A Domain Decomposition Preconditioner for a Parallel Finite Element Solver on Distributed Unstructured Grids

D.C. Hodgson and P.K. Jimack
School of Computer Studies
University of Leeds
Leeds LS2 9JT, UK
E-mail: pkj@scs.leeds.ac.uk

Key words. Distributed Memory, Unstructured Grids, Domain Decomposition Preconditioning.
AMS(MOS) subject classification. 65Y05, 65N55, 65N50, 65N30, 65F10.

Abstract

A number of practical issues associated with the parallel distributed memory solution of elliptic partial differential equations using unstructured meshes in two dimensions are considered. The first part of the paper describes a parallel mesh generation algorithm which is designed both for efficiency and to produce a well-partitioned, distributed mesh, suitable for the efficient parallel solution of an elliptic p.d.e. The second part of the paper concentrates on parallel domain decomposition preconditioning for the linear algebra problems which arise when solving such a p.d.e. on the unstructured meshes that are generated. It is demonstrated that by allowing the mesh generator and the p.d.e. solver to share a certain coarse grid structure it is possible to obtain efficient parallel solutions to a number of large problems. Although the work is presented here in a finite element context, the issues of mesh generation and domain decomposition are not of course strictly dependent upon this particular discretization strategy.

1 Introduction

Finite element methods for the solution of elliptic partial differential equations (p.d.e.'s) on complex domains using unstructured grids have been in use for a number of years. Recently many sophisticated and reliable algorithms have been developed which involve efficient mesh generation and refinement as well as the use of effective preconditioning, leading to the production of a considerable range of quality software, such as the public domain package PLTMG ([3]) for example. Unfortunately, many of the advances incorporated into software such as this are yet to be carried over to non-serial computers, such as distributed memory parallel architectures, with much recent research into the scalable solution of unstructured finite element problems concentrating mainly on the problem of data partitioning [4, 9, 14, 15, 16, 23, 28, 34]. For this paper we take the view that in order to produce efficient, scalable, parallel algorithms it is necessary to consider the data partitioning problem in conjunction with the issues of parallel mesh generation and practical parallel solution (i.e. preconditioning).

The remainder of this introductory section gives further background to the problem under consideration: the parallel solution of 2-d elliptic p.d.e.'s using a distributed unstructured mesh of triangles and an effective parallel preconditioner. Then in sections 2 and 3 we discuss in more detail some new parallel algorithms for mesh generation and domain decomposition preconditioning respectively. The parallel mesh generation is a variant of the work described in [31] however it is designed to be faster and to produce a two level (coarse/fine) mesh data structure. This data structure is then used in the preconditioning algorithms to generalize the work of [7] to unstructured meshes. The final section contains a complete numerical example using the methods developed and a discussion of this and future work.

1.1 Mesh Generation and Data Partitioning

The use of unstructured meshes has a number of advantages over structured meshes, especially when solving problems on complex domains or with the aid of local refinement. As we will see however, for distributed memory parallel computation unstructured meshes can complicate the issues of load balancing and communications significantly. For the solution of finite element problems typical algorithms involve decomposing the mesh into a number of subdomains and allocating each of these to a processor. The contributions to the stiffness matrix (or other finite element matrices) can then be computed element-by-element in parallel on each processor (with a partial assembly being performed for each subdomain as well if desired). Finally, the linear algebra is performed in a distributed fashion across the processors in some manner which should aim to keep the amount of data transfer between processors as low as possible. Hence the decomposition of the elements of the mesh into subdomains should have two main features: each processor should store approximately the same number of node points or elements (to facilitate load balancing), and the number of node points which lie on the boundary between processors should be kept low (since the amount of interprocessor communication is likely to depend heavily upon this).

As has already been indicated there has been a considerable amount of recent research into the problem of partitioning an existing unstructured mesh across distributed memory in a manner compatible with the above requirements. However this is a far from ideal approach since if we wish to solve very large problems in parallel we do not want the mesh generation itself to be a serial bottleneck. Neither do we wish to have to use secondary storage in order to hold a complete mesh which is too large for the primary memory of a single processor. For these reasons we address the problem of generating partitioned meshes in parallel.

The requirement therefore is to generate a mesh in parallel such that, upon completion, each processor has generated about the same number of nodes and the mesh is distributed such that the number of nodes lying on the boundary between processors is as low as possible. Note that this approach differs to that of [2] or [21] for example, where the generation is simply farmed out to waiting processors. Instead it is more akin to the philosophy of [24] and [31] which both make use of a coarse background grid which is partitioned amongst the processors before the generation of the final mesh commences. In fact this background grid turns out to be very important, not only for the parallel mesh generation (see section 2), but also for the domain decomposition preconditioning that is described in section 3.

1.2 Preconditioning

When solving large sparse linear systems of equations such as those that arise when applying the finite element method on a very large or fine mesh, it is usual to use iterative techniques. A good iterative method should not only require considerably less memory than a direct solver but it should also yield solutions in less time. For this paper we restrict our attention to self-adjoint elliptic p.d.e.'s which give rise to symmetric positive definite matrix problems and so allow the use of the preconditioned conjugate gradient method. The rate of convergence of the conjugate gradient algorithm is governed by the condition number of the linear system and this is known to increase significantly for finite element matrices as the mesh size, h, decreases [20]. For this reason it is important that a good preconditioner is available when solving problems on large meshes, such as those that we are likely to wish to generate in parallel.

A number of preconditioners have been used in the finite element solution of elliptic p.d.e.'s on unstructured meshes, such as simple diagonal scaling or incomplete Choleski factorization for example [12, 26]. The second of these is generally superior, especially if a good ordering is chosen for the unknowns [8]. However, parallel methods based upon incomplete factorizations are usually difficult to implement efficiently due to the triangular solves that must be performed at each preconditioning step. From a parallel implementation point of view domain decomposition preconditioners are far more attractive. As the name suggests these methods make use of local solutions, which can be computed independently on decomposed subdomains, in order to precondition the problem. Unfortunately nearly all of the work that has been done on domain decomposition methods appears to be restricted to structured meshes on regular geometries [6, 7, 19, 25, 29]. In section 3 of this paper we show how a certain class of these techniques, known as iterative substructuring methods [6, 29], can be extended to deal with unstructured meshes on irregular geometries, provided we make use of the presence of the underlying background grid that is used in the mesh generation.

2 Parallel Generation of Partitioned Unstructured Meshes

As explained in section 1 our first objective is to generate in parallel a mesh which is distributed across a number of processors in such a way as to approximately minimize the amount of interprocessor communication that will be required when obtaining finite element solutions on this mesh, subject to the constraint that the computational load (in obtaining such solutions) be equally spread amongst the processors. In the first instance we model these requirements by aiming to minimize the number of generated nodes which lie on the boundary between regions on different processors, whilst also attempting to generate an equal number of nodes on each processor. The algorithm that follows makes use of a coarse background grid which covers the domain (or an approximation to it where the boundary region is curved or very detailed). Following [31, 32] the algorithm also requires that at each knot point of the background grid we have a value, which we will refer to as a point distribution value, which defines the desired mesh spacing of the generated mesh in the immediate neighbourhood of this point (note that [32] describes a sequential rather than a parallel algorithm). These point distribution values may be obtained in numerous possible ways: based upon automatic error estimates, initial solutions, the domain geometry or other user-defined requirements.

Using the generator in an adaptive context, the distribution values can be determined by solving the finite element problem on the coarse mesh, or an intermediate mesh of uniform density, and estimating the errors over each element. In order to generate a suitable mesh for an initial value problem, the point distribution values may be based upon the given data so that the initial solution is accurately represented by the mesh (i.e. the mesh is refined where necessary). The generator can be used to produce meshes in which the point creation is driven by the boundary point distribution by leaving the point distribution values as they are after the background grid generation stage and applying a smaller value of the global mesh density factor for the subsequent production of the fine mesh. If the desired distribution of vertices in the final grid is known *a priori*, then the point distribution values can be set by hand or by using a suitable function.

Given a set of point distribution values at each knot point of the background grid, the first stage of the mesh generation algorithm is to cheaply estimate how many nodes will be generated along each edge of this grid and within each element (it is in this respect that our mesh generation algorithm differs most significantly from that in [31]: see subsection 2.1). Following this a weighted graph is produced such that each vertex of the graph represents a coarse element and has a weight equal to the number of predicted nodes for that element, and each edge of the graph represents an edge of the background grid and has a weight equal to the number of nodes that will be generated along it. The vertices of this weighted graph are then partitioned between the processors so as to ensure that the sum of the vertex weights on each processor is about the same, whilst the sum of the weights of those edges joining vertices on different processors is as low as possible. Finally the finite element mesh itself is generated in parallel, with each processor generating the portion of the mesh corresponding to the coarse background elements which it was allocated during the partitioning stage. If the *a priori* estimates of the number of nodes generated in each coarse element and along each coarse edge are sufficiently accurate then the mesh produced by this algorithm will be distributed according to both the load balancing and the low data communication requirements outlined above.

