Proceedings Paper:
Touheed, N. and Jimack, P.K. (1997) Improved parallel mesh generation through dynamic load-balancing. In: Topping, B.H.V., (ed.) Advances in Computational Mechanics with Parallel and Distributed Processing. The First Euro-Conference on Parallel and Distributed Computing for Computational Mechanics, 26 Apr - 01 May 1997, Lochinver, Scotland. Civil-Comp Press , Edinburgh , pp. 105-111. ISBN 0-948749-47-4

White Rose Consortium ePrints Repository
http://eprints.whiterose.ac.uk/

This is an author produced version of a paper published in **Advances in Computational Mechanics with Parallel and Distributed Processing.**

White Rose Repository URL for this paper:
http://eprints.whiterose.ac.uk/1659/

Improved Parallel Mesh Generation Through Dynamic Load-Balancing

N. Touheed and P.K. Jimack
Computational PDE Unit
School of Computer Studies
University of Leeds
Leeds LS2 9JT, UK

Abstract

Parallel mesh generation is an important feature of any large distributed memory parallel computational mechanics code due to the need to ensure that (i) there are no sequential bottlenecks within the code, (ii) there is no parallel overhead incurred in partitioning an existing mesh and (iii) that no single processor is required to have enough local memory to be able to store the entire mesh. In recent years numerous algorithms have been proposed for the generation of unstructured finite element and finite volume meshes in parallel. One of the main problems with many of these approaches however is that the final mesh, once generated, cannot generally be guaranteed to be perfectly load-balanced. In this paper we propose a post-processing step for the parallel mesh generator, based upon a cheap and efficient dynamic load-balancing technique. This technique is described and a number of numerical examples are presented in order to demonstrate that the quality of the partition of the mesh can be improved significantly at only a small additional computational cost.

1 Introduction

This paper is concerned with algorithms for the parallel generation of unstructured meshes of triangles or tetrahedra for complex geometries in two and three dimensions. Parallel mesh generation is a vital component of any distributed memory parallel computational mechanics code since it is highly undesirable that the size of the finite element or finite volume mesh be limited by the amount of memory on a single processor. Moreover, from the point of view of parallel efficiency and scalability it is essential to minimize the sequential bottlenecks within a code. In addition, the parallel overhead associated with partitioning a mesh generated on a single processor should be avoided.

A large number of algorithms and codes have been developed for parallel mesh generation in recent years and these may be divided into two broad categories: those based upon refinement of an initial coarse background mesh (e.g. [5, 10, 13, 12]), and those which mesh the domain in an alternative manner (e.g. [1, 9]). In this paper we are concerned only with the first of these two categories and, for simplicity of exposition, we concentrate on the 2-d case. Extension to 3-d is possible and is the topic of much current research.

The common feature of all of the parallel mesh generators based upon refinement of a background grid is that this grid must first be partitioned across the available processors. The techniques by which this is done vary significantly but they each have the same goal: to ensure that the total number of generated elements or points on each processor is about the same. Hence, if a mesh of uniform density is being generated and the background grid is also of uniform density then we would expect each processor to be assigned about the same number of coarse elements. If, on the other hand, a mesh of non-uniform density is being generated from a uniform background grid then we would expect a potentially different number of coarse elements to be assigned to each processor. A secondary objective when partitioning the background grid is to ensure that the number of generated elements which have an edge on the boundary between two processors is as small as possible. This will ensure that the amount of communication required by the finite element or finite volume solver will be minimized.

In order to attempt to achieve these objectives, *a priori* estimates need to be made about how many elements, edges and nodes will be generated within each coarse element. Inevitably the actual values of these three numbers after generation will not precisely match these estimates. In order to keep the differences as small as possible some authors have developed quite elaborate schemes for improving the quality of their estimates; including the use of neural networks [13] or "virtual refinement" [5] for example. Even with these schemes however final load imbalances of up to 10% are frequently observed in practice.

In this paper we suggest that, so long as a reasonable partition is produced *a priori*, a more profitable use of resources is to improve the quality of the partition after the mesh has been generated in parallel through the use of a parallel post-processing step. This step simply involves making local modifications to the load-balance before the solution phase commences. As will be demonstrated these local modifications are made in a manner designed to strike a balance between the potentially conflicting requirements of

1. improving the load-balance,

2. maintaining data locality,

3. minimizing the number of fine edges shared by two processors,

4. avoiding sequential bottlenecks.

