Automatic domain decomposition on unstructured grids (DOUG)

M.J. Hagger*

Abstract

This paper describes a parallel iterative solver for finite element discretisations of elliptic partial differential equations on 2D and 3D domains using unstructured grids. The discretisation of the PDE is assumed to be given in the form of element stiffness matrices and the solver is automatic in the sense that it requires minimal additional information about the PDE and the geometry of the domain. The solver parallelises matrix-vector operations required by iterative methods and provides parallel additive Schwarz preconditioners. Parallelisation is implemented through MPI. The paper contains numerical experiments showing almost optimal speedup on unstructured mesh problems on a range of four platforms and in addition gives illustrations of the use of the package to investigate several questions of current interest in the analysis of Schwarz methods. The package is available in public domain from the home page `http://www.maths.bath.ac.uk/~mjh/`

Short Title: Automatic Domain Decomposition

Keywords: Partial Differential Equations, Finite Elements, Unstructured Grids, Parallel Solver, Domain Decomposition, Additive Schwarz, MPI, General Purpose Software

AMS(MOS) Classification: 65F10, 65N22, 65N55

*(M.J.Hagger@maths.bath.ac.uk) DERA Winfrith, Winfrith Technology Centre, Winfrith Newburgh, Dorchester DT2 8XJ, United Kingdom. This authors work was part of the UK Engineering and Physical Sciences Research Council Project GR/J88616 on Parallel Methods for Elliptic PDEs.

1 Introduction

In this paper we describe and analyse a general purpose portable package for the parallel iterative solution of linear systems arising from finite element discretisations of elliptic equations in two and three dimensional domains. Our code can be used as a vehicle for the parallel solution of linear systems arising from any scalar linear elliptic equation discretised by finite elements, or for linearisations of nonlinear problems: For the parallelisation of the basic iterative method the user must provide:
Requirement (i) The discretisation in the form of element stiffness matrices and maps between local and global freedoms, such as is provided by a standard finite element package.
Requirement (ii) A choice of iterative method, usually made on the basis of a priori knowledge of the problem.
Requirement (iii) A parallel environment, plus any parallelisation choices necessary for this environment, for example the number of processors to be used.

Our package will automatically subdivide the problem into an appropriate number of subproblems and use parallelised versions of matrix-vector and dot-product operations to implement the iterative method.

Most iterative methods are not robust - or optimal in the sense of scaling linearly with problem size - without preconditioning. Our package provides preconditioners of additive Schwarz type (see [6]) with two levels of sophistication. The first and basic level is essentially a block Jacobi preconditioner constructed from local solves on subdomains with small overlap (including, as a special case, classical diagonal scaling). This basic preconditioner is constructed from the algebraic information in (i) above and thus is quite problem and geometry independent. However, it is well-known that as the number of subdomains increases, the performance of such preconditioners deteriorates and a coarse grid problem must be added to redeem the performance. This is also provided by our package[1] but for this the user must provide more data:
Requirement (iv) The location in space of the points (nodes) at which the unknowns (freedoms) in the problem are to be computed.

The coarse grid is automatically generated by an adaptive technique which seeks to balance the conflicting aims of

(a) Adequate representation of the partial differential equation, and

(b) Complexity which does not perturb the overall parallel performance of the algorithm.

Our overall aim has been to provide an efficient parallel solver which requires minimal input from the user and could then be used in conjunction with a suitable discretisation package. In fact we have already used our solver in conjunction with a commercial finite element code - ENTWIFE marketed by AEA Technology, Harwell, UK.

Our solver is based on the additive Schwarz method which has attracted a lot of attention in the Numerical Analysis literature: see, e.g. [6], [27]. More recently the analysis of this method has been extended to the unstructured case - [9], [11], [5], [10]. The chief novelty of the work described in this paper is that it is (to our knowledge) the first general purpose parallel implementation of this method (although application-specific codes have been widely reported).

---

[1] In the first release of the package the coarse grid problem is only provided in 2D. This is not a limitation of the approach, merely that the coding is not complete for the 3D version. The extension to 3D is the subject of on-going work at the University of Bath.

The main purposes of this paper are to describe the parallel implementation, to highlight the precautions used to maintain parallel efficiency and to demonstrate this efficiency on several hardware platforms.

After two short preliminary sections, Sections 4 – 6 describe in detail the techniques used to preserve parallel efficiency in the implementation of iterative methods through mesh partitioning. In Section 7 we pay particular attention to the construction and implementation of a novel fast coarsening technique based on locally uniform mesh refinement. In Section 8 we give results which show that the resulting algorithm exhibits parallel efficiency across four quite different architectures . Finally in Section 9 we illustrate the use of the code in the empirical investigation of several open questions on the numerical analysis of Schwarz methods.

The algorithm described here is available as the DOUG package. A user guide and information on how to get the package can be found at
`http://www.maths.bath.ac.uk/~mjh/doug` .

2 Parallel implementation

Our code is designed to run on a wide range of computing environments - anything from a single workstation, or an ethernet connected network of workstations, up to a dedicated multiprocessor machine such as an SP/2 or Cray T3D. By basing our code on the MPI (Message Passing Interface) system and assuming nothing about the computational environment, we automatically allow for any "parallel" setup, including just a single processor [2].

We note that MPI is a standard of communication that is being actively supported by a number of computer vendors, including IBM, Silicon Graphics and Cray. In addition a freely available version, MPICH, which runs on a variety of computer architectures is available from ftp://ftp.mcs.anl.gov/pub/mpi.

Unfortunately, as with almost all parallel codes, the performance of any package may be heavily dependent on how fast processors can communicate with each other. In the case of ethernet connected workstations this can be very slow indeed. For machines with faster communication this cost is greatly reduced. Nevertheless we report good parallel performance of our particular algorithm across the entire range of machines tested.

It is important to realise that communication between processors is generally a very slow operation compared to computation speeds. Effective parallel programs must therefore take this into account in their design.

2.1 Blocking vs. non-blocking

In general parallel communication is divided into two types: *Blocking* and *non-blocking* messages. In the case of a blocking message, program flow halts until the message has been sent (or received). With non-blocking messages the program flow continues, with the message being sent (or received) as soon as possible. That is the sending and receiving happens in the *background*. The primary advantage of using non-blocking messages is that whilst the communication is taking place the processor can continue doing useful work. This can make a crucial difference to the overall performance of a parallel code. Where at all possible DOUG uses non-blocking communication.

[2] This first release does not implement the single processor case, as the multiprocessor case has been deemed more interesting in the first instance.

Another crucial aspect of keeping overall communication overheads as small as possible is the use of non-blocking sends in situations where one processor must communicate with a number of others. By starting a number of sends and receives at the same time we impose no restriction on the order that they occur, and therefore greatly reduce any potential bottlenecks caused by one slave lagging behind the others. This approach also means that a *dead-lock* situation, where a number of processors effectively hang-up as they all wait for the other to send or receive a message, cannot happen.

2.2 Communication time - latency costs

For a given parallel machine there are two factors which control the time that a parallel communication takes. The first is a message startup time, sometimes called the communication *latency*. This is generally a fixed amount of time, irrespective of the length of the message. Different architectures have quite widely different latencies. In particular, the latency is often quite large for workstation clusters, which are not designed as dedicated parallel machines. Obviously to avoid wasting too much time in message startups the number of them should be kept as low as possible. This is best achieved by, where possible, grouping a number of sends together, so the message can be sent in one large continuous block. The second time factor is simply how fast data can be sent along whatever communication route is available. This is highly machine dependent, and nothing can be done by the user of any given machine to control it.

3 Notation

From now on we assume that we are solving a scalar elliptic PDE on a bounded 2D or 3D domain Ω using a a finite element method specified as in **Requirement** (i) of the Introduction.

To clarify the discussion on the implementation we will use the following notation. We let $\mathcal{E}$ denote the set of elements - e.g. triangles, quadrilaterals, tetrahedra or bricks. The mesh determines a set of nodes $\mathcal{N}$, with a typical node denoted $p \in \mathcal{N}$. Associated with these nodes is a set of freedoms $\mathcal{F} \subset \mathcal{N}$. A node will not necessarily be a freedom, for example if an essential boundary condition is imposed there. For simplicity only we do not consider Hermite type elements and so each node is assumed to have no more than one freedom. N denotes the number of nodes. Each element, $E \in \mathcal{E}$ has an associated element stiffness matrix, A_E, which is computed from the weak form of the partial differential equation (PDE) by restricting the domain to simply the element, in the usual way for finite element methods. Each element has an associated set of freedoms that lie on it, which is denoted by $\mathcal{F}_E$.

For convenience we will consider the element stiffness matrix, A_E, to be of size $N \times N$, although in practice it is zero everywhere except for the rows and columns corresponding to freedoms that lie on E, and would of course be stored accordingly.