2.1 Estimating Edge and Vertex Weights

The formulae that we use for estimating the number of nodes that will be produced by our parallel mesh generator are as simple and cheap as possible (unlike the more expensive neural network approach used in [31]), whilst still giving reliable predictions. For the number of nodes generated on an edge, E_{ij} say, of the coarse grid we use

$$w_{E_{ij}} = \left\lfloor \frac{\ell_{ij}}{\alpha \frac{\delta_i + \delta_j}{2}} \right\rfloor \tag{2.1}$$

where i and j are the knot points at the ends of the edge, ℓ_{ij} is the length of the edge, δ_i and δ_j are the point distribution values at knot points i and j respectively and α is a scaling value which controls the grid point density. For the number of nodes generated in an element, N_i say, of the coarse grid we initially use the related estimate

$$w_{N_i} = \frac{A_i}{\left(\alpha \frac{\delta_{i_1} + \delta_{i_2} + \delta_{i_3}}{3}\right)^2} \tag{2.2}$$

where A_i is the area of the coarse element and δ_{i_1}, δ_{i_2} and δ_{i_3} are the point distribution values at its three vertices.

As explained below the first of these estimates, for the edge weights, turns out to be exact due to the nature of the parallel mesh generation. The second of these estimates works well for coarse elements for which the three point distribution values, δ_{i_1}, δ_{i_2} and δ_{i_3}, are similar but needs to be improved when they differ significantly. This is achieved by a technique which we call "virtual refinement" in which an imaginary vertex is placed at the centroid of the coarse element (with a point distribution value of $(\delta_{i_1}+\delta_{i_2}+\delta_{i_3})/3$) and w_{N_i} is then taken as the sum of the estimates for the three child elements. By employing this technique to "virtually refine" a background element a number of levels, the performance of the estimator is greatly improved. As an illustration of the effect of this we see that, with one level of refinement, (2.2) is modified to become

$$w_{N_i} = \frac{A_i}{3\left(\alpha\frac{\delta_{i_1}+\delta_{i_2}+\delta_{i_3}}{3}\right)^2}\left[\frac{1}{\left(1+\frac{\delta_{i_1}+\delta_{i_2}-2\delta_{i_3}}{3(\delta_{i_1}+\delta_{i_2}+\delta_{i_3})}\right)^2}+\frac{1}{\left(1+\frac{\delta_{i_1}-2\delta_{i_2}+\delta_{i_3}}{3(\delta_{i_1}+\delta_{i_2}+\delta_{i_3})}\right)^2}+\frac{1}{\left(1+\frac{-2\delta_{i_1}+\delta_{i_2}+\delta_{i_3}}{3(\delta_{i_1}+\delta_{i_2}+\delta_{i_3})}\right)^2}\right] ,$$

which is a significant perturbation when δ_{i_1}, δ_{i_2} and δ_{i_3} differ greatly. In practice we use up to six levels of virtual refinement whenever

$$\frac{\max\left(\delta_{i_1},\delta_{i_2},\delta_{i_3}\right)}{\min\left(\delta_{i_1},\delta_{i_2},\delta_{i_3}\right)} > \gamma ,$$

where the parameter γ is chosen, fairly cautiously, to be 1.2.

In [17] the reliability of these predictors is demonstrated by comparing the estimates with the number of nodes actually generated using the generation algorithm outlined below. The predicted values may now be used to give vertex and edge weights to the dual graph of the coarse background mesh in preparation for its partitioning across the processors.

2.2 Partitioning the Coarse Mesh

The approach that we use to partition the coarse mesh is based upon recursive bisection. This means that we need only consider the more straightforward problem of mesh bisection, so as to ensure load balancing and low communication. As has already been stated we choose to do this by bisecting the weighted dual graph of the background grid using weights based upon estimates of the number of nodes that will be generated in the final mesh.

The bisection technique that we use is described in [14] and is an extension of the popular spectral algorithm [28]. This involves expressing the bisection problem as a discrete constrained minimization:

$$\text{minimize } \frac{1}{4}\sum_{e\leftrightarrow f} w_{E_{ef}}(x_e - x_f)^2 \quad \text{subject to} \quad \sum_i x_i w_{N_i} = 0 ,$$

where $x_i = \pm 1$, corresponding to element i being in one or other subdomain,w_{N_i} are the nodal weights, $w_{E_{ef}}$ the edge weights and the first summation is over the edges of the dual graph whilst the second is over the V vertices. Defining the weighted Laplacian matrix L of the dual graph as

$$L_{ij} = \begin{cases} -w_{E_{ij}} & \text{if vertices } i \text{ \& } j \text{ are connected} \\ \sum_{k=1_{k \neq i}}^{V} w_{E_{ik}} & \text{if } i = j \end{cases}$$

the problem may be written in matrix terms as

$$\text{minimize } \frac{1}{4}\underline{x}^T L \underline{x} \quad \text{subject to} \quad \underline{x}^T \underline{w}_N = 0 \ , \ \ x_i = \pm 1 \ .$$

By introducing $\underline{y}$, $\underline{s}$ and D, where

$$y_i = x_i \sqrt{w_{N_i}} \ \ , \ s_i = \sqrt{w_{N_i}} \ \text{ and } \ D = diag\left(\frac{1}{\sqrt{w_{N_i}}}\right) \ , \tag{2.3}$$

it can be rewritten as

$$\text{minimize } \underline{y}^T S \underline{y} \quad \text{subject to} \quad \underline{y}^T \underline{s} = 0 \ , \ \ y_i = \pm\sqrt{w_{N_i}} \ , \tag{2.4}$$

where $S = D^T L D$, and neglecting the factor of one quarter.

Due to the complexity of this problem we seek an approximate solution by treating the x_i (and therefore the y_i) as continuous rather than discrete variables. Since, for a connected background grid, the matrices L and S are singular with a null space of dimension one, this involves using a Lanczos method (based upon that of [27]) to compute an eigenvector $\underline{u}_2$ which corresponds to the smallest non-zero eigenvalue λ_2 of S and which is orthogonal to the null space of S (see [14]). From (2.3) the partitioning vector $\underline{x}$ is then given by

$$x_i = \frac{u_{2_i}}{\sqrt{w_{N_i}}} \ .$$

We note that since the problem is solved using continuous variables, the property that $x_i = \pm 1$ ($y_i = \pm\sqrt{w_{N_i}}$) is lost and therefore the solution obtained will not be optimal. The partitioning is therefore made by sorting the V vertices of the dual graph according to the size of their entry in $\underline{x}$ and placing elements represented by $x'_i : i = 1 \ldots n$ in one subdomain (with $\underline{x}'$ being the sorted vector) and those by $x'_i : n+1 \ldots V$ in the other, with n chosen so that

$$\left| \sum_{i=1}^{n} w'_{N_i} - \sum_{i=n+1}^{V} w'_{N_i} \right|$$

(where w'_{N_i} is the weight of the node represented by x'_i) is as small as possible. The effect of this is to produce a decomposition of the mesh into two subdomains of approximately equal weight which has a relatively small number of vertices lying on the boundary between the two subdomains.

Recursive application of this bisection technique on each new subdomain allows the coarse mesh to be divided into more that just two pieces (although the number does have to be a power of two).

The partitioning technique used by Löhner *et al* in [24] is a little more general than this, allowing an arbitrary number of subdomains, however it fails to take into account the number of nodes that are going to be created along the boundaries between subdomains – unlike the spectral approach. Also, because we are using the spectral algorithm only on the coarse background grid, our code does not suffer from the high computational overheads often associated with the spectral method on large meshes. Consequently there is no urgent need to implement this part of the generator in parallel for the time being, although the work of Leete *et al* ([23]) shows this to be possible (in the computational examples described in subsection 2.5 the sequential recursive bisection typically accounts for about 15% of the total generation time). Other improvements to the implementation of the spectral technique, using a multi-level approach, are described in [4] and [15].

2.3 Parallel Mesh Generation

Having partitioned the coarse background mesh across the distributed memory of the parallel processors it now remains to generate the finite element mesh. The main difficulty with this is one of ensuring conformity of the mesh between subregions. Note that we choose to treat each coarse background element as a separate subregion to be meshed independently and so each processor will typically have a number of subregions to deal with. (Alternatively, it is possible to treat the union of the coarse elements on each processor as a single region for meshing purposes but we choose not to do this due to the nature of the finite element solution algorithm that is described in section 3.)

The conformity problem turns out to be surprisingly easy to overcome. Firstly we insist that on each coarse edge, E_{ij}, the number of nodes generated should be exactly the number $w_{E_{ij}}$, given by (2.1), with the positions of these nodes given by an unambiguous geometric rule (see [17] for details). We then use a two-dimensional version of Weatherill's algorithm ([32]) for generating a Delaunay mesh on the interior of each subdomain (coarse element). This algorithm has the important property that no additional nodes are ever added along the boundary of the region being meshed. Hence neighbouring subdomains will always have meshes which match up along their interface: not only in the number of nodes but in their precise location too.

