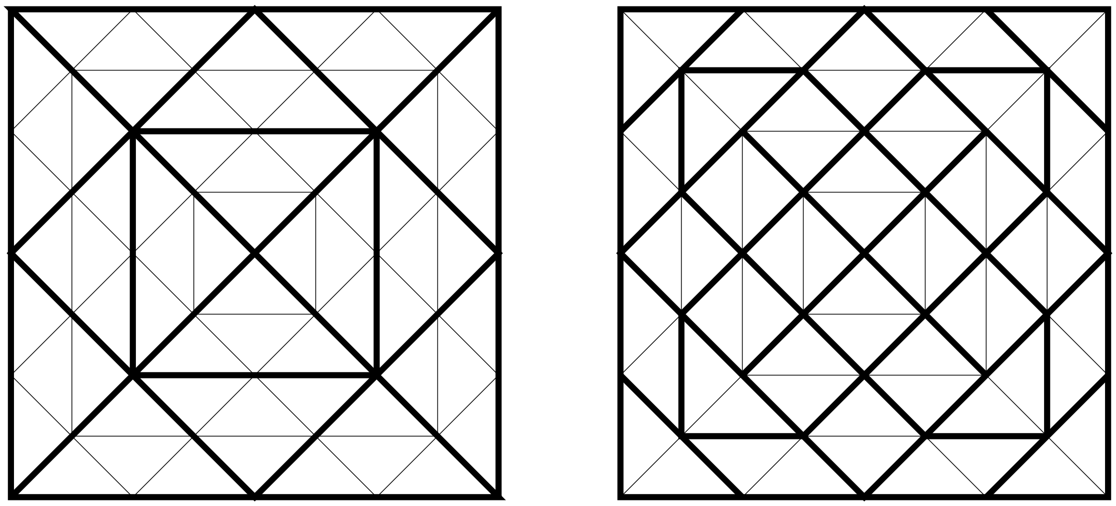

Figure 4: The partition of a 64 element coarse mesh into 16 and 32 subdomains.

Use of the preconditioned GMRES version of the algorithm is clearly seen, from Table 2, to be considerably superior to the original fixed point scheme described in Section 2. In all but the most trivial cases, where the iteration count is unaltered, the GMRES solver requires substantially fewer iterations when used to solve Problem 3 (with an almost identical cost per iteration). Even more significantly however, the rate of increase in the number of iterations appears to be substantially slower as the mesh size, h, is decreased. In fact, it again appears to be the case that the convergence rate may well be independent of h as $h \to 0$.

The dependence of the convergence rate on p, the number of subdomains, is harder to speculate on from this single example however. This is due, at least in part, to the dependence upon the shape of the subdomains mentioned above. Nevertheless, it does appear that the preconditioned GMRES approach is more robust in the sense that the performance seems to be less affected by the apparently poor quality of the partition when $p = 16$ than the original fixed point iteration.

3.3 Comparison with analytic results

We have now presented a new parallel domain decomposition algorithm that is applicable to the wide class of PDE (2.1). Furthermore, the empirical evidence presented in the preceding subsections suggests that the performance of this algorithm as a GMRES preconditioner is nearly optimal. To see why this may be the case it is informative to contrast the proposed technique with more classical DD solvers. To this end, for the remainder of this subsection we restrict our consideration to the particular case where $\underline{b} = \underline{0}$ in (2.1), whereupon the equation is self-adjoint. In this situation it follows that the stiffness matrix (K in (2.3)), given by (2.4), is symmetric and positive-definite (SPD). It is therefore possible to solve (2.3) using the preconditioned conjugate gradient (PCG) algorithm which is more efficient than GMRES in the sense that the search vectors which span the Krylov subspace at each iteration are defined via a two-term recurrence relation (as opposed to a k-term recurrence at iteration k with GMRES).