The global stiffness matrix, A, is then given by

$$A = \sum_{E \in \mathcal{E}} A_E. \qquad (3.1)$$

In our parallel implementation the set of elements is partitioned into subsets, denoted $\mathcal{E}_i$, where $i = 1, ..., N_p$, and N_p is, in this case, the number of available processors. Associated with these subsets of elements are sets of freedoms $\mathcal{F}_i$. A single freedom lies in more than one $\mathcal{F}_i$ when it lies on elements which appear in at least two different $\mathcal{E}_i$.

For each i the contribution to the global stiffness matrix is denoted A_i, and is given by

$$A_i = \sum_{E \in \mathcal{E}_i} A_E.$$

A_i could also be thought of as the element stiffness matrix corresponding to the "macro element" $\cup\{E : E \in \mathcal{E}_i\}$. Hence, since the $\mathcal{E}_i$ partition $\mathcal{E}$, the global stiffness matrix is also given by

$$A = \sum_{i=1}^{N_p} A_i \ . \tag{3.2}$$

For each $\mathcal{E}_i$ we introduce a corresponding restriction of A, denoted $\tilde{A}_i$, defined by

$$\left(\tilde{A}_i\right)_{pq} = \sum_{E \in \mathcal{E}} (A_E)_{pq}, \text{ for } p, q \in \mathcal{F}_i. \tag{3.3}$$

and (3.4)

$$\left(\tilde{A}_i\right)_{pq} = 0, \text{ elsewhere.}$$

Remarks

1) This definition of $\tilde{A}_i$ gives an $N \times N$ matrix which coincides with A in the rows and columns corresponding to those freedoms that lie in $\mathcal{F}_i$, but has zeros elsehwere.

2) In practice we consider $\tilde{A}_i$ as an $N_i \times N_i$ matrix, via an appropriate local numbering of the freedoms in $\mathcal{F}_i$. With this interpretation it is useful to write

$$\tilde{A}_i = R_i A R_i^T$$

where R_i is the rectangular matrix representing the pointwise restriction of a vector defined on $\mathcal{F}$ to a vector defined on $\mathcal{F}_i$ and R_i^T is its transpose. Where we later use, or refer to, $\tilde{A}_i$ in the same context as R_i we will mean the smaller re-numbered version as in this definition.

3) It is important to distinguish between the matrices A_i and $\tilde{A}_i$. If, for example, the domain $\Omega_i := \cup\{E : E \in \mathcal{E}_i\}$ does not touch $\partial\Omega$ then A_i is the stiffness matrix for the PDE with *natural* boundary conditions on $\partial\Omega_i$. On the other hand $\tilde{A}_i$ is the stiffness matrix for the same PDE on the domain $\tilde{\Omega}_i = \cup\{E : \mathcal{F}_E \cap \mathcal{F}_i \neq \emptyset\}$ with *essential* boundary conditions on $\partial\tilde{\Omega}_i$. $\tilde{\Omega}_i$ is simply the extension of Ω_i using a single layer of elements in all directions, except where Ω_i touches $\partial\Omega$.

4) We may also wish to consider further extentions of $\tilde{A}_i$. For example we could define $\mathcal{F}_{i,1}$ as

$$\mathcal{F}_{i,1} = \cup \left\{\mathcal{F}_E : E \in \mathcal{E}, \mathcal{F}_E \cap \mathcal{F}_i \neq \emptyset\right\}.$$

The extended matrix $\tilde{A}_{i,1}$ is then given by replacing $\mathcal{F}_i$ with $\mathcal{F}_{i,1}$ in equation (3.4). This extension method can be applied recursively, to generate $\mathcal{F}_{i,l}$, $l = 2, 3..$

Associated with the set of freedoms $\mathcal{F}$ are vectors of size N, with an entry for each freedom. If $\mathbf{x}$ is such a vector, then there are also corresponding vectors, $\mathbf{x}_i$, associated with each $\mathcal{F}_i$ defined by

$$\begin{aligned} (\mathbf{x}_i)_p &= (\mathbf{x})_p, \text{ if } p \in \mathcal{F}_i \\ &= 0, \text{ otherwise,} \end{aligned}$$

i.e. $\mathbf{x}_i = R_i\mathbf{x}$ using the definition of R_i above.

Since a single freedom may lie on more than one $\mathcal{F}_i$ then the value $(\mathbf{x})_p$ may appear in more than one $\mathbf{x}_i$, and so $\sum_i \mathbf{x}_i \neq \mathbf{x}$. Obviously a significant portion of each vector $\mathbf{x}_i$ is zero, and it is thus stored in an appropriate fashion.

We may also allow a further sub-partitioning of each $\mathcal{E}_i$ into N_{pi} sub-partitions, for some integers N_{pi}. These new sub-partitions are denoted $\mathcal{E}_{ij}$, $j = 1, .., N_{pi}$, $i = 1, .., N_p$. As before each of these subsets of elements have associated set of freedoms, denoted $\mathcal{F}_{ij}$. The reason for this subpartitioning is to allow extra flexibility in the parallel implementation: The number of processors N_p need not coincide with the number of subdomains - see §5.2. Clearly all the previous definitions could be extended to these subpartitions, but since this is quite obvious we shall not contruct the necessary notation explicitly.

In the same way as before, a restricted problem, $\tilde{A}_{ij}$, is defined by

$$\left(\tilde{A}_{ij}\right)_{pq} = \sum_{E\in\mathcal{E}} (A_E)_{pq}\,, \text{ for } p, q \in \mathcal{F}_{ij}.$$

and

$$\left(\tilde{A}_{ij}\right)_{pq} = 0, \text{ elsewhere.}$$

The same remarks as above also apply in this case.

4 Preconditioned iterative methods

The class of iterative methods that we have in mind for our package require the following types of operations:

(i) Vector-update, i.e. $\mathbf{z} = \mathbf{x} + \mathbf{y}$.

(ii) Matrix-vector multiply, i.e. $\mathbf{y} = A\mathbf{x}$.

(iii) Dot products, i.e. $\mathbf{x}.\mathbf{y}$,

(iv) Solution of systems , i.e. $M\mathbf{z} = \mathbf{r}$ for some preconditioner M.

For a symmetric system this allows such iterative schemes as Jacobi's method and conjugate gradients (CG), see [16] for details on these methods. For non-symmetric systems this would mean methods such as Bi-CG, Bi-CGSTAB and CGS, see, for example, [28]. Although not strictly requiring just the above operations, GMRES[25] could also be implemented.

In the initial release of the package we have implemented the CG and Bi-CGSTAB algorithms. These allow solution of symmetric and non-symmetric linear systems respectively and these routines may also serve as a template for using other iterative schemes.

For the preconditioners we consider the additive Schwarz domain decomposition preconditioner, as discussed in [6], which is constructed as follows: Referring to the technology introduced in §3 we have subdivided Ω into N_p non-overlapping subdomains, $\Omega_i = \cup\{E : E \in \mathcal{E}_i\}$. There may be more of these if further subdivisions of the $\mathcal{E}_i$ had been employed. There are also corresponding extended subdomains with one-element overlap $\tilde{\Omega}_i = \cup\{E : \mathcal{F}_E \cap \mathcal{F}_i \neq \emptyset\}$. These could be extended to greater overlap in an obvious way. We further consider a coarse mesh problem, with mesh size H. Let R_H^T denote linear interpolation from the coarse to the fine mesh and let A_H denote an approximation to A on the coarse mesh. (This will be discussed in more detail in §7 .)

Then the (two-level) additive Schwarz preconditioner, M_{ASC}^{-1}, is defined by

$$M_{\mathrm{ASC}}^{-1} = R_H^T A_H^{-1} R_H + \sum_{i=1}^{p} R_i^T \tilde{A}_i^{-1} R_i. \tag{4.5}$$

We also may leave out the coarse grid solve to obtain

$$M_{\mathrm{AS}}^{-1} = \sum_{i=1}^{p} R_i^T \tilde{A}_i^{-1} R_i. \tag{4.6}$$

Details of the specific implementation of these operations in a parallel setting will be given in the following sections.

5 Subdividing the problem

In this section we describe the computation of the partition of the elements $\mathcal{E}$ into subsets $\mathcal{E}_i$ as discussed in §3. This partition has two purposes, but these to some extent conflict with each other. The first aim is to provide an effective parallelisation of the basic linear algebra operations needed in iterative methods (items (i)-(iii) stated at the beginning of §4), and the second aim realtes to the performance of the additive Schwarz preconditioner (4.5).

From the point of view of the first aim, the primary goal of the partition should be to break the domain into pieces such that the quantity of communication between them is minimised. This partition induces a corresponding partition of the stiffness matrix A as in (3.2) and minimises the communication costs per matrix-vector multiply step. This is highly desirable from a parallel computing point of view.