As well as inserting no new points into boundaries, an additional positive feature of Weatherill's algorithm is that in theory the run time only grows linearly with the number of points created. Hence if two processors with a very different number of coarse elements generate about the same number of nodes in total, they should take approximately the same execution time. This means that the actual mesh generation process should itself be well load balanced.

2.4 Parallel Implementation Issues

The requirement of keeping a consistent join between the different subregions which are meshed independently has been discussed in the previous subsection. When an edge of the coarse mesh lies on the partition between two different processors there is no difficulty in ensuring that both processors place identical nodes along this edge, without the need for any communication between them. It is not even necessary that these nodes have the same numbering on the different processors. The fine grid vertices are in fact numbered locally to the processor on which they are created. A local numbering scheme for the fine grid elements and vertices within each coarse element is adopted. In addition to this, the coarse mesh vertices and edges are considered separately and the fine grid nodes along each coarse edge numbered locally.

The local numbering scheme does not pose any problems since for a typical parallel finite element solver (see for example [16]) the global numbering of vertices and elements is not required, however, a data structure must be built which gives information concerning data transfer across interpartition boundaries. Here, each processor needs a list of local vertices which are shared with neighbouring subdomains. For the correct updating of values associated with interpartition boundary vertices, two processors with a common subdomain boundary must have the shared vertices listed in the same order. With the parallel mesher two such data structures are built; one for the transfer of coarse interpartition boundary vertices (IBVs) and one for the coarse interpartition boundary edges (IBEs). This saves unnecessary waste by not explicitly listing all the vertices along the interpartition boundary, i.e. those along the inside of an IBE are implicitly defined.

The data transfer structure for the coarse IBVs is constructed on the master processor after the partitioning of the coarse mesh has taken place. The relevant transfer information for each slave processor is then sent by the master processor along with the coarse subdomain data required for the meshing. To achieve consistent node orderings the coarse grid vertices on each processor are listed in the same order as the global node numberings and in the coarse IBV transfer array the vertices shared with a particular neighbour are listed in ascending order.

The corresponding structure for the IBEs is built in a similar fashion but in parallel once the coarse grid data has been farmed out. Each processor lists the edges it has in common with each of its neighbours and for consistency, the IBEs a processor shares with another are listed in the same order as the global edge numbering. The global numbering of edges is performed beforehand on the master processor along with the coarse transfer data. Since the local edges are not necessarily numbered in the same order as the global edge numbering scheme, the list of common IBEs is not always in ascending order (as is the case with the IBVs). By ensuring that the local numbering of unknowns along each edge is performed in a consistent manner, i.e. in ascending order from the end which has the greater coarse grid global node number, the order of unknowns along an interpartition boundary is consistent for the two processors in question.

This meshing strategy is sufficiently unambiguous at processor interfaces to allow the straightforward implementation of a parallel finite element algorithm yet it also allows the mesh generation to proceed in parallel without the need for any data communication once the background elements have been partitioned and distributed. The only potential bottleneck in the algorithm therefore is the generation and partitioning of this coarse mesh. However, as indicated in the previous subsection, this turns out to be an acceptable cost in practice.

The final stage of almost any mesh generation algorithm is that of mesh smoothing. For example, a possible global approach to this would be to make use of a Laplacian filter (as in [33]) to alter the position of each grid point, $\underline{v}_0$ say, internal to the entire domain:

$$\underline{v}_0^{n+1} = \underline{v}_0^n + \frac{\omega}{m} \sum_{i=1}^{m} (\underline{v}_i^n - \underline{v}_0^n) \ ,$$

where m is the number of neighbouring vertices and ω is a relaxation parameter. In order to perform this smoothing calculation for nodes on subdomain boundaries it would be necessary to undertake a significant amount of interprocessor communication (when the coarse edges are on a partition between two processors). Moreover, the fine mesh would no longer be an exact refinement of the background mesh: a property that will be of use to us in the next section. For these reasons we only apply the smoothing operation to those nodes in the interior of each subdomain. Assuming that the initial coarse grid is itself appropriately smoothed it is expected that this will be adequate.

2.5 Examples

To illustrate the performance of this mesh generator we choose a highly irregular domain and generate a number of meshes on it, using up to 32 nodes of the Intel iPSC/860. The domain we have selected is taken from Bank's PLTMG users' guide ([3]), where it is referred to as the Texas geometry. In each case the meshes generated are of a uniform density and the coarse background grid used comprises of 1331 elements (which takes about 1.2 seconds to generate on a single processor). An example coarse mesh is shown in Figure 1. Table 1 shows some computational results: giving the total number of elements generated, the average number of vertices created on each processor, the maximum percentage nodal imbalance on any processor and the time taken for generation. An example fine mesh, partitioned into four subdomains, is shown in Figure 2 (note that Figures 1 and 2 are illustrative only and do not correspond to the much larger meshes quoted in Table 1). These results show that the *a priori* mesh partitionings have resulted in meshes that are reasonably well load balanced. There is a small imbalance in each example however: with a maximum imbalance in the number of vertices of 5.2% when using 16 processors for example. This is due both to small errors in the estimates of the number of nodes generated in each coarse element and to the slight imperfections in the weighted graph partitioning algorithm that we used. It can also be seen that the generation time increases as larger meshes are generated even though the average number of vertices being created on each processor is roughly equal. This is not due to scalability factors since no communication takes place but rather the nonlinear growth in runtime of our mesh generator with respect to the number of points it creates in each coarse element as can be seen in Figure 3 (see also [17]). This figure shows that, with our particular implementation of the sequential mesh generator, [32], it is faster to generate $\frac{\nu}{10}$ vertices on 10 background elements (say) in turn than to generate ν vertices on a single background element for example.

In [17] the mesh generator has been shown to perform equally well on complex domains where local mesh refinement has been employed. It is also demonstrated that, in terms of minimizing communication costs, the decompositions produced by the parallel mesher are competitive with partitions generated by other methods (e.g. Recursive Spectral Bisection and Recursive Graph Bisection [28]) when applied to the entire triangulation.

No. of Procs.	Total No. of Fine Elements	Average No. of Nodes per Subdomain	Max Positive Percentage Difference from Average	Generation Time (seconds)
2	50153	12877	1.1	16.4
4	111887	14341	0.8	18.1
8	223052	14306	2.1	19.2
16	446151	14347	5.2	22.4
32	827733	13318	3.8	26.0

Table 1: Mesh generation results.

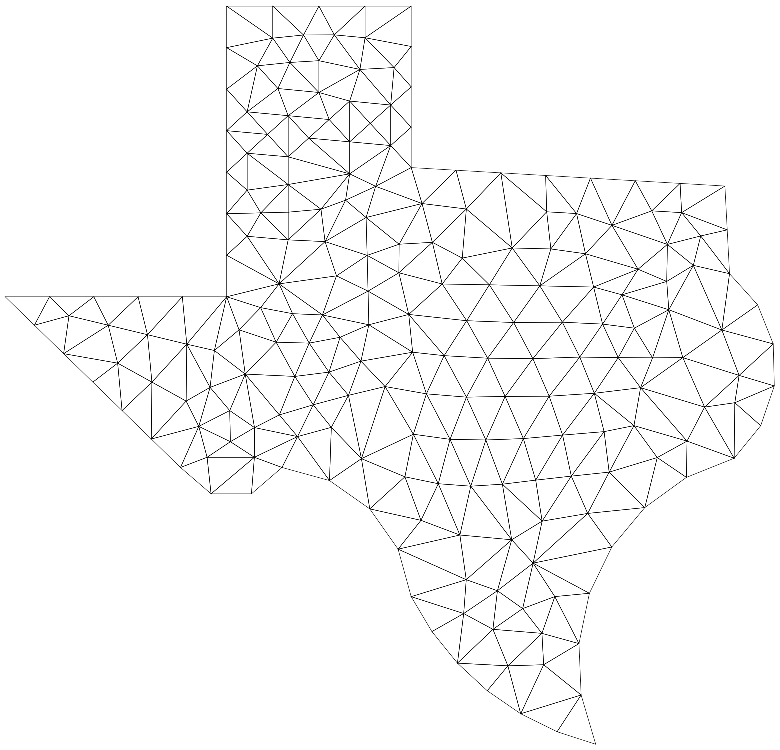

Figure 1: Coarse Texas mesh.