2 The Post-Processing Step

In this section we describe our post-processing algorithm in quite general terms. Section 3 then discusses some of the implementation issues, with reference to a particular parallel mesh generation code ([5]) where necessary. We begin by assuming that a mesh has just been generated in parallel based upon some refinement of a coarse background grid which is partitioned across a distributed memory architecture. We now define the weighted dual graph for the background grid to be such that each node of the graph corresponds to a coarse element (with weight equal to the number of generated elements within it), and each edge of the graph joins nodes whose corresponding coarse elements have a common edge (with weight equal to the number of generated edges along it). The task of the post-processor is to modify the existing partition of this weighted graph in line with the four requirements enumerated at the end of the previous section.

2.1 Group balancing

Following Vidwans *et al* [16], we define a further weighted graph: the weighted partition communication graph (WPCG). This represents the face adjacency of the p processors being used (processors that share at least one edge of a coarse element with a given processor are said to be face adjacent to that processor). A WPCG is obtained by having one vertex for every processor and an edge between two vertices if and only if they are face adjacent to each other. The weight w_{N_i} of the ith vertex is equal to the sum of weights of all coarse elements on the ith processor and the weight $w_{E_{ij}}$ of the edge connecting the ith and j^{th} processors is equal to the sum of weights of all coarse element edges on the interpartition boundary between the two processors.

We now divide the WPCG into two subgroups denoted by Group1 and Group2. Unlike in [16] however we use a weighted version of the spectral bisection method [5] which results in a partition which is based upon having an approximately equal weight in each group, rather than an equal number of processors. Moreover the spectral algorithm is also designed to keep the weight of those edges of the WPCG which are cut by the partition (the "cut-weight") as low as possible. The cost of implementing this algorithm is not significant since the number of processors, p, is always small compared with the size of the coarse mesh.

A full description of the weighted spectral bisection algorithm may be found in [5]. Briefly, a weighted Laplacian matrix, L, for the WPCG is first formed and then scaled by the diagonal matrix $D = \text{diag}\left(\frac{1}{\sqrt{w_{N_i}}}\right)$. The second eigenvector (or Fiedler vector), $\underline{u}_2$, of the scaled matrix $S = D^T L D$ is then found. Finally a partitioning vector, $\underline{x}$, is defined by $x_i = u_{2_i}/w_{N_i}$ for $i = 1, ..., p$. The two subgroups are then defined by sorting the p vertices of the WPCG according to the size of their entry in $\underline{x}$

and placing elements represented by x_i': $i = 1...n$ in one group (with $\underline{x}'$ being the sorted vector) and those by x_i': $i = n + 1, ..., p$ in the other, with n chosen so that

$$\left| \sum_{i=1}^{n} w_{N_i}' - \sum_{i=n+1}^{p} w_{N_i}' \right| \qquad (1)$$

is as small as possible (where w_{N_i}' is the weight of the vertex represented by x_i').

If the generated mesh is quite uniformly distributed across the processors then we would expect each group to contain about the same number of processors and an almost identical total weight. If the generated mesh is not uniform or the partition is not well load-balanced however then the number of processors in each group may be very different. In either case the cut-weight resulting from this bisection will generally be small. In the next stage of the algorithm we use the idea of local migration from the "larger" to the "smaller" group so that after migration each group contains approximately the same average weight per processor without there being a significant increase in the cut-weight.

2.2 Local migration

As mentioned above the subgroups formed in the last subsection may not be ideally balanced. To balance them we now migrate nodes of the weighted dual graph (i.e. coarse elements and their generated meshes) from the "larger" to the "smaller" group. There are many ways to do this but, due to the non-linear complexity of the Kernighan and Lin algorithm ([8]), we apply the ideas of Fiduccia and Mattheyses ([3]) who suggest a similar algorithm but whose complexity is linear.

We first decide which of the subgroups is to be the Sender and which the Receiver. We then define the number Mig_{tot} to equal the total weight of all the nodes which are to be migrated from the Sender to the Receiver. Let N_1 and N_2 be the number of processors in Group1 and Group2 respectively. Also let Ave be the average weight per processor in the WPCG and Ave_1 and Ave_2 be the average weights per processor in Group1 and Group2 respectively. Then the calculation of Sender, Receiver and Mig_{tot} is shown in figure 1 below (in order to calculate Mig_{tot} one simply multiplies the average excess load per processor by the number of processors in the Sender group). Note that if the combined weight of the nodes transferred between the Sender and the Receiver is nearly or exactly equal to Mig_{tot} then the two groups will be load-balanced upon completion.