With respect to the second aim, however, theoretically a more effective preconditioner is produced if these subdomain problems are strongly connected to neighbouring subdomains, (i.e. share many freedoms), this concept being usually referred to as *overlap*. However, it is obvious that strongly connected subdomains mean a high degree of communication when computing the action of the preconditioner, which from a parallel computation point of view is not desirable. At first glance it is not perhaps immediately obvious how it is possible to construct these subdomain problems without referring back to the domain the matrices arise from. However, as we will see later, the subdivision used for the matrix-vector multiply can be used as a first stage in constructing the subdomains required for the preconditioner.

For the remainder of the discussion an effective subdivision is then considered as one that results in a minimal amount of parallel communication during a matrix-vector multiply operation.

5.1 Graph partitioning

In this subsection we describe how to partition $\mathcal{E}$ in to the subsets $\mathcal{E}_i$. Partitioning by elements, rather than freedoms, has two main advantages: The finite element code produces a data structure in an element form, which we wish to keep. If we then partition by freedoms in some fashion, then it becomes necessary to decide how to allocate elements whose freedoms lie on boundaries of the partition. Secondly if we require that the partitions are all the same size then we know that each of the processors will be given the same amount of work, as the matrices A_i will then be (nearly) all the same size.

The partitioning needs to be carried out in such a way as to minimise any resulting communication costs for the matrix-vector operation. This is equivalent to choosing a partition such that $\mathcal{F}_p \cap \mathcal{F}_q$ is as small as possible for $p, q = 1, .., N_p$, $p \neq q$.

The partitioning is performed in DOUG by producing a connected graph, with the element stiffness matrices as graph nodes, and graph edges where two elements share an edge (in 2D) or a face (in 3D). (In practice this is determined by counting the number of freedoms that two elements share.)

This graph is an easily stored sparse matrix and we can then apply highly efficient graph partitioning techniques to it to yield a partition of the element stiffness matrices. In our implementation we use a package called METIS[24], which uses a multilevel recursive bisection algorithm to subdivide the graph into an arbitrary number of partitions. These partitions satisfy two important criteria: Firstly they are load balanced, in the sense that a (nearly) equal number of elements (or graph nodes) appear in each partition. Secondly the number of *cut edges* is minimised in some sense, a cut edge being one where the two elements connected by it appear in two different partitions.

These partitions then induce subsets of element stiffness matrices, which, as described above, yield subdivisions of the stiffness matrix. In Figure 1 we illustrate a three dimensional mesh that has been partitioned into six, using this approach.

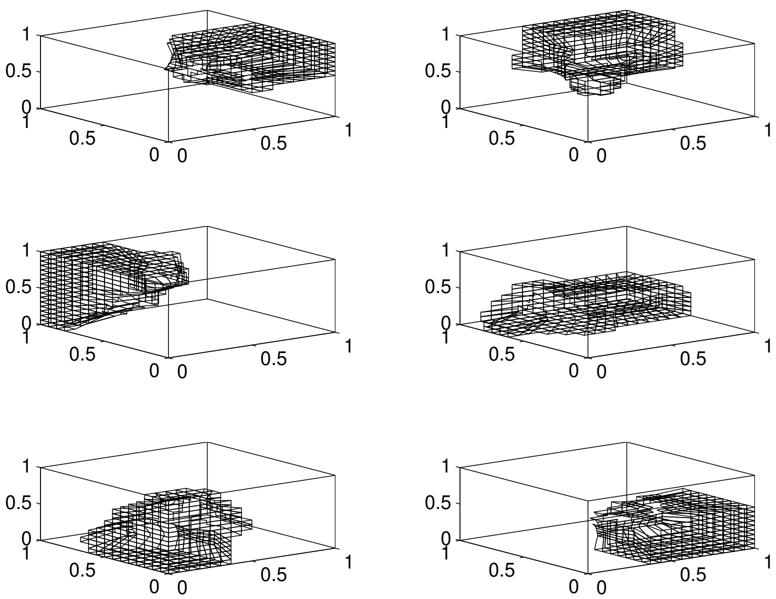

Figure 1: *Six subgraphs of a 3D mesh*

This initial partition of $\mathcal{E}$ allows us to efficiently implement matrix-vector multiplications, $A\mathbf{x}$, in parallel, as it produces good choices for the submatrices A_i. Moreover, the partitions $\mathcal{E}_i$ can be easily used as a base from which to construct the overlapping subdomains $\tilde{\Omega}_i$, in the manner described in §4.

Remarks

1) Although the theory for the additive Schwarz method indicates a dependence on the spatial size of the overlap, numerical evidence suggests that practically this bound is quite pessimistic, see for example [7, 26], and hence in DOUG we only allow a single node overlap approach at this time.

2) Practically the idea of minimising the connections between subdomains is attractive

from the parallel computing point of view, as it minimises the communication costs. However, it is likely that a good choice of subdomains from the preconditioning point of view is such that the connections are *maximised* in some sense. This is where the conflicting ideas regarding the choice of subdomains lie.

5.2 Further subdividing

One of the goals of the implementation is to allow the code to run on an arbitrary number of processors, from one upwards. Hence it is clear that producing subdomains by simply subdividing the problem amongst the available processors may not be enough. For example if we have only one or two processors then the subproblems may be too big to effectively solve using direct methods. To facilitate this we allow the processors to further partition their local problems, into the sub-partitions $\mathcal{E}_{ij}$, and construct the smaller subdomain matrices $\tilde{A}_{ij}$. The same partitioning technique as used before is also applied in this case.

The choice of the number of local sub-partitions, ie N_{pi}, is based on a desired optimal size of subproblem, which is taken as a number of degrees of freedom in the subproblem, the default value in DOUG being 1000.

In this way the effect of the number of processors will be essentially hidden from the user, DOUG will subdivide the problem amongst its available processors as it sees fit.

Remarks

1) It is likely if fast, modern parallel computing machines are used to solve very large problems the best value for the size of subproblem may be far larger than 1000.

2) For the rest of the discussion, to aid clarity, we will assume that no further subdivisions have been made, i.e. $N_{pi} = 1$. Note that since any further subdivisions are local to a processor this entails *no change at all* to any parallel communication requirements.

6 Implementation of the algorithm

In this section we discuss the implementation of the primary operations required for an iterative solver, showing what we believe to be a model implementation.

Each processor has stored the values for its freedoms $\mathcal{F}_i$ on the subdomain $\mathcal{E}_i$ in the vector $\mathbf{x}_i$. As already discussed these vectors overlap to a small extent. Certain operations, particularly those involving matrices, will require some parallel communication to maintain the consistency of these vectors, in the sense that shared entries are the same on all the slaves.

6.1 Master - Slave setup

The processor setup is such that one processor, called the master, is effectively in charge of the computations, and deals with the subdivision and distribution of the stiffness matrix data to the other processors, which are called slaves. The master starts the computation with an initial guess and then monitors the convergence to determine when to stop. If a coarse grid problem is being used then the master processor handles the solution of this also.

6.2 Vector updates

This is the simplest operation, and occurs frequently in iterative methods, it can be generally written as

$$\mathbf{y} = \alpha\mathbf{x} + \beta\mathbf{y}.$$

The parallel version requires no communication since all the data we need is local and any shared data has the same operation performed on it for all processors, it can thus be written as

$$\mathbf{y}_i = \alpha\mathbf{x}_i + \beta\mathbf{y}_i, \text{ for } i = 1, .., N_p.$$

6.3 Matrix-vector multiply

The general form of this operation is

$$\mathbf{y} = A\mathbf{x}.$$

We wish to compute $\mathbf{y}$ or equivalently its pointwise restrictions $\mathbf{y}_i$ for each i. A simple way to implement this is to first compute the *local* contributions

$$\hat{\mathbf{y}}_i = A_i\mathbf{x}_i, \tag{6.7}$$

which are each purely local loops over the elements $E \in \mathcal{E}_i$. The elements of $\mathbf{y}_i$ coincide with those of $\hat{\mathbf{y}}_i$ except at freedoms which belong to more than one $\mathcal{F}_i$. For such freedoms some communication is needed to assemble $\mathbf{y}_i$.

Although this approach will give the correct answer it is not the most efficient, as all the communication will occur at the end of the operation. However, it is clear that in general the quantity of shared data is greatly outweighed by the amount of local-only data, since the shared data lies only near the boundaries of the $\mathcal{E}_i$. Hence if we know which of the locally stored element stiffness matrices produce contributions to the shared data we can re-order the computations and produce a more efficient parallel version.

The elements that need to be processed to produce the data that must be sent to other processors is simply given by the set $\mathcal{S}_i$, defined by

$$\mathcal{S}_i = \{E : E \in \mathcal{E}_i, \mathcal{F}_E \cap \mathcal{F}_j \neq \emptyset, \text{ for some } j \neq i\}.$$

The efficient version of the matrix-vector operation is now given.