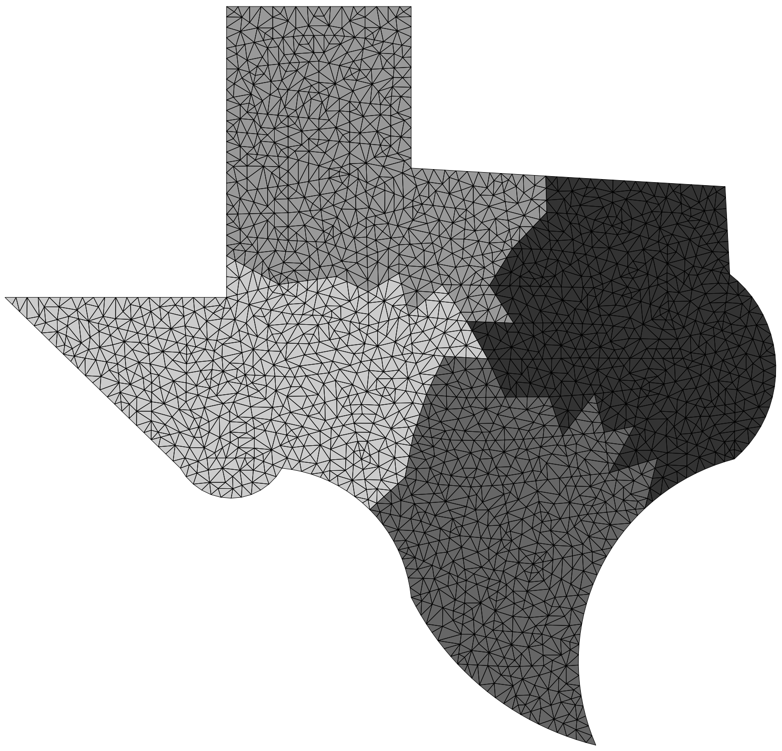

Figure 2: Fine Texas mesh.

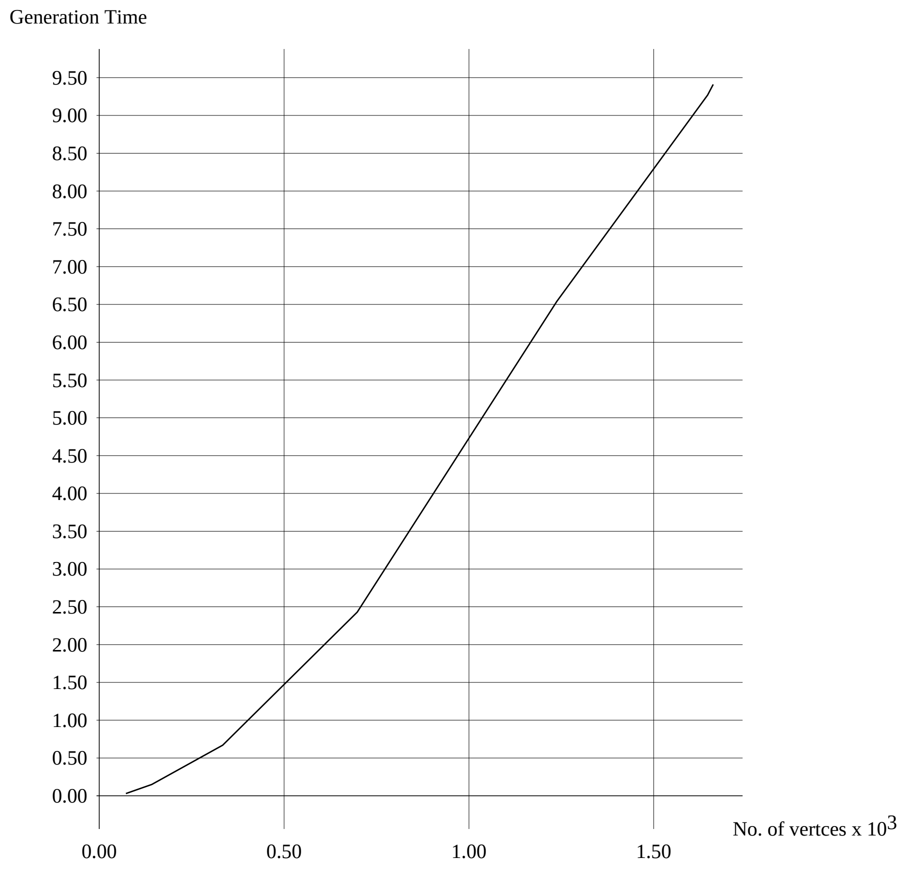

Figure 3: Growth in run time with respect to the number of vertices generated.

3 A Parallel Domain Decomposition Preconditioner

We now consider solving a general second order elliptic p.d.e. whose weak solution, $u \in H^1_E(\Omega)$, satisfies a relationship of the form

$$a(u,v) = (f,v) \quad \forall v \in H^1_0(\Omega)\ , \tag{3.1}$$

for some $\Omega \in \Re^2$, where the bilinear form $a(\cdot,\cdot)$ is symmetric and elliptic, $(\cdot,\cdot)$ represents the usual L^2-inner product, and we will assume for simplicity that u satisfies the Dirichlet condition $u = u_E$ on the domain boundary, $\partial\Omega$. Once a partitioned triangulation of the problem domain, Ω, has been constructed using the approach described in section 2, it is possible to form a finite element discretization of (3.1) on this mesh. For the purposes of this paper we will only consider a piecewise linear discretization although the ideas presented here do extend to higher order elements, through the use of hierarchical basis functions for example (see [1] and references therein). This will lead to the necessity to solve one or more sparse linear systems of the form

$$K\underline{x} = \underline{b} \; , \tag{3.2}$$

where K is a symmetric positive definite stiffness (or Jacobian) matrix, $\underline{b}$ is a known vector and $\underline{x}$ is a vector of unknowns which correspond to values at the nodes of the finite element mesh (see [20] for example).

Using piecewise linear elements, if we order the nodes of the finite element mesh starting with all of the nodes generated inside the first coarse element, followed by all of the nodes inside the second coarse element, up to all of the nodes inside the last coarse element, followed by all of the nodes on the boundary between coarse elements (this excludes those on the Dirichlet boundary), then the resulting finite element system, (3.2), takes the block-arrowhead form

$$\begin{bmatrix} A_{II_{(1)}} & & & & A_{IB_{(1)}} \\ & A_{II_{(2)}} & & & A_{IB_{(2)}} \\ & & \ddots & & \vdots \\ & & & A_{II_{(N)}} & A_{IB_{(N)}} \\ A^T_{IB_{(1)}} & A^T_{IB_{(2)}} & \cdots & A^T_{IB_{(N)}} & A_{BB} \end{bmatrix} \begin{bmatrix} \underline{x}_{I_{(1)}} \\ \underline{x}_{I_{(2)}} \\ \vdots \\ \underline{x}_{I_{(N)}} \\ \underline{x}_B \end{bmatrix} = \begin{bmatrix} \underline{b}_{I_{(1)}} \\ \underline{b}_{I_{(2)}} \\ \vdots \\ \underline{b}_{I_{(N)}} \\ \underline{b}_B \end{bmatrix} . \tag{3.3}$$

Here it is assumed that there are N subdomains and that $\underline{x}_{I_{(i)}}$ represents those unknowns inside subdomain i and $\underline{x}_B$ represents those unknowns on the boundary between subdomains (i.e. the boundary between coarse elements). Moreover, it is possible to compute and assemble each of the blocks $A_{II_{(i)}}$, $A_{IB_{(i)}}$ and $\underline{b}_{I_{(i)}}$ independently on the processor storing subdomain i. In addition, each processor can compute and assemble its own contributions to the remaining block of K, A_{BB}, independently – storing them in the blocks $A_{BB_{(i)}}$ say (hence $A_{BB} = A_{BB_{(1)}} + ... + A_{BB_{(N)}}$). For the remaining block of $\underline{b}$, $\underline{b}_B$, each processor can again compute its own contributions but some communication is required to enable the full $\underline{b}_B$ to be known on each processor.