Having established the required load to be transferred, the next issue to address is that of how many nodes (i.e. coarse elements) each processor in the Sender group should actually send and which processors in the receiver group they should be sent to. Again we build upon the algorithm of Vidwans *et al* [16], by defining the concept of candidate processors. Processors in each group that are face-adjacent to at least one processor in the other group are called candidate processors. We only allow the candidate processors to be involved in the actual migration

```
    if(Ave₁ ≤ Ave₂){
        Sender = Group2;
        Receiver = Group1;
        Migₜₒₜ = N₂ * (Ave₂ − Ave);
    }
    else{
        Sender = Group1;
        Receiver = Group2;
        Migₜₒₜ = N₁ * (Ave₁ − Ave);
    }
```

Figure 1: Calculation of Sender, Receiver and Mig_{tot}.

of nodes from Sender to Receiver. Let N_{tot} be the total weight on all candidate processors of the Sender group. Then if the ith candidate processor in the Sender group is face adjacent to more than one candidate processor in the Receiver group we migrate nodes to that candidate processor which has the "longest" boundary (by this we mean that the cut-weight between the two processors involved is maximum as compared to other possible pairs). The amount of load shifted from the ith candidate processor in Sender group is denoted by Mig_i and is given by,

$$\text{Mig}_i = \left(\frac{N_i}{N_{tot}}\right) * \text{Mig}_{tot}, \qquad (2)$$

where N_i is the total weight of the ith processor.

Finally, it is necessary to decide precisely which nodes in the weighted dual graph of the coarse mesh should be transferred. Our aim is to transfer those nodes which result in as low a cut-weight as possible. The fundamental ideas behind this are the concepts of the "gain" and "gain density" associated with moving a node onto a different processor. For a node, k say, which is situated on the ith candidate processor in the Sender group, we define the gain(k) associated with this node to be the net reduction in the cut-weight that would result if this node were to migrate to the Receiver group (the jth candidate processor in the Receiver group say). The calculation of gain(k) is shown in figure 2.

$$\text{gain}(k) = \sum_{(k,l)} \begin{cases} w_{E_{kl}} & \text{if } l \in j^{th} \text{ processor,} \\ -w_{E_{kl}} & \text{if } l \in i^{th} \text{ processor,} \\ 0 & \text{otherwise.} \end{cases}$$

Figure 2: The calculation of gain.

The gain density of a node is defined as the gain of the node divided by the weight of the node. The bulk of the work needed to make a move consists of selecting the base node (a node which is about to be shifted from one processor to another processor is called a base node), moving it, and then updating the gains of its neighbouring nodes. We solve the first problem, that of selecting a base node, by choosing the node with the largest gain density on the ith processor whose weight is less than or equal to Mig_i. We shift the node to the receiving processor and update

the gains of its neighbouring nodes (observe that in general the node k will have three neighbours when we are working with triangulations of two-dimensional domains and four neighbours when we are working with triangulations of three-dimensional domains) using the algorithm outlined in figure 3. Observe that, if the gain associated with the base node is positive, then transferring it will not only improve the load-balance but will also reduce the total cut-weight between the two groups.

```
    For each nₖ which is a neighbour of the node k {
        Let pₖ be the processor to which nₖ belongs;
        if (pₖ == j) then
            decrement gain(nₖ) by 2*w_{E_{nₖk}};
        else if(pₖ == i) then
            increment gain(nₖ) by 2*w_{E_{nₖk}};
    }
```

Figure 3: Updating the gains.

2.3 Divide and conquer and parallel implementation

Once we have obtained Sender and Receiver groups with the same average weights, it is possible to recursively apply the above splitting algorithm to each of these two processor groups in parallel: bisecting them and load-balancing them. The recursion terminates when every group consists of a single processor: each with approximately the same load.

This divide and conquer approach naturally permits a certain degree of parallelism in its implementation. Further parallelism is also facilitated by the fact that it is possible for more than one sending processor in a Sender group to migrate data onto its corresponding receiving processor at any given time. To ensure that no data conflicts arise as a result of this parallel communication an additional global communication is required at each step of the recursion. Full details are given in subsection 3.1.