It is important to note however that in order to solve a SPD system using the PCG algorithm the preconditioner must itself be a SPD matrix. This is not the case for the preconditioner proposed in Subsection 3.2 however, even when $\underline{b} = \underline{0}$. To illustrate this one need only consider the simplest case of $p = 2$ where the preconditioner, M say, is given by

$$M^{-1} = \begin{bmatrix} I \\ 0 \\ \frac{1}{2}I \end{bmatrix} \begin{bmatrix} A_1 & 0 & B_1 \\ 0 & \tilde{A}_2 & \tilde{B}_2 \\ B_1^T & \tilde{B}_2^T & A_s \end{bmatrix}^{-1} \begin{bmatrix} I \\ M_2 \\ I \end{bmatrix} + \begin{bmatrix} 0 \\ I \\ \frac{1}{2}I \end{bmatrix} \begin{bmatrix} \tilde{A}_1 & 0 & \tilde{B}_1 \\ 0 & A_2 & B_2 \\ \tilde{B}_1^T & B_2^T & A_s \end{bmatrix}^{-1} \begin{bmatrix} M_1 \\ I \\ I \end{bmatrix}. \tag{3.2}$$

Note that in (3.2) the symmetry of K when $\underline{b} = \underline{0}$ implies that

$$\begin{aligned} C_1 &= B_1^T \\ \tilde{C}_1 &= \tilde{B}_1^T \\ C_2 &= B_2^T \\ \tilde{C}_2 &= \tilde{B}_2^T \end{aligned}$$

in the notation of Figure 2. Despite this fact it is clear from (3.2) that M is not generally a symmetric matrix and cannot therefore be used as a preconditioner in the PCG algorithm. Hence, even when $\underline{b} = \underline{0}$, the preconditioner proposed in Subsection 3.2 should always be used with a GMRES (or similar) solver. At first sight this may appear to be a drawback; however the results presented below (in Table 3) suggest that this is not the case.

In [6], an analysis is presented of a symmetric version of the preconditioner proposed here given by, in the case $p = 2$ (for simplicity of presentation),

$$M^{-1} = \begin{bmatrix} I \\ M_2 \\ I \end{bmatrix}^T \begin{bmatrix} A_1 & 0 & B_1 \\ 0 & \tilde{A}_2 & \tilde{B}_2 \\ B_1^T & \tilde{B}_2^T & A_s \end{bmatrix}^{-1} \begin{bmatrix} I \\ M_2 \\ I \end{bmatrix} + \begin{bmatrix} M_1 \\ I \\ I \end{bmatrix}^T \begin{bmatrix} \tilde{A}_1 & 0 & \tilde{B}_1 \\ 0 & A_2 & B_2 \\ \tilde{B}_1^T & B_2^T & A_s \end{bmatrix}^{-1} \begin{bmatrix} M_1 \\ I \\ I \end{bmatrix} . \quad (3.3)$$

The results of this analysis show that the preconditioner is optimal in the sense that the number of PCG iterations required to solve (2.3) (with $\underline{b} = \underline{0}$ and $c = 0$ in (2.4)) is independent of both h (as $h \to 0$) and p (as $p \to \infty$). Table 3 shows the performance of this symmetric preconditioner on the two diffusion examples above: Problem 1 and Problem 3. We refer to this as the additive Schwartz variant of our proposed algorithm.

	Problem 1					Problem 3				
Elements/Procs.	2	4	8	16	32	2	4	8	16	32
1024	7(3)	9(3)	13(4)	16(4)	18(4)	8(5)	12(6)	17(7)	21(8)	24(9)
4096	7(3)	9(3)	14(4)	16(4)	22(4)	9(5)	13(6)	17(8)	21(9)	27(10)
16384	7(3)	8(3)	13(4)	16(4)	21(5)	9(6)	14(7)	17(9)	23(11)	27(11)
65536	6(3)	8(3)	12(4)	15(4)	21(5)	9(6)	14(7)	18(9)	25(12)	28(12)
262144	6(3)	8(3)	12(4)	15(4)	19(5)	8(6)	14(7)	18(10)	25(12)	29(13)
1048576	6(3)	7(3)	11(4)	14(4)	19(5)	8(6)	13(8)	18(10)	24(13)	29(14)

Table 3: The performance of the Additive Schwartz variant of the proposed algorithm on two self-adjoint test problems: figures quoted represent the number of iterations required to reduce the 2-norm of the initial residual by a factor of 10^6 (and figures in brackets are the equivalent number of GMRES iterations when using the preconditioner of Subsection 3.2).