Algorithm 1 [Matvec] Compute matrix-vector multiply in parallel

1) *Compute all the matrix-vector entries corresponding to any freedoms in $\mathcal{F}_i$ which are shared with other $\mathcal{F}_j$, that is*

$$\tilde{\mathbf{y}}_i = \sum_{E \in \mathcal{S}_i} A_E\mathbf{x}_i. \tag{6.8}$$

2) *Start non-blocking communication sends to the slaves seeking the entries of $\tilde{\mathbf{y}}_i$ corresponding to shared freedoms. Also set up the required receives for the incoming data.*

3) Compute the rest of $\hat{\mathbf{y}}_i$*, i.e. continue summing right-hand side of (6.8) over elements* $E \in \mathcal{E}_i \backslash \mathcal{S}_i$*.*

4) Wait for the incoming data to arrive and process the data to produce $\mathbf{y}_i$ *from* $\hat{\mathbf{y}}_i$*.*

With this form of algorithm the communication overhead can be greatly reduced, since the communication in step 4 is overlapped with the local only computations in step 3. The principal idea is that step 4 of the algorithm would in practice require no waiting at all since the necessary communication would have occurred contemporaneously with the computation in step 3. In our experiments we find that this is generally the case.

In practice it is convenient to implement this updating of $\hat{\mathbf{y}}_i$ by having one subroutine which is called by each slave with essentially the argument $\hat{\mathbf{y}}_i$, and a second subroutine which returns the updated vector $\mathbf{y}_i$. Between these subroutine calls the shared data parts of the vector $\mathbf{y}_i$ may not be modified. Any other computation which does not affect these particular entries may be carried out.

6.4 Dot products

The dot product of the two vectors $\mathbf{x}$ and $\mathbf{y}$ is simply given by

$$\mathbf{x}.\mathbf{y} = \sum_p (\mathbf{x})_p (\mathbf{y})_p .$$

Since $\mathcal{F}_i \cap \mathcal{F}_j \neq \emptyset$ for some $i \neq j$, it is first convenient to introduce unique sets of freedoms $\bar{\mathcal{F}}_i$ defined by

$$\begin{aligned} \bar{\mathcal{F}}_1 &= \mathcal{F}_1 \\ \bar{\mathcal{F}}_i &= \mathcal{F}_i \backslash \{\bar{\mathcal{F}}_1 \cup ... \cup \bar{\mathcal{F}}_{i-1}\}, \quad i = 2, .., N_p. \end{aligned}$$

Then the dot product operation is given by

$$\mathbf{x}.\mathbf{y} = \sum_i \sum_{p \in \bar{\mathcal{F}}_i} (\mathbf{x}_i)_p \, (\mathbf{y}_i)_p \, .$$

In practice this means that each processor performs a sum over the freedoms in $\bar{\mathcal{F}}_i$ and these values must then be summed over all the processors and the result communicated to all the processors. In fact this final summation can be easily implemented via the MPI_ALLREDUCE subroutine with an MPI_SUM flag (a standard MPI command).

Unfortunately in this case it is not possible to overlap computation with communication, however, the quantity of data being communicated is relatively small, being only a single value from each processor.

Remark

Practically the dot product operation turns out to be somewhat of a bottleneck in the computation, since all the processors must synchronise and perform communication before computations can continue. There appears to be little that can be done to alleviate this problem, except ensuring that the number of dot products is kept as small as possible.

6.5 Preconditioner solve

The additive Schwarz preconditioner of §4 is considered in two forms: Firstly with no coarse solve and secondly with a coarse solve. For the moment we simply consider the former case, where the preconditioner is a sum of subdomain solves.

Globally the preconditioner requires an operation of the following form

$$\mathbf{z} = \sum_i \tilde{A}_i^{-1}\mathbf{x}.$$

The parallel version is then to compute $\hat{\mathbf{z}}_i$ as

$$\hat{\mathbf{z}}_i = \tilde{A}_i^{-1}\mathbf{x}_i,$$

then by parallel communication $\mathbf{z}_i$ is produced from $\hat{\mathbf{z}}_i$, in the same way as in §6.3.

The implementation of the preconditioner with a coarse grid solve will be discussed in §7.7.

In the case where there are further subdivisions (i.e. $N_{pi} \neq 1$), it may be possible to order the direct solves such that those on the boundary of a subdomain are performed first and communication can be carried out whilst the rest of the solves are performed. However, in general, this is likely to be of limited use since the number of further subdivisions is usually quite small and hence almost all of them touch the edges of the subdomain. For this reason this approach has not been implemented in DOUG, as yet.

7 Coarse mesh implementation

The use of an arbitrary coarse mesh in domain decomposition methods for the solution of elliptic problems on an unstructured mesh would appear to be a well founded idea. Optimal convergence can be proved up to an unknown constant. Chan, Smith and Zou discuss this approach in [11, 10], where the following general assumptions are made on the coarse mesh (for more precise details see [11, 10]:

(A1) The coarse mesh covers all of the Neumann boundary

(A2) No coarse mesh element lies completely outside the fine mesh

(A3) The coarse mesh covers a significant part of the fine mesh.

Remark. Assumption (A1) is perhaps slightly misleading, and is better interpreted as "The coarse mesh covers all the freedoms in the fine mesh problem", which is to say that only Dirichlet boundary points of the fine mesh can lie outside the coarse mesh.

For the standard additive Schwarz method, as already discussed, we require the fine and coarse meshes to be shape regular but not necessarily quasi-uniform. Then if the assumptions (A1)-(A3) are satisfied the estimate (9.14) holds. The proof of this result requires that the operator that maps the coarse space to the fine space satisfies certain H^1 and L^2 approximation properties, however, Chan and Zou are able to prove these for the (simple) natural finite element interpolation, under the assumption (A1). This is the form of interpolation which we shall use.

7.1 Prolongation and Restriction

The interpolation operator R_H^T from coarse to fine mesh was introduced in §4. Its matrix representation is very simple: Suppose that $\{\phi_i^H\}_{i=1}^{N_c}$ are piecewise linear basis functions on the coarse mesh. If $\mathbf{u}^H$ is a vector defined at the coarse mesh nodes, then

$$\sum_i u_i^H \phi_i^H$$

is the corresponding piecewise linear interpolant on the coarse mesh and its value at each of the fine mesh nodes $\mathbf{q}_j^h$ is

$$\sum_i u_i^H \phi_i^H(\mathbf{q}_j^h) \ .$$

So assembly of the matrix representing R_H^T with respect to the usual hat function bases requires simply the evaluation of the coarse mesh basis functions at the fine mesh nodes i.e.

$$(R_H)_{ij} = \phi_i^H(\mathbf{q}_j) \tag{7.9}$$

(see also Hagger[20, Chapter 2]), Note that the fine mesh points requiring these calculations do not include any points lying on the Dirichlet boundaries as these have no associated freedom. It is of some practical importance to realise that the matrix R_H is necessarily sparse, since the non-zero entries only occur where fine mesh points lie inside the support of a coarse mesh basis function.

Two choices of the coarse mesh matrix A_H are available to us, firstly we could choose A_H as the stiffness matrix arising from the weak form of the elliptic problem discretised on the coarse mesh. The second choice is to choose A_H as

$$A_H = R_H A R_H^T. \tag{7.10}$$

Note that this second choice would be the same as the first if the coarse mesh was a nested version of the fine mesh, since in this case the fine space is a subspace of the coarse space.

7.2 Choice of coarse mesh

Subject to the constraints outlined above, at first glance there would seem to be a very great deal of freedom in the choice of coarse mesh. In particular - as proved in [11], [10] - apart from satisfying the assumptions (A1)-(A3) above (which mainly concern the area that the coarse mesh covers), the coarse mesh apparently need not bear any similarity to the fine mesh in order to yield an asymptotically optimal preconditioner - see §9 for a more detailed discussion of the notion of optimality. From one point of view this is perhaps not entirely surprising as one commonly understood function of the coarse mesh is to essentially build into the preconditioner a global connection between the subdomain problems, hence eliminating the deterioration caused by the subdivision of the domain.

However, this view point based on asymptotics of condition number estimates may be misleading. It is likely that in a real problem an effective coarse mesh problem must, to some extent, demonstrate certain qualitative features inherent in the fine mesh problem, otherwise the constant, $C(a)$, in (9.14) may become very large with respect to the mesh parameters. For example a fine mesh that is greatly refined in a certain region, to resolve a particular property of the elliptic PDE, would correspondingly require a coarse mesh that also is more refined in this region.

Coarsening methods for producing a coarse mesh from the fine mesh are discussed in many papers, see, for example, [5, §5] and [3]. For such methods, in order to satisfy the assumptions (A1) - (A3) some modification of the resulting coarse mesh may be necessary, as typically the resulting coarse mesh boundary is smaller than the original fine mesh boundary. See also [9] and [19] for more discussion on mesh coarsening.

Many of these coarsening procedures are aimed at producing a hierarchy of meshes in order to then apply multilevel solvers. Here we have only a two-level method and a more basic and cheaper coarsening method which we shall now describe can be applied.