The implementation of this load-balancing algorithm that is used for the numerical experiments described in section 4 was completed using the MPI library ([11]). This is ideally suited to the divide and conquer philosophy since it provides explicit mechanisms for the definition and splitting of processor groups. To implement this divide and conquer philosophy we make use of the function MPI_Comm_split() available in the MPI library. This function takes as input a communicator, a colour, and a key. All processors with the same colour are placed into the same new communicator, which is returned in the fourth argument. The processes are ranked in the new communicator in the order given by the key. In our application we assign the value 1 (value 0) to colour if the processor is in the Sender group (Receiver group) and the key is taken to be the ID (rank) of the processor.

When a coarse element migrates from the Sender group to the Receiver group we have to update numerous data structures (which not only involve the processors on the Receiver and the Sender groups but may also involve processors outside these two groups), so it is necessary to

maintain the presence of the initial group. This means that each processor is a member of two groups: the initial group (called the I_Group) which consists of all p processors and remains unchanged throughout, and the current group (referred to here as "the Group") which is variable and changes with each application of the Divide and Conquer algorithm.

Note that the above Divide and Conquer algorithm is repeated until all Groups contain exactly one processor. If a group consists of a single processor before the algorithm terminates then that processor is not entirely idle since it must still communicate with other processors in case neighbouring coarse elements are migrated between two processors.

3 Implementation Issues

An overview of the above post-processing algorithm is given in figure 4. In this section we briefly consider how this may be efficiently implemented and describe what is meant by the step in figure 4 entitled "modify the necessary data structures to reflect the migration". A much more comprehensive discussion of these issues for the particular parallel mesh generator described in [5] may be found in [14].

While (Any Groups contain two or more processors){
    Find the maximum load Max and the avarege load Ave
    of the Group;
    Find the percentage of maximum imbalance max_imb
    in the Group by using the formula;
      max_imb = ((Max - Ave) / Ave) * 100;
    If (The Group contains more than one processor){
      Send the contribution to the Laplacian matrix to
      processor 0;
      If (Rank of the processor is 0){
        Form the Laplacian matrix after receiving the
        contribution from other processors;
        Find the Fiedler vector and by using it decide
        the Receiver and Sender groups;
      }
      If (max_imb is more than a given tolerance){
        Move load from processors in the Sender Group
        to processors in the Receiver in such a way that
        after migration the two Groups have the same
        average load and the increase in the cut weight
        is as small as possible;
      }
    }
    If (The migration effects the current processor){
      Modify the necessary data structures to reflect the
      migration;
    }
    Divide the Group into two Groups (i.e. from now on
    both Sender and Receiver will be called Group);
}

Figure 4: Parallel post-processing algorithm.

3.1 Avoiding data conflicts due to parallel communication

As outlined in subsection 2.3 one aspect of the parallel implementation is to allow more than one sending processor in a Sender group to migrate data to its corresponding receiving processor at a given time. In fact all of the communication of coarse elements may be left to the end of each divide and conquer iteration and completed concurrently. The difficulty with this is that when a coarse element (and its corresponding region of the generated mesh) is transferred, it is not just the sending and receiving processor that are required to communicate. If a neighbour of the coarse element happens to reside on a third processor then this also needs to be notified of the transfer (to facilitate communication within the parallel finite element or finite volume solver when this is used). Clearly a difficulty will arise if this neighbouring coarse element is also being transferred to another processor.

One way to overcome this difficulty would be to route messages forward from processors that neighbouring coarse elements have just left. This would not be straightforward to implement however. We prefer to avoid the difficulty by making an all-to-all global communication immediately before the data migration phase of each divide and conquer iteration. This communication informs each processor of the destination of each coarse element that is about to be migrated, hence allowing all problems of locating neighbours to be avoided. The price that we pay for this simplicity is the introduction of a global synchronization point within the algorithm as well as the cost of these global communications (although the communication costs associated with forwarding messages to neighbours are saved of course).

Yet another approach to dealing with this difficulty of locating neighbouring coarse elements is to use a colouring of the weighted dual graph in which neighbouring elements have different colours. By only transferring coarse elements one colour at a time it becomes easy for these elements to keep track of their neighbours. This method appears to work well in two dimensions [14, 15] however, for problems in three dimensions the number of different colours required becomes prohibitive and so we take this approach no further.