The independence of the number of iterations from h as $h \to 0$ can clearly be seen in Table 3 however the lack of dependency on p is not so apparent for the relatively small values used here. Comparison with the number of GMRES iterations required to solve the same problems using (3.2) (and its generalizations to $p = 4, 8, 16$ and 32) clearly shows the advantage of the latter approach, despite the slightly increased cost at each iteration. Furthermore, the analytical results in [6], which apply to the additive Schwartz variant of the algorithm, might reasonably be used to provide some (although certainly not rigorous) theoretical basis for the parallel DD solver introduced here.

4 Parallel performance

In this section we attempt to assess the parallel performance of the proposed domain decomposition preconditioner by considering a specific implementation of the preconditioned GMRES algorithm using MPI (message passing interface [29]). We begin with a short introduction and then focus on two representative test problems: one which uses uniform mesh refinement and the other using local mesh refinement. All calculations reported took place on a 32 processor SG Origin 2000 computer which has a NUMA (non-uniform memory access) virtual shared memory architecture. The non-uniform nature of the memory access means that, even when there are no other users on the machine, timings of the same run may vary by a few percent according to how memory has been allocated. For this reason, all timings quoted in the section represent the best time that was achieved over five consecutive repetitions of the same computation (always in single-user mode).

4.1 Assessing parallel performance

One of the simplest metrics for assessing the quality of a parallel program is to consider the *speedup* that it provides when solving a single problem on p processors. Here, speedup is defined to be the time required to solve the problem using the best available algorithm (and implementation) on a single processor divided by the time required to solve the problem using the parallel algorithm on p processors. This is the metric that we use in this section.

In order to apply this metric it is first necessary to establish the best available sequential solver for the problems that we consider here. This is of course a highly non-trivial issue and the best sequential algorithm could well vary from one problem to another within the wide class of PDEs defined by equation (2.1). It is likely however that for many such PDEs the best sequential algorithm will be based upon multigrid in some way and so we have used as our benchmark sequential code a generalized conjugate gradient solver with a multilevel ILU preconditioner similar to that described in [9]. This sequential solver is the same one that is used for the sparse linear systems that must be solved on each processor at step 4.1 of the parallel algorithm given in Figure 3 and contains a number of parameters to control the amount of fill-in (via a drop tolerance) and the maximum number of hierarchical levels permitted. Whenever a sequential time is quoted it is the best time that we were able to obtain for a range of different choices of these parameters. Similarly, for the parallel timings a range of choices for these parameters were also considered in order to permit the best times to be recorded.

A further issue that must be addressed when obtaining parallel results is the accuracy to which it is necessary to solve the systems at step 4.1 in Figure 3. If these systems are solved very accurately then unnecessary time is wasted; however highly inaccurate solutions lead to the number of GMRES iterations increasing significantly. Generally, a reduction in the 2-norm of the residual by a factor of 10^2 appears to give optimal or near-optimal solution times however, in the timings which follow, the figures quoted are always the best ones obtained over a range of different test values for the reduction in the 2-norm of the residual.

In addition to comparing the parallel solution time on p processors with the best sequential solution time it is also informative to compare with the time taken by a sequential version of the p-subdomain preconditioned GMRES solver. Whilst this does not provide a true speedup figure it does demonstrate clearly the level of parallelism achieved by the p-subdomain solver (we will refer to this figure as the *parallel speedup*). Moreover, it also allows one to assess the quality of the p-subdomain preconditioner itself by comparing this sequential time with that obtained for other choices of p (and by comparing with the best sequential time). In some cases it may be seen that the sequential time using p subdomains is greater than the sequential time using q subdomains, where q is some integer multiple of p. In such situations it may be possible to obtain a better parallel time on p processors by using the q subdomain version of the algorithm and so we define the *optimal speedup* to be the speedup ratio corresponding to the best parallel time on p processors using q subdomains (where q may be any integer multiple of p between p and 32 inclusive).

4.2 An example with uniform mesh refinement

For this first assessment of the parallel performance of our proposed domain decomposition preconditioner we return to Problem 2, defined in subsection 3.1. This is solved on a mesh of 1048576 triangular elements which is a uniform refinement of an initial coarse grid of 256 congruent triangular elements. (Note that the choice of 256 elements in the coarse mesh (rather than 64 say) makes little difference to the quality of the preconditioner but does allow more flexibility when selecting a partition into p subdomains.)