A common approach in dealing with systems of the form (3.3) is that of the Schur Complement Method (SCM). Here it is noted that

$$A_{II_{(i)}}\underline{x}_{I_{(i)}} + A_{IB_{(i)}}\underline{x}_B = \underline{b}_{I_{(i)}} \qquad \text{for } i = 1, ..., N \tag{3.4}$$

and

$$\sum_{i=1}^{N} A^T_{IB_{(i)}}\underline{x}_{I_{(i)}} + A_{BB}\underline{x}_B = \underline{b}_B \; ,$$

hence

$$(A_{BB} - \sum_{i=1}^{N} A^T_{IB_{(i)}} A^{-1}_{II_{(i)}} A_{IB_{(i)}})\underline{x}_B = \underline{b}_B - \sum_{i=1}^{N} A^T_{IB_{(i)}} A^{-1}_{II_{(i)}} \underline{b}_{I_{(i)}} \; .$$

This means that we may first solve the problem

$$S\underline{x}_B = \underline{g} \tag{3.5}$$

for the unknowns on the subdomain boundaries (where

$$S = A_{BB} - \sum_{i=1}^{N} A_{IB_{(i)}}^T A_{II_{(i)}}^{-1} A_{IB_{(i)}} \tag{3.6}$$

and

$$\underline{g} = \underline{b}_B - \sum_{i=1}^{N} A_{IB_{(i)}}^T A_{II_{(i)}}^{-1} \underline{b}_{I_{(i)}} \tag{3.7}$$

in (3.5)), then we can solve each of the subdomain problems of (3.4) for the rest of the unknowns: $\underline{x}_{I_{(i)}}$, for $i = 1, ..., N$. Such an approach is ideal for distributed memory parallel computing because each system in (3.4) is entirely independent and may therefore be solved in parallel with the others when required. Moreover, if an iterative method, such as the Conjugate Gradient (CG) algorithm (see [11]), is used to solve (3.5) then it is not usually necessary to explicitly form the matrix S of (3.6). This is very important because it would be extremely computationally expensive to construct the matrix S, however the CG algorithm, for example, only needs to know the product of S with given direction vectors, $\underline{v}_B$ say. From (3.6) we see that

$$S\underline{v}_B = \sum_{i=1}^{N} A_{BB_{(i)}}\underline{v}_B - \sum_{i=1}^{N} A_{IB_{(i)}}^T \left(A_{II_{(i)}}^{-1} \left(A_{IB_{(i)}}\underline{v}_B \right) \right) , \tag{3.8}$$

and may therefore be obtained using only matrix-vector multiplications and subdomain solves (some local communication is also required between processors sharing interpartition boundary vertices). Each of these operations may also be carried out in parallel on each subdomain if necessary.

An advantage of using the SCM is that it yields a smaller system to solve than the original one. In fact, the fewer the number of subdomains, the shorter one expects the boundary between them to be and so the smaller the Schur Complement matrix S will become. This must however be balanced against the fact that at each CG iteration it is necessary to perform an accurate solve on each subdomain (due to (3.8)) and these solves will of course get larger as the number of subdomains get fewer. On structured problems it is often possible to make use of fast subdomain solvers as described in [6], thus leading authors to tend to be less concerned about the size of the subdomain problems than about the size of the Schur Complement system. This in turn can lead to the habit of letting the number of subdomains be equal to the number of processors that are being used to solve the problem. In this work we do not do this, preferring instead to try and keep the number of subdomains independent of the number of processors as far as possible. This is achieved by treating each coarse element in our background grid as a separate subdomain rather than combining all of those coarse elements which are assigned to the same processor into a single subdomain.

As with the original finite element matrix, K, in (3.2) and (3.3), the condition number of the Schur Complement matrix S also depends upon the finite element mesh from which it is ultimately derived. The precise nature of this dependence is hard to quantify but from numerical experiments it can be seen that the condition number grows at a rate of $O(h^{-1})$ as a finite element mesh (of grid size h) is uniformly refined (see e.g. [5]). For this reason it is essential to find a good preconditioner for the linear system (3.5) if we wish to obtain a reasonable convergence rate when solving the problem on a finite element mesh with a very large number of degrees of freedom. In this section we extend the preconditioner analyzed by Dryja and Widlund in [7], to deal with such a system on an unstructured mesh. This extension is only possible because of the presence of two levels to our unstructured grid, with the fine mesh being embedded within the original coarse mesh that was used for its parallel generation.

The principle behind the preconditioner considered in [7] is that it approximates the Schur complement matrix S in both a local and a global manner, and in a way which makes it easy to invert. This is achieved by the use of a coarse substructure on which the problem may be solved cheaply to remove low frequency errors, and through the solution of independent local problems for the unknowns on each edge of the substructure to eliminate high frequency errors. Subsection 3.1 describes how this approach is developed in the context of our nested unstructured meshes. The rest of this section considers the performance of this preconditioner and issues relating to its parallel implementation. It may be observed that without the coarse substructure solves the parallel implementation would be a lot more straightforward however it is also demonstrated that such a preconditioner is not as effective. An asymptotic result from [7] is also quoted to indicate why the full preconditioner performs so well.

3.1 The Preconditioner

We wish to precondition the linear system (3.5) where the unknowns $\underline{x}_B$ represent nodal values on the boundary between subdomains (coarse elements). Inspection of Figure 4 shows that these unknowns may be divided into two categories: those at the knot points of the coarse mesh and those lying within an edge of the coarse mesh. If we order these unknowns with the coarse mesh knot points

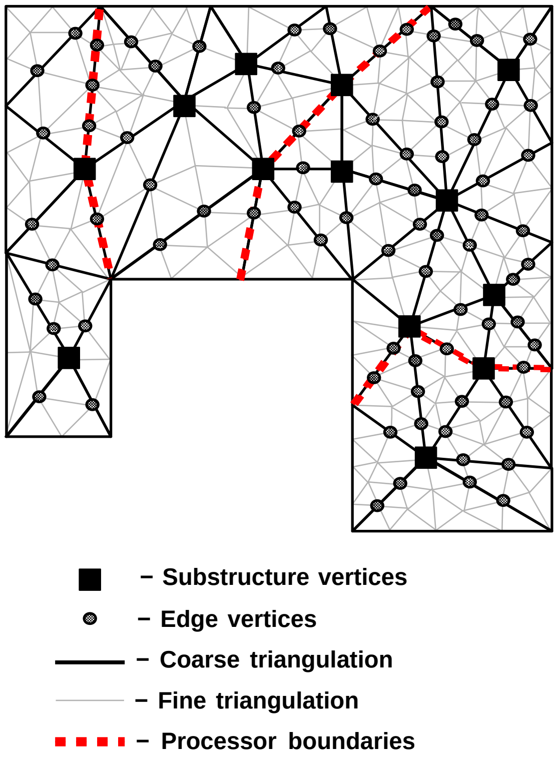

Figure 4: Two-level unstructured mesh, distinguishing node categories.

last then we may write the Schur Complement system (3.5) in the block matrix form

$$\begin{bmatrix} S_{EE} & S_{EV} \\ S_{EV}^T & S_{VV} \end{bmatrix} \begin{bmatrix} \underline{x}_E \\ \underline{x}_V \end{bmatrix} = \begin{bmatrix} \underline{g}_E \\ \underline{g}_V \end{bmatrix} . \tag{3.9}$$

Here $\underline{x}_E$ is a vector of the unknowns within the coarse mesh edges and $\underline{x}_V$ is a vector of the unknowns at the coarse mesh knot points. The remaining blocks represent the corresponding splitting of S (defined by (3.6)) and $\underline{g}$ (defined by (3.7)) subject to this ordering. From (3.6) we can see that

$$S_{EE} = A_{EE} - A_{IE}^T A_{II}^{-1} A_{IE} \tag{3.10}$$

and

$$S_{VV} = A_{VV} - A_{IV}^T A_{II}^{-1} A_{IV} , \tag{3.11}$$

where A_{II} is the block diagonal matrix given by

$$A_{II} = \begin{bmatrix} A_{II_{(1)}} & & & \\ & A_{II_{(2)}} & & \\ & & \ddots & \\ & & & A_{II_{(N)}} \end{bmatrix}$$

and the system (3.3) has been expressed as

$$\begin{bmatrix} A_{II} & A_{IE} & A_{IV} \\ A_{IE}^T & A_{EE} & A_{EV} \\ A_{IV}^T & A_{EV}^T & A_{VV} \end{bmatrix} \begin{bmatrix} \underline{x}_I \\ \underline{x}_E \\ \underline{x}_V \end{bmatrix} = \begin{bmatrix} \underline{b}_I \\ \underline{b}_E \\ \underline{b}_V \end{bmatrix} , \tag{3.12}$$

using the same ordering of the nodes on the boundary between subdomains as above.

We may further define our ordering of the unknowns on the boundary between subdomains by requiring that the nodes lying on each individual edge of the coarse mesh be numbered contiguously. Hence we may express $\underline{x}_E$, for example, as

$$\underline{x}_E = (\underline{x}_{E_{(1)}}^T, \underline{x}_{E_{(2)}}^T, \ldots \underline{x}_{E_{(L)}}^T)^T ,$$

where $\underline{x}_{E_{(i)}}$ is a vector of those unknowns lying within the i^{th} edge of the coarse mesh (for $i = 1, \ldots, L$ say). With this ordering we can now derive a simple preconditioner for the system (3.9):

$$M = \begin{bmatrix} \tilde{S}_{EE} & \\ & \tilde{S}_{VV} \end{bmatrix} . \tag{3.13}$$

Here $\tilde{S}_{EE}$ is the block diagonal matrix of $A_{EE} - A_{IE}^T A_{II}^{-1} A_{IE}$ which has a block $(S_{EE_{(i)}})$ for each edge i of the coarse mesh, and $\tilde{S}_{VV}$ is the diagonal of $A_{VV} - A_{IV}^T A_{II}^{-1} A_{IV}$. Hence applying this preconditioner within the context of the preconditioned Conjugate Gradient algorithm requires solution of a system such as

$$M\underline{z} = \underline{r}$$

at each iteration. From the definition (3.13) of M this may be expressed as

$$\begin{aligned} \underline{z} &= \begin{bmatrix} \tilde{S}_{EE}^{-1} & \\ & \tilde{S}_{VV}^{-1} \end{bmatrix} \underline{r} \\ &= \sum_{i=1}^{L} R_{E_{(i)}}^T S_{EE_{(i)}}^{-1} R_{E_{(i)}} \underline{r} + R_V^T \tilde{S}_{VV}^{-1} R_V \underline{r} \end{aligned} \tag{3.14}$$

say, where the rectangular matrices $R_{E_{(i)}}$ and R_V act as restriction operators which return only those coefficients associated with the i^{th} coarse edge and the coarse mesh knot points respectively.