3.2 Updating data structures

Having generated a mesh in parallel and then decided which background elements to migrate locally in order to improve the partition of this mesh, the final stage of the algorithm is to complete this migration. As indicated in subsection 3.1 there are a number of practical considerations to be made here. These are due to the requirement that the consistency of the distributed mesh data be maintained so that the parallel finite element or finite volume solver will still function correctly. Clearly the exact details of what data structures within a code are affected by these local migrations will depend upon the specific implementations of the parallel mesh generator and the differential equation solver. In [14] details of the bookkeeping associated with modifying the partition resulting

from one such parallel mesh generator (see [5]) are explained in some depth.

The main requirements for this particular generator and solver are that all vertices and edges of the background grid which lie on the interprocessor boundary should be identified. This means that whenever a coarse element is transferred it is necessary to check which of its vertices, if any, are added to the interprocessor boundary, and which, if any, are removed. A similar check is required for each edge of the coarse element so that data structures on the processors on either side of the boundary may be kept consistent (again, see [14] for further details).

4   Computational Examples

In this section we present two representative example problems in which the parallel post-processing procedure is used to improve the performance of a parallel mesh generation code. The mesh generator is again taken from [5], as is one of the geometries which we use. The other geometry is taken from [2].

In the first example the domain is L-shaped (geometry 1 in [5]) and the generated mesh is heavily refined in the regions surrounding three of the corners. Table 1 shows how the original parallel mesh generator performs and also how the load-balance is improved by the post-processing step. The time taken by the slowest processor in the mesh generation phase is 8.3 seconds on a Cray T3D whilst the time taken by the last processor to finish the post-processing step is a mere 0.2 seconds. One measure of the improvement in the load-balance is the percentage by which the processor with the largest load exceeds the average load across all of the processors: the maximum imbalance. This has been reduced from 10.4% to just 3.4%. A small price is paid for this significant improvement in that the total number of fine triangle edges which lie on the interprocessor boundary has increased slightly: from 3091 edges to 3168 edges. Nevertheless, practical experience of the trade-off between cut-weight and load-balance suggests that this is a price well worth paying [4].

The second example involves the generation of a mesh on a slightly more complex domain which is taken from [2] (Bank's "Texas" geometry). In contrast with the previous example the mesh that is generated is not so non-uniform in its density throughout the domain and so the number of coarse elements associated with each processor does not vary so dramatically. Table 2 shows how the original mesh generator performs for this problem and how the post-processing step improves the load-balance. In this case the initial maximum imbalance is only 2.7% but this figure is still improved to just 0.6% by the post-processing step. As with the first example the dynamic re-balancing is also very cheap: taking just 0.2 seconds to complete on a Cray T3D, as compared with an initial parallel mesh generation time of 84.1 seconds. Finally, the total number of fine element edges lying on the partition boundary has only been increased from 11533 to 11563.

Note that the reason that the final load-balance is so much better in the second example than in the first is

Processor	Original		Modified	
	coarse	fine	coarse	fine
1	112	13044	111	13457
2	112	13304	111	13404
3	26	14963	22	13725
4	22	14146	21	13857
5	161	13058	161	13357
6	46	14012	43	13523
7	16	14013	16	14013
8	18	13625	18	13360
9	16	13767	16	13641
10	41	13895	38	13545
11	168	13072	175	13627
12	130	12647	132	13539
13	72	13687	70	13485
14	102	13002	106	13390
15	173	13261	175	13463
16	115	13342	115	13452
average	83	13552	83	13552

Table 1: The performance of the post-processing algorithm on example one.

that the elements of the background grid are much more evenly spread across the processors in this case. When a very large number of fine elements are generated in just a few background elements it is significantly more difficult to obtain a precise load-balance by partitioning the background grid. It is important therefore that, in regions of heavy local refinement, there should be a sufficient number of coarse elements to permit the possibility of obtaining a reasonable load-balance.

5   Discussion

In this paper we have introduced a post-processing algorithm for the parallel generation of unstructured meshes for use in parallel finite element or finite volume analysis. The algorithm is based upon performing a local modification of the partition of an underlying background grid from which the mesh was generated in parallel. This modification aims to improve the load-balance whilst respecting data locality and ensuring that the length of the partition boundary is not increased unnecessarily.

We have successfully demonstrated an implementation of this algorithm for two different problems in two dimensions. In addition it has been shown that the execution time of the code, implemented in C using MPI, is extremely competitive. It should be noted however that the post-processing step described here can only be as effective as the coarse mesh allows it to be. For example, if the background grid only has a small number of elements which are evenly spread across the domain and the fine mesh is very fine in some particularly local regions, then it is possible that even an optimal solution of the corresponding load-balancing problem may have a very large imbalance and/or cut-weight.