Table 4 shows the time taken by the best sequential algorithm and the sequential times taken for the p-subdomain preconditioned GMRES algorithm for $p = 2, 4, 8, 16$ and 32. The parallel times are then given, followed by the speedups achieved. The following two rows show the parallel speedup and the optimal speedup respectively (as defined in the subsection above).

	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$
Sequential time	234.4	351.0	301.2	271.0	262.6	238.2
Parallel time	—	191.0	83.1	42.9	20.6	10.7
Speedup	—	1.2	2.8	5.5	11.4	21.9
Parallel speedup	—	1.8	3.6	6.3	12.7	22.3
Optimal speedup	—	1.4	2.8	5.7	11.4	21.9

Table 4: Solution times (in seconds) and speedups for the proposed algorithm on the uniform mesh refinement example.

4.3 An example with local mesh refinement

For our second assessment of parallel performance we consider a fourth PDE of the form (2.1) which has been selected because the accurate and efficient finite element solution requires the use of local mesh refinement. This PDE takes the same form as Problem 1 from Subsection 3.1 however $f(\underline{x})$ is now chosen such that the exact solution is given by

$$u = (1 - (2x_1 - 1)^{100})(1 - (2x_2 - 1)^{100}) \quad \forall \underline{x} \in \Omega = (0, 1) \times (0, 1) \ . \tag{4.1}$$

This has a value of 1 in the interior of Ω but tends to 0 very rapidly in a thin layer near to the boundary: allowing the Dirichlet condition $u = 0$ to be satisfied throughout $\partial\Omega$.

Structured initial grid

Again a coarse mesh of 256 congruent triangular elements was used when solving this problem on $2, 4, 8, 16$ and 32 processors. The final mesh for this problem is obtained via local refinement (up to eight levels) based upon the interpolation error in the known solution and contains 760628 elements (most of which are situated in the boundary layer). Figure 5 illustrates this mesh (with a maximum of just three levels of refinement). The subdomains are created by partitioning the coarse mesh in such a way that the final number of elements in each subdomain is similar, which means that there are differing numbers of coarse elements in each subdomain for the cases $p = 8, 16$ or 32. The partitions into 2 and 4 subdomains are easily achieved using the symmetries of the problem and the mesh: again see Figure 5.

Table 5 shows the results of the same set of timings that were used for the previous test problem. This includes both sequential and parallel solution times as well as a range of speedup metrics for $p = 2, 4, 8, 16$ and 32.

Unstructured initial grid

We now compute solutions to the same problem based upon the local refinement of the unstructured initial grid of 560 elements illustrated in Figure 6. Once more a maximum of eight levels of refinement are permitted, yielding a final mesh containing 828430 elements. This is also partitioned so as to