7.2.1 The base coarse mesh

For the method outlined here we propose that the starting point of the coarse mesh is simply a *uniform* mesh, irrespective of the structure of the fine mesh. In particular the coarse mesh would be chosen to cover the entire fine mesh domain, for example in 2D by using a rectangle encompassing all the fine mesh points. The initial mesh diameter of the coarse mesh is determined by the diameter of the largest fine mesh element, together with some restrictions on the size, in terms of freedoms, of the coarse mesh problem.

Clearly using a uniform coarse mesh may result in some elements of the coarse mesh not satisfying assumption (A2), since they may lie outside the fine mesh, which could have an irregular boundary. However, this can be countered during the process of calculating the interpolation matrix (7.9), by disregarding any elements of this coarse mesh that contain no fine mesh freedoms.

From this initial stage a local refinement of the coarse mesh is performed, using hierarchical refinement based on the "slave variables" method discussed in [6, §7]. The base mesh is simply a mesh of triangles with the usual nodal basis functions, with support over the surrounding triangles. Correspondingly nodes that are produced by the refinement scheme again have nodal basis functions, but this time the support is over smaller triangles. Since the basis functions corresponding to the original nodes are non-zero at the new node points this is a hierarchical mesh. We illustrate this in Figure 2, where nodes $N_1 - N_4$ belong to the original mesh. The refined node, N_5, has support as shown by the shaded area, where the slave nodes $S_1 - S_4$ are needed to define the triangles for this support. Higher levels of refinement proceed in the same hierarchical fashion.

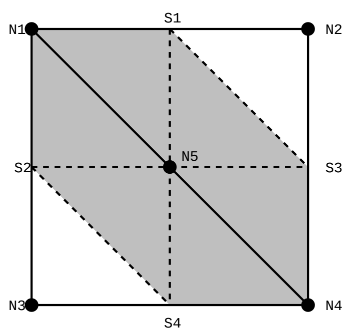

Figure 2: *Two levels of hierarchical basis functions, showing the slave nodes.*

To decide where to place the additional nodes each coarse element is examined to determine the number of fine mesh points lying inside it. This is then used as a criteria to decide where in the coarse mesh to refine. By balancing the number of refinements against the distribution of fine mesh points, we attempt to match the fine mesh structure, to obtain a coarse mesh that captures the essential properties of the fine mesh while at the same time maintaining a coarse mesh complexity which does not damage the overall parallel performance of the algorithm. We will now give more precise details on the practical details of this method.

7.2.2 Refinement Technique

For the purposes of determining the coarse mesh nodes the initial element is a square (or rectangle) in 2D and a cube in 3D. At the stage of computing basis functions these squares or cubes are divided up into an appropriate number of triangles or tetrahedra. The first level of refinement is to place an additional node at the geometric centre of this square or cube. For further refinements the element is then considered as 4 small elements in 2D, or 8 in 3D, and further nodes can again be placed at the geometric centres of these smaller elements, if they, in turn, contain more than a set tolerance of fine mesh points. This latter tolerance should not be too small in order to prevent the resulting coarse mesh being over-refined in one particular area.

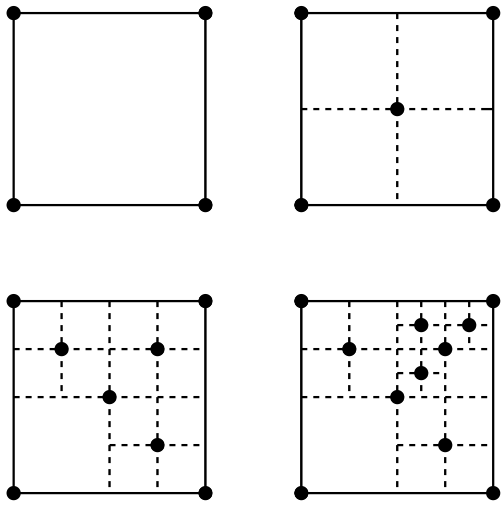

Figure 3: *Coarse mesh refinements*

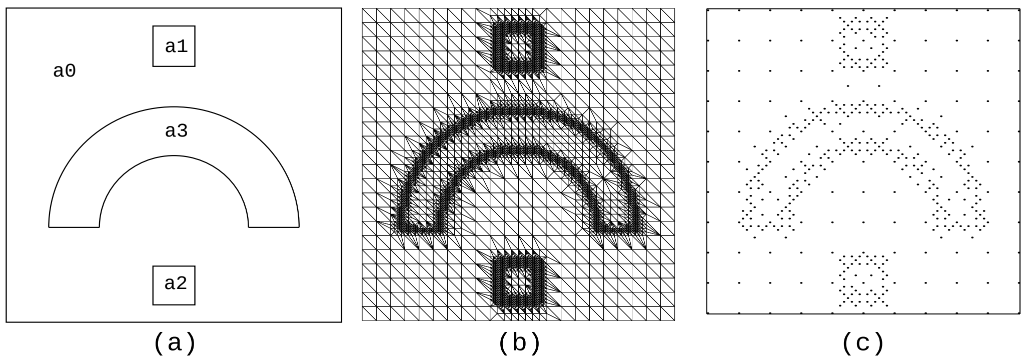

Figure 4: *2D Example of coarse mesh*

In Figure 3 we illustrate a possible sequence of refinements in 2D, starting from four nodes on a square. At each successive level of refinement the nodal basis functions are non-zero over smaller domains. The dashed lines indicate where the "slave nodes (or variables)" lie.

A more realistic example is given in Figure 4(a) - 4(c), this arises from a problem motivated by computational electromagnetics and discussed also in [17]. The PDE to be solved is Poisson's equation with coefficient having different constant values on each of the indicated regions in Figure 4(a). Figure 4(b) is the fine mesh used to discretise this problem. Figure 4(c) is the coarse mesh that results from the refinement technique discussed here. Results in §9, (together with numerical experiments in [17]), show that this coarse mesh is very effective for this problem.

7.3 Evaluation of R_H

Practical experience with calculating the interpolation matrix (7.9), see Hagger[20, Chapter 3], suggests that the approach of using a uniform base grid with hierarchical refinements would greatly simplify the required computations. For a uniform coarse mesh, evaluating the coarse basis functions at a specified point is very simple, since almost no work is needed to determine *where* in the coarse mesh the non-zero basis functions are. Contrast this with a coarsened mesh approach: Here additional work must first be done on the set of coarsened nodes to produce a triangulation, which can itself be a complicated task. Then for each fine mesh point we must locate in the coarse mesh which triangle this lies on - but since the coarse mesh is unstructured this may not itself be an easy task, and could require considerable searching of data structures.

Finally, we comment that the coarsening procedure proposed here would easily parallelise, as each processor could conveniently be given a specified region of the coarse domain to work on. At this time, however, this approach has not yet been implemented in DOUG.

7.4 An alternative approach

As we have seen, using a hierarchical coarse mesh has a number of practical advantages. One drawback is that at present there is no rigorous theory describing the behaviour of such a mesh and basis functions. An alternative is to use the same set of nodes for the coarse mesh, and produce a triangulation of this set of points. From this the interpolation matrix is then computed using just standard linear basis functions on triangles (or tetrahedra).

The standard approach to triangulating a set of points is to use a Delaunay triangulation, see [29]. The result of such an algorithm, using the mesh points as in Figure 4(c), is demonstrated in Figure 5.

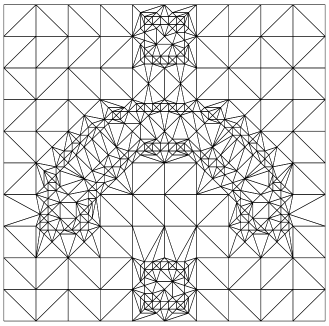

Figure 5: *2D Delaunay Triangulated coarse mesh*

Although arising from an unstructured mesh, this triangulated coarse mesh is essentially structured, so that the costs in locating where in the coarse mesh a particular point lies can be minimised. However, it will never be as straightforward as in the hierarchical version above.

Both of these coarse mesh approaches have been implemented in DOUG, the choice is left up to the user. The Delaunay triangulated mesh is generally more effective, with fewer overall number of iterations in the solver, but does take more time at the setup stage.

7.5 Choice of coarse problem

The second practical point concerns the choice of A_H. Since the coarse mesh that we propose does not necessarily have boundaries that coincide with the fine mesh, but merely encompasses them, it is difficult to see what boundary conditions must be imposed on the coarse problem if we were to construct the coarse mesh stiffness matrix, A_H by directly approximating the PDE in the coarse space. Moreover, DOUG has no knowledge of the finite element method or the PDE being solved, and so has no way of directly constructing the stiffness matrix for the coarse problem. However, we can avoid these difficulties by using instead the choice (7.10) for the coarse mesh problem. The coarse problem is then constructed algebraically from the fine problem using merely the prolongation and restriction matrices.