The block diagonal preconditioning given by (3.14) is clearly straightforward to implement. Unfortunately, when the number of subdomains is quite large its performance is not too good due to the individual edge problems, involving $S_{EE_{(i)}}^{-1}$, becoming very small. Since we intend using the elements of our coarse background mesh as our subdomains this is likely to occur quite frequently and so a better preconditioner is required.

Utilizing the ideas presented for structured mesh preconditioning in [7, 29] we may now make use of the coarse background mesh to introduce a global contribution to our preconditioner. This can be achieved by interpolating from the interface unknowns onto the unknowns at the knot points of the coarse mesh, then solving a coarse mesh system and interpolating back onto the substructure edges. To implement this it is necessary to alter the preconditioning step in the preconditioned CG algorithm from (3.14) to

$$\underline{z} = \sum_{i=1}^{L} R^T_{E_{(i)}} S^{-1}_{EE_{(i)}} R_{E_{(i)}} \underline{r} + \tilde{R}^T_V K_V^{-1} \tilde{R}_V \underline{r} \, . \tag{3.15}$$

Here $\tilde{R}^T_V$ interpolates linearly from the coarse knot points onto the coarse mesh edges and the matrix K_V is the stiffness matrix (corresponding to the bilinear form $a(\cdot,\cdot)$ in (3.1)) on the coarse mesh.

In [7] it is demonstrated that for a solution on regular meshes the preconditioner from which (3.15) has been generalized causes the condition number of the preconditioned Schur system to grow as $(1+\log(\frac{H}{h}))^2$ as $h \to 0$ (where H indicates the size of the coarse substructure). We expect equally good performance in the case of this preconditioner. The following subsection investigates this with the aid of some computational examples.

3.2 The Effectiveness of the Preconditioner

The performance of the preconditioner is now considered by solving the Poisson equation

$$-\Delta u = 1 \quad \text{in the domain } \Omega,$$

subject to the Dirichlet boundary condition $u\,|_{\partial\Omega} = 0$, on the meshes which were generated in order to provide the results given in subsection 2.5. In each case the conjugate gradient solver was run until a relative residual of 10^{-8} was achieved. Table 2 shows the number of iterations required to achieve convergence using the Schur complement method with and without domain decomposition preconditioning (SCM-P1 indicates use of the purely local preconditioner given by (3.14) and SCM-P2 corresponds to the preconditioner given by (3.15) which includes the coarse mesh solve). As can be seen, the full preconditioner (SCM-P2) substantially reduces the number of iterations required for each problem and also causes a slower growth in the number of iterations as the mesh is refined. The local preconditioner (SCM-P1), whilst being of some benefit, is significantly less effective than this and so is not considered any further. Timing and efficiency details for our parallel implementation of this solver appear in table 4.

No. of	No. of Schur	No. of Iterations		
Elements	Unknowns	SCM	SCM-P1	SCM-P2
23819	8868	330	219	49
50153	14649	430	243	57
111887	23118	542	263	63
223052	33335	665	277	69
446151	49725	805	297	74
827733	67630	948	308	80

Table 2: Iteration counts.

3.3 Parallel Implementation Issues

Having established the effectiveness of the proposed domain decomposition preconditioner it is necessary to demonstrate that it may be implemented in parallel in an efficient manner. Inspection of (3.15) shows that the preconditioning step requires a number of small independent subproblems to be solved. These consist of an edge problem of the form $R^T_{E_{(i)}} S^{-1}_{EE_{(i)}} R_{E_{(i)}} \underline{r}$ for each edge i (from 1 to L) of the coarse mesh, and a global problem on the coarse mesh itself: $\tilde{R}^T_V K^{-1}_V \tilde{R}_V \underline{r}$. Below we show how each of the edge problems may be constructed and solved on the individual processor which stores that coarse edge, with all communication between pairs of processors which may share a common coarse edge being carried out in parallel with other computations (i.e. with hidden communication). A serial algorithm is then described for solving the global substructure problem which, it is shown, may be implemented in parallel with the coarse edge problems in such a way as to minimize this serial overhead.

Edge Solvers

These edge problems involve not only the values along the particular edge but also contributions from the two subdomains adjoining it. If l_i and r_i denote the number of the subdomain to the "left" and "right" respectively of edge i, the Schur submatrix for an edge i is given by

$$S_{EE_{(i)}} = A_{EE_{(i)}} - A^T_{I_{(l_i)}E_{(i)}} A^{-1}_{II_{(l_i)}} A_{I_{(l_i)}E_{(i)}} - A^T_{I_{(r_i)}E_{(i)}} A^{-1}_{II_{(r_i)}} A_{I_{(r_i)}E_{(i)}} \quad (3.16)$$

(generalizing the notation introduced in (3.12)). The $S_{EE_{(i)}}$ are symmetric positive definite but are dense so each edge solve requires an application of the standard Choleski decomposition method. Although this may appear to be quite computationally intensive the edge problems are considerably smaller than the sparse interior problems which must also be solved at each iteration and experiments show that the relative computation costs between the edge and subdomain operations are quite similar, so the dense factorization of the edge problems is not detrimental.

Since the full Schur Complement matrix (3.6) is never actually formed, the Schur edge blocks, $S_{EE_{(i)}}$, require composing before they can be factorized. This too is quite expensive since, as seen from (3.16), a number of solves over the two subdomains on either side of the edge must be performed: the actual number being equal to the number of nodes along the edge. (Despite this high computational cost an efficient implementation is still far superior to using an iterative technique for these dense edge systems.)

For an edge lying on a partition between two different processors, contributions towards the corresponding Schur edge matrix are required from the two processors responsible for the adjacent coarse elements. This necessitates the transmission of partially assembled Schur edge matrices between processors. Since these matrices are dense (but symmetric), the full lower triangle must be sent, which makes this communication very expensive. However, it is possible to hide this cost by overlapping the communication with computation, using the algorithm presented in table 3. At step (3) the matrices are transmitted using asynchronous sends which return immediately and do not wait for the message to be received by the destination processor. This allows further computation to be performed at stage (4) whilst the communication is taking place. Because the computation of step (4) is quite intensive, the expectation is that no idle waiting will be required at step (5) and thus the communication is totally hidden.

Coarse Grid Solve

The other term in the preconditioning step of (3.15) is a global coupling term requiring the use of the interpolation operator $\tilde{R}_V^T$. This is very cheap to implement since we already know the required distances from each edge node to the coarse mesh points from when the mesh was originally generated (see subsection 2.3). All that is now required is to solve the coarse mesh problem, involving the matrix K_V.

Since the global coupling term is independent of all of the edge subproblems it may be solved in parallel with them. This is achieved by identifying which of the processors has the least amount of work assigned to it (recall that the mesh generator aims to give an equal amount of work to each processor but, as illustrated in subsection 2.5, does not always achieve this perfectly), and performing the coarse grid solve serially on this processor. There are of course some communication overheads associated with assembling this problem onto one processor but these too can be overlapped with the

```
(1)  Post asynchronous receives for non-local contributions to those Schur edge
     matrices which lie on an interface between processors
(2)  Form local contributions to these Schur edge matrices
(3)  Send these partially formed Schur matrices to neighbouring processors
(4)  Form and factor Schur matrices for edges internal to each processor's
     partition
(5)  Wait for an asynchronous receive to be satisfied (if necessary)
     • Sum local and non-local contributions of received Schur edge matrix
     • Factor this Schur edge matrix
     • Repeat (5) until all edges shared with another processor are accounted for
```

Table 3: Algorithm for forming and factorising Schur edge matrices

solution of the Schur edge problems so as to hide the costs.

In Table 4 we give the corresponding parallel run times for the problems presented in subsection 3.2. The timings are separated into the time required for calculating the components of the preconditioner (startup time) and the time required for the (preconditioned) conjugate gradient iterations to converge (iteration time). The startup costs for the preconditioned solve can be seen to be slightly higher than for the unconditioned solve because of the additional work required in forming and factoring the Schur edge matrices.

3.4 Scalability

We examine the parallel performance of the preconditioned solver by the use of the fixed time scalability measure ([30]). This is favoured over the usual fixed size scalability analysis since the problem size is no longer restricted to that which can be solved on one processor. As the number of processors is increased we allow the size of problem to also grow and thus can gather results for more natural problem sizes.