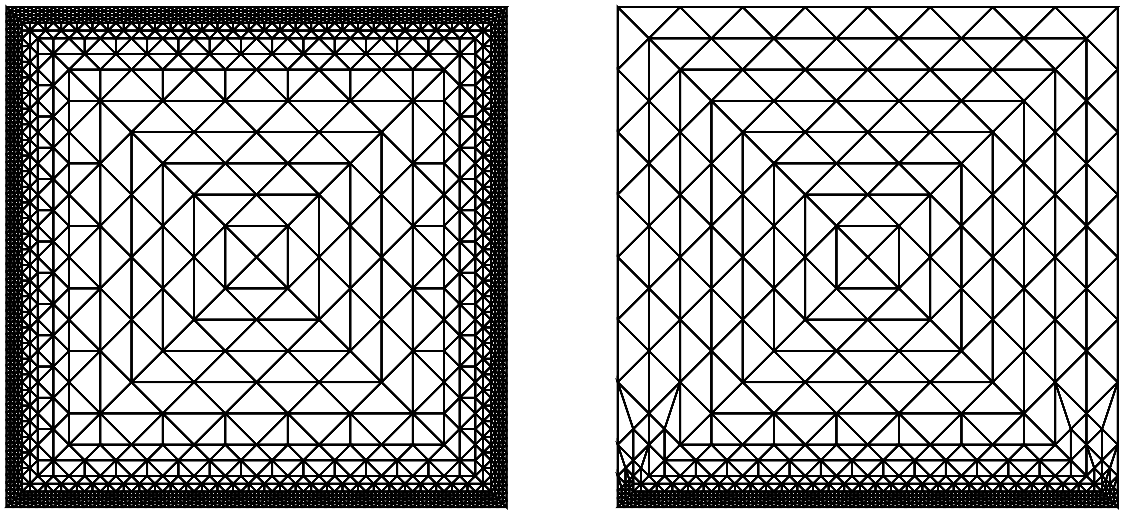

Figure 5: The final fine mesh using at most three level of refinement (left) and the corresponding mesh on one processor when $p = 4$ (right) for the first local mesh refinement example with a 256 element structured coarse mesh.

	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$
Sequential time	73.4	99.5	92.8	84.7	88.9	91.1
Parallel time	—	54.1	25.6	12.0	6.8	5.3
Speedup	—	1.4	2.9	6.1	10.8	13.8
Parallel speedup	—	1.8	3.6	7.1	13.1	17.2
Optimal speedup	—	1.5	3.1	6.1	10.8	13.8

Table 5: Solution times (in seconds) and speedups for the proposed algorithm on the first local mesh refinement example.

ensure that the final number of elements in each subdomain is similar: although this is a harder problem than for the structured initial grid (see Figure 6 for an example when $p = 4$). Table 6 shows the corresponding timings and speedups for this set of computations.

4.4 Discussion

The parallel results presented in this section are representative figures which provide evidence that the proposed algorithm provides a simple and practical means of obtaining good speedup ratios on a moderate number of parallel processors. Tables 4 to 6 clearly demonstrate that the sequential version of this domain decomposition preconditioner is competitive with our best available sequential solver and also that the parallel implementation can scale well to provide a useful parallel solver.

Closer inspection of this parallel solver reveals that the major contribution to the loss of efficiency that does occur is through inexact load-balancing in the preconditioning step. This can arise for two main reasons. The most obvious cause is any lack of equality in the size of the fine mesh on each subdomain. Clearly if one subdomain has a mesh with many more elements than the others they will spend a significant amount of idle time waiting for that processor at each synchronization point (e.g. each inner product in the GMRES algorithm). Such a situation arises when partitioning the

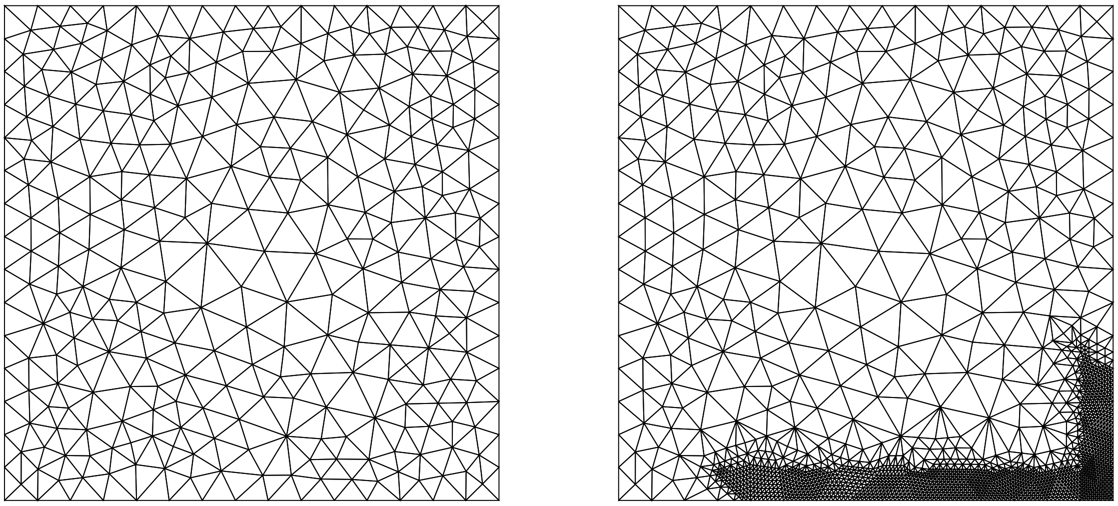

Figure 6: The initial 560 element unstructured mesh (left) and the resulting final mesh on one processor (with at most three levels of refinement) when $p = 4$ (right).