Practically we do not compute the coarse mesh stiffness matrix precisely from equation (7.10), since the fine mesh stiffness matrix A is never assembled. Instead we assemble A_H in a sparse form as

$$A_H = R_H \left(\sum_{E \in \mathcal{E}} A_E \right) R_H^T. \tag{7.11}$$

Note that this computation can easily be performed in parallel, since each slave has only elements $\mathcal{E}_i$. Hence by providing the slave with the rows of R_H^T that correspond to the freedoms $\mathcal{F}_i$ it can compute a section of the coarse problem. The full coarse problem A_H is then assembled on the master processor, where it is solved (using a direct factorisation, in this case).

7.6 Size of the coarse problem

One important issue concerns the size of the coarse problem, N_H. As the coarse mesh is constructed in an automatic fashion, we must first decide the maximum number of nodes that it may have. A complexity analysis for the structured mesh case (see [8]) indicates that the best choice of coarse problem depends on a number of factors including the number of subdomain problems, the method of solving them and the number of processors available. The situation in the structured case is simplified in that the coarse mesh size, H, is closely related to the diameter of the subdomains. For the unstructured case this is not so simple, as the subdomains are now constructed using algebraic load-balancing methods, so that the diameters can vary widely in general.

In the DOUG implementation the coarse mesh solve is carried out on the master processor. Since this operation is to be performed at the same time as the slave processors are computing their subdomain solves it is apparent that the coarse problem must not be too large compared to the subdomain problems, otherwise the preconditioning solve will be *dominated* by the coarse mesh solves, and there will be no parallel speedup at all. This gives an approach for determining the size of the coarse problem, we require that for each processor

$$\text{Work on coarse problem} \;<=\; \text{Total work to solve subproblems.}$$

Assuming that we are using a direct factorisation method for solving both the coarse problem and the subproblems, and that the LU factors are found and stored in advance of the solution process, the work estimate for each backsolve of a problem of size n is $8n^{3/2}$ in 2D and $8n^{5/3}$ in 3D. (This is based on a frontal solver method, and assumes that the problems come from a finite element setting such that the matrices are banded, see [14] and [20, Chapter 1]) Each slave processor is given a problem with approximately N/N_p freedoms, which it will further subdivide such that each subproblem has N_s degrees of freedom, where $N_s = 1000$ in the default setting for DOUG. Hence N_H should have an upper bound given by

$$N_H = \left[\left(\frac{N}{N_s N_p} \right) N_s^{3/2} \right]^{2/3}, \text{ in 2D,}$$

$$N_H = \left[\left(\frac{N}{N_s N_p} \right) N_s^{5/3} \right]^{3/5}, \text{ in 3D.}$$

For our present implementation one quarter of these freedoms are first allocated to the uniform part of the coarse mesh, the part built before the refinement takes place. The remaining freedoms are then available for use by the hierarchical refinement process.

7.7 Parallel Implementation

Three main stages are involved in the coarse mesh solve: Firstly we must restrict a fine mesh vector, $\mathbf{r}$, into a coarse mesh vector, that is, compute $\mathbf{r}_H = R_H \mathbf{r}$. Secondly the linear system $A_H \mathbf{z}_H = \mathbf{r}_H$ must be solved, and thirdly the coarse vector must be prolonged into a fine vector, $\mathbf{z} = R_H^T \mathbf{z}_H$.

To minimise communication costs and maximise parallelisation of this operation, the first and third stages are carried out by the slave processors. More precisely, at the beginning of a preconditioner solve step, each slave computes the action of the restriction on the part of the fine mesh vector that it has stored and transmits it to the master processor. The master processor then assembles a coarse mesh vector from these parts, solves the linear system and then transmits back (via non-blocking sends) parts of this solution to each slave. Meanwhile the slaves are solving the subdomain problems, and begin their communications related to this solve. As soon as a slave receives its part of the coarse mesh vector it prolongs it and adds the result onto its local vector.

We illustrate the implementation of the preconditioner solve stage in the following algorithm:

Algorithm 2 [Msolve] Compute the action of the preconditioner in parallel

1) Restrict the local vector and send to the master, i.e. compute $R_H \mathbf{x}_i$.

2) Start the non-blocking receive for the result from the master.

3) Compute the local subdomain solve(s)

4) Send updates on shared entries to the other slaves (as in the matrix-vector algorithm).

5) Wait for the shared entry receives or the result from the master. If the result from the master has arrived then immediately prolong it and add the result to the local vector.

It is crucial that the receive from (2) completes before the work in (3) is complete, otherwise the coarse mesh solve will begin to dominate the preconditioner work, and there will be no effective parallel speedup. In practice this is achieved by limiting the size of the coarse mesh as in §7.6.

8 Parallelisation results

Here we give preliminary results using DOUG on a variety of parallel architectures. The linear system has $N = 15000$ and arises from the solution of a problem in electromagnetics, based on a mesh as in Figure 4(b). More specific details of this problem are given in §9, where it corresponds to Experiment 1, problem 5.

In the following the number of processors refers to the number of slave processors, an additional *master* processor is also being used, but the majority of work is done by the slaves.

Four different computer architectures were tested for parallel efficiency ("speedup") using the DOUG code: Firstly an Intel i860 provided by Daresbury Laboratories (Daresbury, Warrington, UK). This allows use of up to 15 processors in this case. Secondly an IBM SP/2, again provided by Daresbury Laboratories, up to 7 processors could be used in this case. The third architecture was a SUN Solaris Sparc-1000 multiprocessor, up to 3 processors could be used in this case, and communication is via shared memory. Finally an ethernet cluster of UNIX machines was tested, this was comprised of a mixture of SUN Solaris and SGI machines.

#Proc	Solver time	Efficiency
3	16.4s	-
7	7.59s	92%
15	4.45s	92%

Table 1: Intel i860

#Proc	Solver time	Efficiency
2	3.005s	-
3	2.0436s	98%
4	1.6562s	90%
5	1.3255s	90%
6	1.1292s	89%
7	1.0214s	84%

Table 2: IBM SP/2

In all but the IBM case, the MPICH implementation of MPI was used for parallel communication, the SP/2 has a vendor implementation of MPI. Note that exactly the same Fortran code was used on all the different machines. In the calculation of parallel efficiency, we let p_0 denote the smallest number of processors tested and t_0 the time for that configuration to solve a given problem. Then if p processors require time t for that problem, we set

$$\text{efficiency} = \frac{t_0 \times p_0}{t \times p} \times 100.$$

A 100% efficiency would then mean an optimal parallel speedup from the smallest number of processors.

#Proc	Solver time	Efficiency
1	32.2s	-
2	16.5s	98%
3	10.6s	101%

Table 3: SUN Solaris

#Proc	Solver time	Efficiency
1	32.75s	-
2	19.35s	85%
3	12.12s	90%
4	9.87s	82%

Table 4: SUN/SGI network

Overall, the parallel efficiency results are quite promising. The apparent anomaly in the SUN Solaris multiprocessor results in Table 3 is explained by the fact that this machine is shared by other users, so the timings can fluctuate to a small degree. The same is also true for the results in Table 4, although here the ethernet performance can vary as well.

Although not reported here, the floating point performance of the core routines, that is the matrix-vector multiply and preconditioner solve, is good for all the machines. Even for quite large numbers of processors, the overhead in communication times is small compared to the total computation time. The approach of always trying to overlap the parallel communication with computations is believed to be the main reason for this success.

In particular the results for the ethernet cluster are very surprising. Conventional wisdom is that the relative slow speed of ethernet connections would make the parallel efficiency quite low. It is believed that there are a number of reasons for our good performance, including the fact that messages from one processor to another are parcelled together in large blocks before sending, thereby reducing the total communication latency time.

9 Numerical Experiments

In this section we give some numerical experiments, computed using the DOUG software illustrating some recent results and open questions concerning the performance of additive Schwarz preconditioning (4.5) for certain ill-conditioned symmetric problems. Firstly we give a brief review of the theory of (4.5).

9.1 Theory

Consider the elliptic PDE of the form

$$-\nabla . a \nabla u = f \text{ on } \Omega \ , \tag{9.12}$$

$$u = g \text{ on } \partial\Omega_D \ , \qquad \frac{\partial u}{\partial n} = g' \text{ on } \partial\Omega_N \ , \tag{9.13}$$

where $\Omega \subset \mathbb{R}^2$ or $\mathbb{R}^3$ is a polygonal or polyhedral domain with boundary $\partial\Omega$ which is partitioned into two pieces $\partial\Omega_N$ and $\partial\Omega_D$. In 2D we assume $\partial\Omega_N$, $\partial\Omega_D$ consist entirely of straight line segments; in 3D we assume they consist of planar pieces. The coefficient a is a positive valued piecewise constant function on Ω, which may have large jumps across internal boundaries.

This is relevant to many applications such as groundwater flow, electromagnetics and semiconductor modelling (e.g. [1, 2, 12, 13, 20, 21, 22, 23]) where the coefficients have large discontinuities across interfaces between (often complicated) regions with differing material properties.