For our analysis the problem size on one processor was taken to be the largest possible that could be dealt with using primary memory. The time taken to perform 60 iterations was then taken as the fixed time for the further tests. The coarse and fine mesh sizes were increased with the number

No. of Procs.	Total No. of Fine Elements	Solution Method	Startup Time	Iteration Time	Total Time
2	50153	SCM	0.4	92.9	93.3
		SCM-P2	2.3	21.1	23.4
4	111887	SCM	0.8	129.2	130.1
		SCM-P2	4.3	23.6	27.9
8	223052	SCM	1.4	171.1	172.5
		SCM-P2	6.3	25.2	31.5
16	446151	SCM	2.0	214.3	216.3
		SCM-P2	10.0	27.2	37.3
32	827733	SCM	2.7	261.4	264.1
		SCM-P2	13.5	29.5	43.1

Table 4: Parallel timings.

of processors, keeping the ratio of fine to coarse grid elements reasonably constant, such that the problem was always solved in the same fixed time (for 60 iterations) and the MFLOP count taken. The growth in the MFLOP rate when solving on increasing numbers of processors thus gives the scalability of the solver. Rather than keep a fixed coarse mesh, we choose to increase the number of coarse grid elements as the number of processors used is incremented since it is apparent that for the mesh generation this must be the case. Also, by maintaining the fine:coarse element ratio we minimize any scalability abnormalities due to the differing MFLOP rates of various operations (see [13] for an explanation of the consequences of using the fixed time metric).

Table 5 gives the results of our fixed time scalability analysis for the parallel preconditioned solver running on the iPSC/860. We have used the same geometry as in subsection 2.5, again applying uniform refinement. Since the data partitioning is not optimal, the decomposition of the mesh is a factor in the parallel scaling. Hence *this is not a measure of the scalability of the solver itself* but a more realistic practical indication of the scalability of the overall algorithm. The efficiencies are quite respectable but could be improved by employing a better load balancing strategy for the coarse solve as discussed in the next section.

No. of Procs.	Coarse Elements	Fine Elements	MFLOP Rate	Fixed Time Speedup	Fixed Time Efficiency
1	203	23251	1.51	1.0	100%
2	423	47647	2.74	1.8	92%
4	835	95248	5.42	3.5	89%
8	1331	150050	10.15	6.7	84%
16	2244	252307	18.25	12.1	76%
32	4556	510361	31.25	20.8	65%

Table 5: Scalability results.

4 Discussion

4.1 A Complete Numerical Example

In sections 2 and 3 we showed the mesh generator and parallel preconditioned domain decomposition solver applied to a non-trivial domain geometry. Here we take a simple square domain but solve a problem whose solution has regions of steep gradient and thus requires a computational mesh which contains heavy local refinement.

The problem considered is

$$-\Delta u = f \quad \text{on the domain } \Omega = (0,10) \times (0,10),$$

where f is taken so as to permit the unique solution

$$(\tanh(100x + 100) - \tanh(100x - 800))(\tanh(100y + 100) - \tanh(100y - 700)) \tag{4.1}$$

and the Dirichlet boundary conditions are taken from the above solution. From (4.1) we know where the solution changes rapidly and can therefore set the point distribution values for the parallel mesher accordingly. (In a more practical problem we would not expect to know the solution in advance and so the point distribution values could not be chosen so well *a priori*. This issue is addressed in [18], where it is demonstrated that one may first select uniform distribution values throughout the coarse grid points and then use an *a posteriori* error estimate (based upon the solution found on the resulting uniform mesh) to assign more appropriate, non-uniform, point distribution values.)

The coarse mesh and an example fine mesh are shown in Figures 5 and 6 respectively. Results for the mesh generation on 16 processors are given in Table 6. Although the mesh generation involves creating regions which are highly refined, the mesh decomposition has resulted in equally good load balancing as for the uniform refinement examples (Table 1).

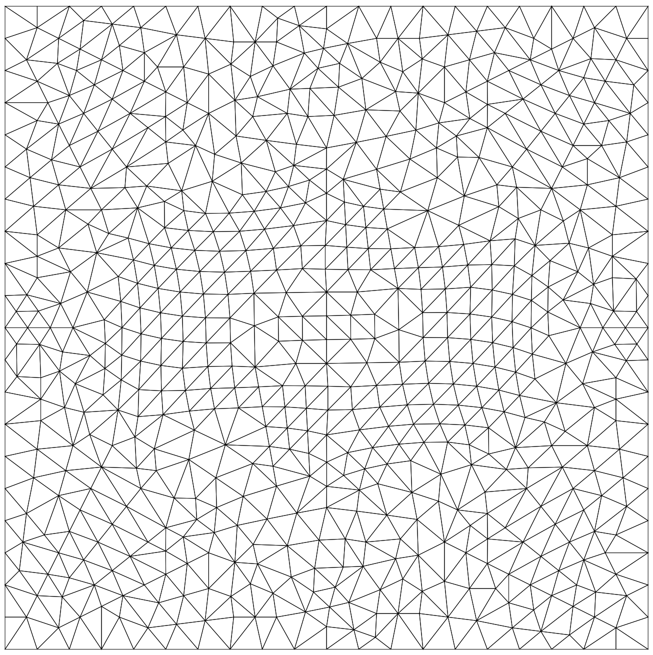
Figure 5: Coarse square mesh.

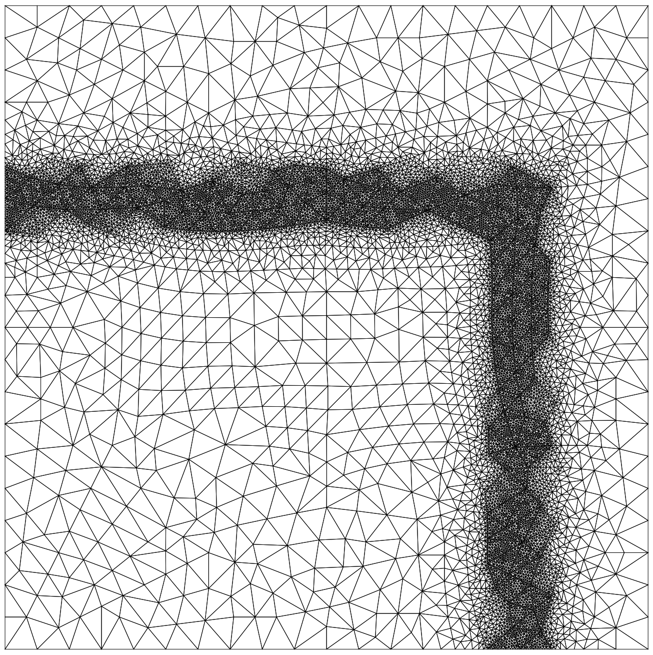
Figure 6: Fine square mesh.

Coarse mesh : 1260 elements Final mesh : 383336 elements							
Subdomain Number	No. of Nodes	Diff. From Average No.	Meshing Time	Subdomain Number	No. of Nodes	Diff. From Average No.	Meshing Time
1	12192	-0.6%	23.1	2	12859	+4.8%	26.7
3	12054	-1.8%	29.1	4	12280	+0.1%	24.9
5	12602	+2.7%	26.5	6	12407	+1.1%	28.4
7	11756	-4.2%	21.3	8	12265	+0.0%	22.9
9	12364	+0.8%	27.4	10	12283	+0.1%	25.4
11	12607	+2.8%	28.5	12	11964	-2.5%	25.3
13	12553	+2.3%	29.5	14	12002	-2.2%	24.3
15	12196	-0.6%	23.3	16	11918	-2.9%	22.0

Table 6: Parallel mesh generation results.

A comparison of the pre- and un-conditioned solves for this problem is given in Table 7. Although the initialization costs for the preconditioned solve (SCM-P2) appear quite high, it must be noted that this time is independent of the p.d.e. being solved and hence for more ill-conditioned problems it will not be so dominant. In such cases we would expect the preconditioned solve to achieve a greater speedup over the standard SCM.

Size of Schur System: 26784 unknowns				
Method	No. Its.	Startup Time	Iteration Time	Total Time
SCM	728	3.3	208.3	211.6
SCM-P2	78	18.9	28.1	47.0

Table 7: Performance of parallel preconditioner.

From theory on locally refined structured grids (e.g. see [1]) it is known that the condition number of the preconditioned system is governed by the worst case subdomain, i.e. the most refined coarse grid element in our context. In order to stop one subdomain from causing ill-conditioning, we could therefore choose to adapt the background mesh slightly before the parallel grid generation takes place.