	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$
Sequential time	103.3	128.6	120.4	122.3	122.0	124.9
Parallel time	—	69.0	32.8	17.4	11.3	8.0
Speedup	—	1.5	3.1	5.9	9.1	13.0
Parallel speedup	—	1.9	3.7	7.0	10.8	15.6
Optimal speedup	—	1.6	3.1	5.9	9.1	13.0

Table 6: Solution times (in seconds) and speedups for the proposed algorithm on the second local mesh refinement example.

coarse mesh for the two non-uniform local refinement examples described in the previous subsection. Table 7 illustrates this by showing the maximum, minimum and average number of elements in the meshes created by each processor in the cases where $p = 2, 4, 8, 16$ and 32. Note that the cases where there is a large relative difference between the average and the maximum number of elements on each processor (over 10% for example) clearly correspond to those calculations in Tables 5 and 6 which yield the poorest parallel efficiencies (i.e. parallel speedup divided by p).

The second reason for inexact load-balancing in the preconditioner stems from the fact that the systems being solved on each processor (i.e. the systems in step 4.1 of Figure 3) are different, even when they are of the same size. These differences can lead to significant variations in the time required to solve the systems when a sparse iterative (or direct) solver is used. For example, the iterative solver that we use requires a multilevel ILU decomposition to be computed and, for a given drop tolerance, the size of this decomposition depends not only on the size and sparsity of the original system, but also on the actual values of the non-zero entries of the sparse matrix. This makes guaranteeing a well load-balanced preconditioner (especially on an unstructured or locally refined grid) a very challenging task.

The final point that we mention in this section concerns a further cause of inefficiency in the parallel solver. This relates to the differences between the best sequential solution times and the sequential solution times when using the p-subdomain preconditioner. When these differences are

	256 element structured coarse grid				
	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$
Maximum	381918	191826	98422	50344	28329
Minimum	381918	191826	94837	46789	22692
Average	381918	191826	96638.5	49031.75	25461.375
	560 element unstructured coarse grid				
	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$
Maximum	422329	213948	111531	58043	29786
Minimum	406101	202107	94964	44675	19830
Average	414215	207107.5	103553.75	51776.875	25888.4375

Table 7: The maximum, minimum and average number of elements in the meshes created when solving the two local mesh refinement examples from Subsection 4.3.

small good speedups are attainable but when they are large the efficiency of the parallel solver is inevitably diminished. As has already been mentioned in Subsection 3.2 a major issue for domain decomposition solvers such as the one considered here is subdomain shape. This issue has been addressed by a number of authors (e.g. [19, 26]) and it is generally accepted that for isotropic problems subdomains with good (i.e. small) aspect ratios are to be preferred. It is unlikely that this will always be the case however, especially for highly directional problems (e.g. convection dominated) and so further research needs to be undertaken in this area to understand the issues more fully.

5 Conclusions

In this paper we have presented a new parallel domain decomposition implementation that allows for efficient mesh refinement without a significant degradation in convergence rates as either the mesh size is decreased or the number of processors is increased. The motivation for this technique comes from the work of [5] and [30, 31] and requires a number of assumptions to be satisfied in order to achieve its full potential. In particular, the approach of [5] provides a much simplified load-balancing strategy based upon equidistributing an initial coarse-grid error however it may not always be guaranteed that, for complex problems requiring significant local refinement, an ideal load balance will be achieved on the final mesh. This will in turn have some detrimental effects on the parallel performance of the proposed solver, as is observed for some of the computations using local mesh refinement described in Subsection 4.3. For example, when sixteen processors are used with a structured initial grid the largest mesh contains only 2.6% more elements than the average and a parallel speedup of 13.1 is achieved. On the other hand, for the unstructured initial grid partitioned into sixteen subdomains the largest mesh contains over 12.1% more elements than the average and the parallel speedup is reduced to just 10.8.