Suppose this problem is discretised by piecewise linear finite elements as in §3 and we solve the resulting linear system by conjugate gradients with preconditioner $M = M_{ASC}$ as in (4.5). The theory ([6, 10]) shows that in the general unstructured case the condition number, κ, of the preconditioned system has an upper bound given by

$$\kappa(M^{-1}A) \leq C(a)\left(1 + \frac{H}{\delta}\right)^2, \tag{9.14}$$

where H is the diameter of the coarse mesh, δ is the size of the overlap of the subdomains and C is independent of h, H and δ. Hence by a suitable choice of overlapping subdomains and coarse mesh problem, the convergence of the algorithm is independent of the mesh size, h.

There are a number of interesting theoretical (but practically relevant) questions related to this estimate. First it is known that (as indicated), $C(a)$ depends on the coefficient a (in fact in the worst case it grows linearly in $\max a/\min a$ - see [18]). In [18], [17] we have proved that this growth in condition number is fortunately not translated into a correpondingly poor convergence of the preconditioned conjugate gradient method. - in fact it converges extremely well even if a is very discontinuous. This is illustrated and explained in Experiment 1 below.

In Experiment 2 we examine the robustness of the automatic coarse mesh generation technique described in §7. We do this in the context of the singularly perturbed problem

$$\lambda\nabla^2 u + u = d \tag{9.15}$$

where d is discontinuous, λ is a small parameter and, as $\lambda \to 0$, u has an interior layer of decreasing width across the discontinuity in d. The question arises whether a uniform coarse grid will provides an adequate coarsening (in terms of quality of preconditioner) whenever λ is small and, if the answer is no, whether the cheap adaptive coarsening procedure described in §7 will be robust as $\lambda \to 0$.

9.2 Experiment 1

Here we consider a two dimensional problem motivated by static field computations in electromagnetics, see [15], the problem in this case is in the form (9.12) (9.13) . Here there are three interior regions with varying material properties, as in Figure 4(a). Homogeneous Neumann conditions are imposed on the left and right boundaries and Dirichlet conditions 1 and 0 are imposed on the top and bottom respectively.

To solve this problem we used finite elements with a mesh as shown in Figure 4(b). This features 30856 triangular elements, and results in 15000 degrees of freedom. Five different sets of material properties were tested, as shown in Table 5.

In Table 6 we show the number of conjugate gradient (CG) iterations required to satisfy the convergence criterion of reducing the 2-norm of the residual by a factor of 10^{-5}. To demonstrate the robustness of the method against discontinuous coefficients five different

Problem	a0	a1	a2	a3
1	1.0	1.0	1.0	1.0
2	1.0	1.0(-1)	1.0(1)	1.0(2)
3	1.0	1.0(-2)	1.0(2)	1.0(4)
4	1.0	1.0(-3)	1.0(3)	1.0(6)
5	1.0	1.0(-4)	1.0(4)	1.0(8)

Table 5: Coefficient values for the test problems

problems are tested, each based on the same mesh, but have progressively worsening discontinuities. In addition, for demonstration purposes only, we have modified the code to allow a more arbitrary number of subdomains, so as to examine the effectiveness of the coarse mesh with respect to this.

Three different preconditioners are tested in each case: One with no coarse solve (AS), a second with a coarse solve but based on a purely uniform coarse mesh, with 121 nodes, (ASC*), and the third with coarse mesh pictured in Figure 4(c) (ASC), which has 465 nodes, and was generated automatically by DOUG. An initial guess of **0** was used for the CG algorithm.

15 Subdomains				30 Subdomains			
Problem	AS	ASC*	ASC	Problem	AS	ASC*	ASC
1	70	25	27	1	87	26	28
2	92	27	26	2	116	31	28
3	109	30	29	3	131	35	33
4	117	31	31	4	149	36	35
5	135	31	31	5	165	36	34
60 Subdomains				**120 Subdomains**			
Problem	AS	ASC*	ASC	Problem	AS	ASC*	ASC
1	123	33	33	1	158	39	34
2	150	36	32	2	186	40	36
3	172	38	37	3	213	44	39
4	196	39	37	4	241	45	43
5	220	39	37	5	272	45	44

Table 6: Results for the 15,30,60 and 120 subdomain case, for the electromagnetics problem.

We have proved in [18], [17] that, notwithstanding the linear growth of $C(a)$ in (9.14) with respect to the ratio $\max a / \min a$, the number of conjugate gradient iterations grows only logarithmically in this ratio. The results for the preconditioner with no coarse solve (AS) show the expected logarithmic growth in the number of CG iterations required, as the jumps in the coefficient worsen. Additionally an increase in the number of subdomains also results in the expected increase in iterations (see §9.14). For the remaining two preconditioners (ASC* and ASC) the growth with respect to the number of subdomains is almost identical, indicating that, for this problem, a simple uniform coarse mesh would be sufficient, and the refinement is superfluous. Here the growth of the number of iterates is only very mild indeed leading to the conjecture that a suitable coarse mesh may completely remove the dependence on the coefficient, even when it does not resolve the discontinuity. This is an open question in the theory of Schwarz methods and is the subject of ongoing work.

9.3 Experiment 2

To more closely examine the effect of the coarse mesh problem, and how important it is to have a coarse mesh that matches the fine mesh to some extent, we consider the problem (9.15) with the function d taken to be piecewise constant, with two values on Ω as shown in Figure 7. The Dirichlet condition $u = d$ is imposed on $\partial\Omega$. For $\lambda = 0$ the exact solution is $u = d$, and for small $\lambda > 0$ there is a layer of width $O(\lambda)$ across which u changes rapidly.

Since we are using finite elements to solve this problem it is clear that, in order to resolve the layer, more nodes will be needed near the interface as λ is made smaller. To test how necessary it is for the structure of the coarse mesh to reflect that of the fine mesh (see §7.2), we construct a "fair" fine mesh that has an optimal, or near optimal, placing of mesh points, in the sense that the error is equidistributed (see [4]), over the domain.

For the corresponding one-dimensional problem on $[0,1]$ with Dirichlet boundary conditions it is quite easy to do this. If the jump in d is placed at $x = 1/2$ and we then assume that the true solution is well approximated for small λ by

$$u = \tanh\left(\frac{x - \frac{1}{2}}{\lambda}\right),$$

To produce an equidistributed mesh we use a *monitor function*, M, (see [4]). The n mesh points, x_i, are then chosen such that

$$\| M \|_{[x_{i-1},x_i]} = \| M \|_{[x_i,x_{i+1}]}, \text{ for } i = 1,..,n, \tag{9.16}$$

where the norm is the L^1 norm. (This means that we must take care to ensure the monitor function is positive valued for our domain.) The monitor function that we shall use is given by $\left[u'''\right]^{\frac{1}{4}}$. The optimal mesh points, x_i, in the domain $(0, 1/2)$ are then chosen such that

$$\int_{x_i}^{x_{i+1}} \left[u'''\right]^{\frac{1}{4}} dx = C,$$

for a constant value C, which is determined by the integral of the monitor function over $(0, 1/2)$. The points for $(1/2, 1)$ are then simply a reflection about $x = 1/2$. Additionally a mesh point is always placed at $x = 1/2$.

Examples of the optimal placing of a small number of mesh points for various values of λ are shown in Figure 6. Notice that for $\lambda = 10^{-3}$ the points are very closely clustered around the jump, as we would expect, since the layer is $O(\lambda)$. This distribution is then used to create a "fair" mesh for the two dimensional problem (9.15). An example of this is given in Figure 8, for the $\lambda = 10^{-2}$ case.

The two-dimensional problem is on a unit square, the line dividing the two regions with different values of d is chosen as an oblique line such that the solution has no simple symmetry about the jump, see Figure 7.

For a range of values of λ, the fine mesh has approximately 10000 degrees of freedom and 19000 elements. For the additive Schwarz preconditioner the number of subdomains

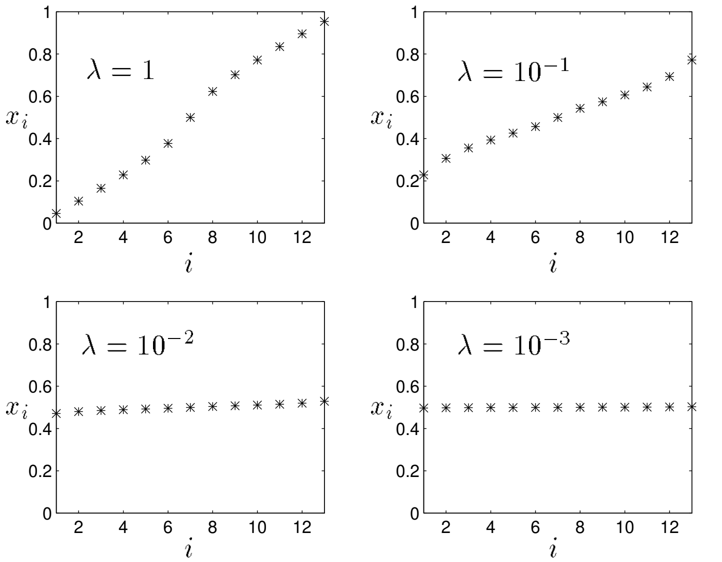

Figure 6: Optimal placing of 13 mesh points for $\lambda = 1, 10^{-1}, 10^{-2}$ and 10^{-3}.

has been fixed at 60. We now consider the effect of using a succession of increasingly refined coarse meshes. Two different types of coarse mesh are considered, one using a hierarchical refinement and Delaunay triangulation (as in §7.2.2), and secondly a series of uniform meshes, with similar numbers of nodes.