4.2 Further Issues

It is clear that the coarse mesh solve is a potential bottleneck for the domain decomposition preconditioner. Parallelization of this solve, using a conjugate gradient method for example, would result in a scalable algorithm but would be inefficient due to the communication requirements in the inner iteration where there is no scope for overlapping it with computation. By employing a serial solve and performing it on the least loaded processor we may hide some, if not all, of the computation, depending on the original load balancing. However, this is not particularly satisfactory since there may still be a high level of parallel slackness.

The technique we propose to overcome this is to take into account the cost of the coarse grid solve when performing the partitioning of the coarse grid prior to the mesh generation. The idea is to construct a weighted complexity model for the computational requirements of a coarse element (in terms of the number of internal and edge unknowns) and also for the coarse solve itself. Each dual graph vertex can then be weighted with the value given by the complexity model for the corresponding coarse element. The cost related to the coarse grid solve is added to one of the lower weighted vertices, v^* say, and the graph partitioned. The processor allocated v^* is thus responsible for the coarse solve and the whole computation is load balanced. This produces a scalable method without the need to parallelize the coarse grid solve and which is relatively straightforward to implement.

It is well known (e.g., [9]) that subdomains with bad aspect ratios have a negative impact on the conditioning of interface problems. In terms of the decomposition of unstructured meshes where the production of good aspect ratio subdomains is non-trivial, this brings in an added complication to the partitioning stage. This has been noted in unstructured preconditioning work by Farhat *et al* [10] and Le Tallec *et al* [22]. This constraint may in fact be in opposition to the communication minimization requirement that we have previously been trying to satisfy. Designing mesh partitioning algorithms which achieve all these aims for arbitrary meshes is therefore highly challenging. However, by treating each coarse mesh element as a subdomain, the problem of producing subdomains with good aspect ratios is transformed from one of partitioning to one of coarse grid generation. Of course the major objective of finite element mesh generation is to produce elements of a "good" shape and so the production of coarse elements/subdomains with good aspect ratios can be achieved with a suitable meshing algorithm.

In this paper we have presented a mesh generation and a domain decomposition solution approach which both rely on the same underlying coarse grid. It has been shown that this underlying coarse mesh allows these methods to be implemented efficiently on a parallel machine. The numerical examples used here have all involved the solution of simple linear equations, however the methods can also be applied to nonlinear problems as discussed in [16]. Future work will be directed at the repartitioning required after the mesh generation stage in order to improve the efficiency of the solver. It will also be interesting to try and extend the results to non-symmetric systems of equations resulting from the discretization of non-self-adjoint partial differential equations. Finally it should also be possible to have more than two levels to the hierarchy of meshes so as to get a true multigrid approach.

Acknowledgements

Our use of the Intel iPSC/860 parallel computing facility at the EPSRC Daresbury Laboratory throughout this work has been through SERC grant GR/J27066. The first author also acknowledges financial support from the EPSRC in the form of a research studentship.

References

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

[2] T. Arthur and M. J. Bockelie, A Comparison of Using APPL and PVM for a Parallel Implementation of an Unstructured Grid Generation Program, Tech. Report 191425, NASA Computer Sciences Corporation, Hampton, Virginia, 1993.

[3] R. E. Bank, *PLTMG: A Software Package for Elliptic Partial Differential Equations*, (SIAM, 1990).

[4] S. T. Barnard and H. D. Simon, A Fast Multilevel Implementation of Recursive Spectral Bisection for Partitioning Unstructured Problems, Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing, Norfolk, Virginia, 1993.

[5] R. Barrett, et al, Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, (SIAM, 1993).

[6] J. H. Bramble, J. E. Pasciak and A. H. Schatz, The Construction of Preconditioners for Elliptic Problems by Substructuring, I, *Math. Comp.* 47(1986) 103–134.

[7] M. Dryja and O. B. Widlund, Some Domain Decomposition Algorithms for Elliptic Problems, in *Iterative Methods for Large Linear Systems* (Academic Press, 1990) 273–291.

[8] L. C. Dutto, The Effect of Ordering on Preconditioned GMRES Algorithm for Solving The Compressible Navier-Stokes Equations, *Int. J. Num. Meth. in Eng.* 36(1993) 457–497.

[9] C. Farhat and M. Lesoinne, Automatic Partitioning of Unstructured Meshes for the Parallel Solution of Problems in Computational Mechanics, *Int. J. Num. Meth. in Eng.* 36(1993) 745–764.

[10] C. Farhat, J. Mandel and F. X. Roux, Optimal Convergence Properties of the FETI Domain Decomposition Method, *Comp. Meth. in Apl. Mech. Eng.* 115(1994) 365–385.

[11] G. H. Golub and C. F. Van Loan, *Matrix Computations*, (Johns Hopkins University Press, 2nd edition, 1989).

[12] A. Greenbaum, C. Li and H. Z. Chao, Comparison of Linear System Solvers Applied to Diffusion-Type Finite Element Equations, *Numer. Math.* 56(1989) 529–546.

[13] J. L. Gustafson, The Consequencies of Fixed Time Performance Measurement, Proceedings of Twenty Fifth Hawaii International Conference on System Sciences, Kauai, Hawaii, 1992.

[14] B. Hendrickson and R. Leland, An Improved Spectral Graph Partitioning Algorithm for Mapping Parallel Computations, *SIAM J. Sci. Comput.* 16(1995) 452–469.

[15] B. Hendrickson and R. Leland, A Multi-level Algorithm for Partitioning Graphs, Tech. Report SAND 93-1301, Sandia National Laboratories, 1993.

[16] D. C. Hodgson and P. K. Jimack, Efficient Mesh Partitioning for Parallel P.D.E. Solvers on Distributed Memory Machines, Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing, Norfolk, Virginia, 1993.

[17] D. C. Hodgson and P. K. Jimack, Parallel Generation of Partitioned, Unstructured Meshes, *Advances in Engineering Software* 27(1996) 59–70.

[18] D. C. Hodgson and P. K. Jimack, A Parallel Generator of Partitioned Unstructured Meshes Integrated with a Parallel Adaptive FE Solver, Proceedings of NAFEMS Fifth International Conference on Reliability of F.E.M.s for Engineering Applications, Amsterdam, The Netherlands, 1995.

[19] K. H. Hoffmann and J. Zou, Parallel Efficiency of Domain Decomposition Methods, *Parallel Computing* 19(1993) 1375–1391.

[20] C. Johnson, *Numerical Solutions of Partial Differential Equations by the Finite Element Method*, (Cambridge University Press, 1987).

[21] A. I. Khan and B. H. V. Topping, Parallel Adaptive Mesh Generation, *Computer Systems in Engineering* 2(1991) 75–101.

[22] P. Le Tallec, E. Saltec and M. Vidrascu, Solving Large Scale Structural Problems on Parallel Computers using Domain Decomposition Techniques, in B.H.V. Topping & M. Papadrakakis, eds., *Advances in Parallel and Vector Processing for Structural Mechanics*, (Civil-Comp Press, 1994).

[23] C. A. Leete, B. W. Peyton and R. F. Sincovec, Toward a Parallel Recursive Spectral Bisection Mapping Tool, Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing, Norfolk, Virginia, 1993.

[24] R. Löhner, R. Camberos and M. Merriam, Parallel Unstructured Grid Generation, *Comp. Meth. in Apl. Mech. Eng.* 95(1992) 343–357.

[25] G. Meurant, Domain Decomposition Methods for P.D.E.s on Parallel Computers, *Int. J. Supercomputer Applics.* 2(1988) 5–12.

[26] J. M. Ortega, *Introduction to Parallel and Vector Solution of Linear Systems*, (Plenum Press, 1988).

[27] B. N. Parlett, H. D. Simon and L. Stringer, Estimating the Largest Eigenvalue with the Lanczos Algorithm, *Math. Comp.* 38(1982) 153–165.

[28] H. D. Simon, Partitioning of Unstructured Problems for Parallel Processing, *Computing Systems in Engineering* 2(1991) 135–148.

[29] B. F. Smith, An Optimal Domain Decomposition Preconditioner for the Finite Element Solution of Linear Elasticity Problems, *SIAM J. Sci. Stat. Comput.* 13(1992) 64–378.

[30] X.-H. Sun and J. L. Gustafson, Toward a Better Parallel Performance Metric, *Parallel Computing* 17(1991) 1093–1109.

[31] B. H. V. Topping and A. I. Khan, Sub-Domain Generation Method for Non-Convex Domains, in B.H.V. Topping & A.I.Khan, eds., *Information Technology for Civil and Structural Engineers* (Civil-Comp Press, 1993).

[32] N. P. Weatherill and O. Hassan, Efficient 3D Grid Generation using the Delaunay Triangulation, *Comp. Fluid Dynamics '92* 2(1992) 961–968.

[33] N. P. Weatherill, A Method for Generating Irregular Computational Grids in Multiply Connected Planar Domains, *Int. J. Num. Methods in Fluids* 8(1988) 181-197.

[34] R. D. Williams, Performance of Dynamic Load Balancing for Unstructured Mesh Calculations, *Concurrency: Practice and Experience* 3(1991) 457–481.