In situations where the load balance on the final mesh becomes too poor there will clearly be some benefit from introducing an additional communication phase to the solution process in which the problem is repartitioned using a dynamic load-balancing algorithm (e.g. [25]). The assessment of precisely how poor the load balance must become before this repartitioning step becomes cost-effective is still an open question however. Further local communication between processors will also be required if the final fine mesh obtaineded by combining the refined meshes produced on each processor turns out to be non-conforming. In this work however, because of the additional layer of local refinement that is undertaken on each processor immediately beyond the boundary of its own subdomain, this situation did not arise in any of the examples described.

Finally, we note that there are a number of possible extensions of the work considered here. These include implementation of the technique in three dimensions and its application to nonlinear problems, both of which are currently being undertaken with encouraging initial results. A more ambitious extension is to use the approach described to parallelize arbitrary sequential codes without actually altering those codes in any way: they would merely be used as sequential solvers (on each processor) for the systems that correspond to that shown at step 4.1 of Figure 3.

Acknowledgements

The work of REB was supported by the National Science Foundation under contract DMS-9706090 and the work of PKJ, whilst visiting UCSD, was supported by a Research Grant from the Leverhulme Trust.

References

[1] M. Ainsworth, *A Preconditioner Based on Domain Decomposition for h-p Finite Element Approximation on Quasi-Uniform Meshes*, SIAM J. Numer. Anal., 33, 1358–1376, 1996.

[2] M. Ainsworth, *A Hierarchical Domain Decomposition Preconditioner for h-p Finite Element Approximation on Locally Refined Meshes*, SIAM J. on Sci. Comp., 17, 1395–1413, 1996.

[3] S.F. Ashby, T.A. Manteuffel and P.E. Taylor, *A Taxonomy for Conjugate Gradient Methods*, SIAM J. Numer. Anal., 27, 1542–1568, 1990.

[4] R.E. Bank, *PLTMG Users' Guide 8.0*, Society for Industrial and Applied Mathematics, 1998.

[5] R.E. Bank and M. Holst, *A New Paradigm for Parallel Adaptive Meshing Algorithms*, to appear in SIAM J. on Sci. Comp, 1999.

[6] R.E. Bank, P.K. Jimack and S.V. Nepomnyaschikh, *A Weakly Overlapping Domain Decomposition for the Adaptive Finite Element Solution of Elliptic Partial Differential Equations*, School of Computing Research Report 99.17, University of Leeds, 1999.

[7] R.E. Bank and R.K. Smith, *General Sparse Elimination Requires no Permanent Integer Storage*, SIAM J. Sci. Stat. Comp, 8, 574–585, 1987.

[8] R.E. Bank and R.K. Smith, *The Incomplete Factorization Multigraph Algorithm*, SIAM J. on Sci. Comp., 20, 1349–1364, 1999.

[9] R.E. Bank and C. Wagner, *Multilevel ILU Decomposition*, Numerische Mathematik, 82, 543–576, 1999.

[10] R. Biswas and R.C. Strawn, *A New Procedure for Dynamic Adaption of Three-Dimensional Unstructured Grids*, Appl. Numer. Math., 13, 437–452, 1994.

[11] J. Bramble, J. Pasciak and A.H. Schatz, *The Construction of Preconditioners for Elliptic Problems by Substructuring, I*, Mathematics of Computation, 47, 103–134, 1986.

[12] J. Bramble, J. Pasciak and A.H. Schatz, *The Construction of Preconditioners for Elliptic Problems by Substructuring, II*, Mathematics of Computation, 49, 1–16, 1987.

[13] J. Bramble, J. Pasciak and A.H. Schatz, *The Construction of Preconditioners for Elliptic Problems by Substructuring, III*, Mathematics of Computation, 51, 415–430, 1988.

[14] J. Bramble, J. Pasciak and A.H. Schatz, *The Construction of Preconditioners for Elliptic Problems by Substructuring, IV*, Mathematics of Computation, 53, 1–24, 1989.

[15] J. Bramble, J. Pasciak and J. Xu, *Parallel Multilevel Preconditioners*, Mathematics of Computation, 55, 1–21, 1990.

[16] P.J. Capon and P.K. Jimack, *An Adaptive Finite Element Method for the Compressible Navier-Stokes Equations*, in Numerical Methods for Fluid Dynamics 5 (M.J. Baines and K.W. Morton, eds.), OUP, 1995.