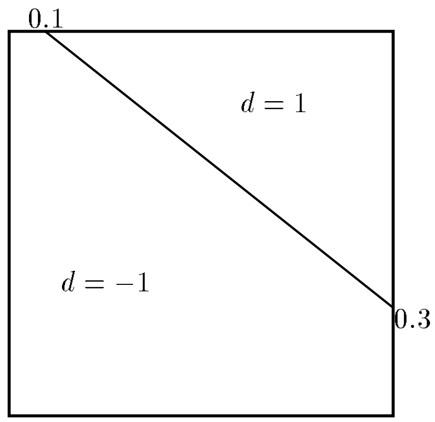

Figure 7: Domain for Experiment 2

In Table 7 we show the number of conjugate gradient iterations required for a sequence of hierarchically refined coarse meshes, for various values of λ (N_c is the number of nodes in the coarse mesh). Note that the first column is simply a uniform coarse mesh, the later columns are for refinements of this mesh. As λ is made smaller we see a large increase in the number of iterations required with the uniform mesh, but this increase is much smaller for the hierarchically refined coarse mesh. In Table 8 the results for simply a uniform coarse mesh are given. Although using a more refined coarse mesh does help in this case, it does not give as much improvement as in the hierarchically refined case, for

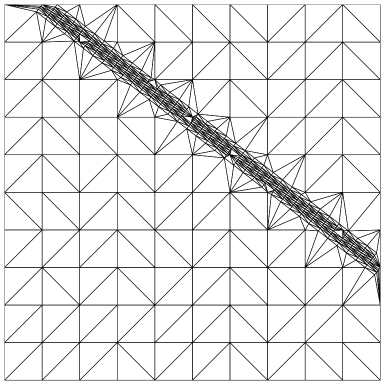

Figure 8: A 2D mesh created using an optimal 1D mesh

$\lambda \backslash N_c$	90	190	290	390	490	680
10^{-1}	31	23	22	22	21	18
10^{-2}	50	35	29	28	24	22
10^{-3}	115	79	76	72	58	48
10^{-4}	221	159	158	154	116	99
10^{-5}	346	303	275	229	200	101

Table 7: Iterations required for hierarchically refined coarse meshes, with Delaunay triangulation

the same number of nodes.

$\lambda \backslash N_c$	72	156	210	306	420	783
10^{-1}	33	25	24	21	20	18
10^{-2}	54	42	40	38	37	31
10^{-3}	132	112	113	97	94	76
10^{-4}	245	224	212	217	179	154
10^{-5}	352	343	349	334	290	287

Table 8: Iterations required for uniform coarse meshes

In practice it is highly likely that a fine mesh is produced using an adaptive refinement method, or some other technique that aims to produces optimal meshes. Hence, based on this experience, it appears to be important that the coarse mesh reflects this.

References

[1] R. E. Alcouffe, A. Brandt, J. E. Dendy,Jr, and J. W. Painter, *The multigrid method for the diffusion equation with strongly discontinuous coeffients*, SIAM J. on Sci. Stat. Comp., 2 (1981), pp. 430–454.

[2] S. F. Ashby, R. D. Falgout, S. G. Smith, and A. F. B. Tompson, *Modeling groundwater flow on MPPS*, in Proceedings of the scalable parallel libraries conference, 1993.

[3] R. E. Bank and J. Xu, *An algorithm for coarsening unstructured matrices*, Numer. Math., 73 (1996), pp. 1–36.

[4] G. F. Carey and H. T. Dinh, *Grading functions and mesh redistribution*, SIAM J. Numerical Analysis, 22 (1985), pp. 1028–1040.

[5] T. F. Chan, S. Go, and J. Zou, *Multilevel domain decomposition and multigrid methods for unstructured meshes: Algorithms and theory*, in Proceedings of the Eighth International Conference on Domain Decomposition, R. Glowinski, J. Périaux, Z.-C. Shi, and O. B. Widlund, eds., Chichester, 1996, Wiley and Sons.

[6] T. F. Chan and T. P. Mathew, *Domain decomposition algorithms*, in Acta Numerica, A. Iserles, ed., CUP, 1994, pp. 61–143.

[7] T. F. Chan, T. P. Mathew, and J. P. Shao, *Efficient variants of the vertex space domain decomposition algorithm*, SIAM J. on Sci. Comp., 15 (1994), pp. 1349–1374.

[8] T. F. Chan and J. P. Shao, *Parallel complexity of domain decomposition methods and optimal coarse grid size*, Parallel Computing, 21 (1995), pp. 1033–1049.

[9] T. F. Chan and B. Smith, *Domain decomposition and multigrid algorithms for elliptic problems on unstructured meshes*, Contemporary Mathematics, 180 (1994), pp. 175–189.

[10] T. F. Chan, B. Smith, and J. Zou, *Overlapping schwarz methods on unstructured meshes using non-matching coarse grids*, Numer. Math., 73 (1996), pp. 149–167.

[11] T. F. Chan and J. Zou, *Additive Schwarz domain decomposition methods for elliptic problems on unstructured meshes*, Numerical Algorithms, 8 (1994), pp. 329–346.

[12] R. K. Coomer and I. G. Graham, *Domain decomposition methods for device modelling*, in Domain Decomposition Methods in Science and Engineering, D. Keyes and J. Xu, eds., Providence, 1995, American Mathematical Society.

[13] ——, *Massively parallel methods for semiconductor device modelling*, Computing, 56 (1996), pp. 1–28.

[14] I. S. Duff, A. M. Erisman, and J. K. Reid, *Direct methods for sparse matrices*, Oxford University Press, 1986.

[15] C. R. I. Emson, J. Simkin, and C. W. Trowbridge, *A status report on electromagnetic field computation*, IEEE Trans. Magnetics, 30 (1994), pp. 1533–1540.

[16] G. H. Golub and C. F. van Loan, *Matrix Computations, second edition.*, John Hopkins University Press, 1989.

[17] I. G. Graham and M. J. Hagger, *Additive Schwarz, CG and discontinuous coefficients*. To appear in the Proceedings of the Ninth International Conference on Domain Decomposition methods, 1997.

[18] ——, *Unstructured additive Schwarz - CG method for elliptic problems with highly discontinuous coefficients*. To appear in SIAM J. Sci. Comp., 1997.

[19] H. Guillard, *Node-nested multi-grid method with Delaunay coarsening*, tech. report, INRIA, Sophia Antipolis, France, 1993.

[20] M. J. Hagger, *Iterative solution of large, sparse systems of equations, arising in groundwater flow models*, PhD thesis, University of Bath, 1995.

[21] B. Heise, *Parallel solvers for fem-bem equations with applications to non-linear magnetic field problems*, in Numerical Treatment of Coupled Systems, Notes on Numerical Fluid Dynamics Vol 51, W. Hackbusch and G. Wittum, eds., Braunschweig, 1995, Vieweg.

[22] B. Heise and M. Kuhn, *Parallel solves for linear and non-linear exterior magnetic field problems based on upon coupled FE/BE formulations*, tech. report, Institutsbericht Nr. 486, Johannes-Kepler-Universtat, Linz, 1995.

[23] M. C. Hill, *Solving groundwater flow problems by conjugate-gradient methods and the strongly implicit procedure*, Water resources research, 26 (1990), pp. 1961–1969.

[24] G. Karypis and V. Kumar, *METIS: Unstructured graph partitioning and sparse matrix ordering system*, Dept. Computer Science, Univ. of Minnesota, Minneapolis, August 1995.

[25] Y. Saad and M. H. Shultz, *GMRES: A generalized minimal residual algorithm for solving non-symmetric linear systems*, SIAM J. on Sci. Stat. Comp., 7 (1986), pp. 856–869.

[26] B. F. Smith, *An optimal domain decomposition preconditioner for the finite element solution of linear elasticity problems*, SIAM J. on Sci. Stat. Comp., 13 (1992), pp. 364–378.

[27] B. F. Smith, P. E. Bjorstad, and W. D. Gropp, *Domain decomposition : parallel multilevel methods for elliptic partial differential equations*, Cambridge University Press, Cambridge, 1996.

[28] H. A. Van der Vorst, *BI-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems*, SIAM J. Sci. Stat. Comput., 13 (1992), pp. 631–644.

[29] D. F. Watson, *Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes*, The Computer Journal, 24 (1981), pp. 167–172.