[17] M. Dryja, *An Additive Schwartz Algorithm for Two- and Three-Dimensional Finite Element Elliptic Problems*, in Second International Symposium on Domain Decomposition Methods (T. Chan *et al*, eds.), Society for Industrial and Applied Mathematics, 1989.

[18] M. Dryja and O.B. Widlund, *Some Domain Decomposition Algorithms for Elliptic Problems*, in Iterative Methods for Large Linear Systems, Academic Press, 1990.

[19] C. Farhat, M. Maman and G.W. Brown, *Mesh Partitioning for Implicit Computations via Iterative Domain Decomposition: Impact and Optimization of Subdomain Aspect Ratio*, Int. J. for Numer. Meth. in Eng., 28, 989–1000, 1995.

[20] C. Farhat, J. Mandel and F.X. Roux, *Optimal Convergence Properties of the FETI Domain Decomposition Method*, Computer Methods for Applied Mechanics and Engineering, 115, 365–385, 1994.

[21] G.H. Golub and C.F. Van Loan, *Matrix Computations*, John Hopkins Press, 3rd edition, 1996.

[22] M. Griebel and P. Oswald, *On Additive Schwartz Preconditioners for Sparse Grid Discretizations*, Numerische Mathematik, 66, 449–463, 1994.

[23] B. Heise and M. Jung, *Parallel Solvers for Nonlinear Elliptic Problems Based Upon Domain Decomposition Ideas*, Parallel Computing, 22, 1527–1544, 1997.

[24] D.C. Hodgson and P.K. Jimack, *A Domain Decomposition Preconditioner for a Parallel Finite Element Solver on Distributed Unstructured Grids*, Parallel Computing, 23, 1157–1181, 1997.

[25] G. Karypis and V. Kumar, *Parallel Multilevel k-Way Partitioning Scheme for Irregular Graphs*, SIAM Review, 41, 278–300, 1999.

[26] D.E. Keyes, D.K. Kaushik and B.F. Smith, *Prospects for CFD on Petaflops Systems*, in CFD Review 1998 (M. Hafez and K. Oshima, eds.), World Scientific, 1998.

[27] C.-H. Lai, P.E. Bjorstad, M. Cross and O.B. Widlund (eds.), *The Eleventh International Conference on Domain Decomposition Methods: Domain Decomposition Methods in Sciences and Engineering*, Domain Decomposition Press, 1999.

[28] R. Lohner, *An Adaptive Finite Element Scheme for Transient Problems in CFD*, Comp. Meth. in Appl. Mech. and Eng., 61, 323–338, 1987.

[29] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard*, International Journal of Supercomputer Applications, 8, No. 3/4, 1994.

[30] W.F. Mitchell, *The Full Domain Partition Approach to Distributing Adaptive Grids* Appl. Numer. Math., 26, 265–275, 1998.

[31] W.F. Mitchell, *A Parallel Multigrid Method Using the Full Domain Partition*, Electronic Transactions on Numerical Analysis, 6, 224–233, 1998.

[32] Y. Saad and M. Schultz, *GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems*, SIAM J. on Scientific Computing, 7, 856–869, 1986.

[33] B.F. Smith, *A Parallel Implementation of an Iterative Substructuring Algorithm for Problems in Three Dimensions*, SIAM J. on Scientific Computing, 14, 406–423, 1993.

[34] W. Speares and M. Berzins, *A 3-D Unstructured Mesh Adaptation Algorithm for Time-Dependent Shock Dominated Problems*, Int. J. for Numer. Meth. in Fluids, 25, 81–104, 1997.

[35] O.B. Widlund, *Some Schwartz Methods for Symmetric and Nonsymmetric Elliptic Problems*, in Fifth International Symposium on Domain Decomposition Methods (D.E. Keyes *et al*, eds.), Society for Industrial and Applied Mathematics, 1992.

[36] J. Xu, *Iterative Methods by Space Decomposition and Subspace Correction*, SIAM Review, 34, 581–613, 1992.

[37] X. Zhang, *Multilevel Schwartz Methods*, Numerische Mathematik, 63, 521–539, 1992.