There are a number of alternative parallel dynamic

Processor	Original		Modified	
	coarse	fine	coarse	fine
1	302	200856	297	197704
2	224	194426	229	197578
3	214	194531	219	198018
4	259	196816	260	198715
5	262	203560	255	198326
6	244	196841	245	198142
7	257	199044	257	198152
8	268	196539	271	198148
9	240	198223	240	198223
10	225	199385	225	199385
11	221	199123	220	198393
12	227	199179	227	199179
13	233	196701	233	196701
14	224	196846	224	196846
15	256	199915	255	199194
16	238	198702	237	197983
average	243	198168	243	198168

Table 2: The performance of the post-processing algorithm on example two.

load-balancing algorithms that we might have used as part of the post-processing step in this work. In [14, 15] we make comparisons with two such algorithms: our own implementations of [6] and [16]. The conclusion there is that for problems with a fairly uniform final mesh all of the approaches implemented work very well. However, for more demanding (and perhaps more realistic) examples in which there is heavy local refinement in some regions of the domain, the technique described in this paper appears to find a good balance between maintaining a low cut-weight and distributing the dual graph in a balanced fashion.

Recently, parallel versions of the publicly available software packages METIS [7] and Jostle [17] have been announced and so it may also be possible to make use of these within the post-processing step. In addition, current research is looking at the application of this post-processing step to a 3-d parallel adaptive code ([12]). This particular code is rather different from that used for the results in section 4 since it makes use of "halo" elements on the boundary between processors. These significantly simplify the parallel solver by allowing each processor to have copies of those elements immediately on the other side of each processor boundary. This use of halo elements serves to complicate the dynamic load-balancing phase however since halo information must also be passed when elements are relocated. Nevertheless, initial re-balancing results obtained for typical meshes generated and adapted in 3-d appear to be very encouraging.

Acknowledgements

References

[1] 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.

[2] R.E. Bank, *"PLTMG Users' Guide 7.0"*, SIAM, Philadelphia, 1994.

[3] C.M. Fiduccia and R.M. Mattheyses, *"A Linear_Time Heuristic for Improving Network Partitions"*, Proceedings of the Nineteenth IEEE Design Automation Conference, IEEE, pp. 175–181, 1982.

[4] D.C. Hodgson and P.K. Jimack, *"Efficient Mesh Partitioning for Parallel Elliptic Differential Equation Solvers"*, Computing Systems in Engineering, 6, pp. 1–12, 1995.

[5] D.C. Hodgson and P.K. Jimack, *"Efficient Parallel Generation of Partitioned, Unstructured Meshes"*, Advances in Engineering Software, 27, pp. 59–70, 1996.

[6] Y.F. Hu and R.J. Blake, *"An Optimal Dynamic Load Balancing Algorithm"*, Preprint DL-P-95-011 of The Central Laboratory for the Research Councils, Daresbury Laboratory, Daresbury, Warrington, Cheshire WA4 4AD, UK, 1995.

[7] G. Karypis and V. Kumar, *"A Coarse-Grain Parallel Formulation of Multilevel k-way Graph Partitioning Algorithm"*, Proceedings of Eighth SIAM Conference on Parallel Processing for Scientific Computing, SIAM, Philadelphia, 1997.

[8] B. Kernighan and S. Lin, *"An Efficient Heuristic Procedure for Partitioning Graphs"*, Bell System Technical Journal, 29, pp. 209–307, 1970.

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

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

[11] Message passing Interface Forum, *"MPI: A Message Passing Interface Standard"*, Int. J. of Supercomputer Applications, 8, no. 3/4, 1994.

[12] P.M. Selwood, M. Berzins and P.M. Dew, *"3D Parallel Mesh Adaptivity: Data-Structures and Algorithms"*, Proceedings of Eighth SIAM Conference on Parallel Processing for Scientific Computing, SIAM, Philadelphia, 1997.

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

[14] N. Touheed and P.K. Jimack, *"Parallel Dynamic Load-Balancing for Adaptive Distributed Memory PDE Solvers"*, School of Computer Studies Research Report 96.34, University of Leeds, Leeds LS2 9JT, UK, 1996.

[16] A. Vidwans, Y. Kallinderis and V. Venkatakrishnan, *"Parallel Dynamic Load_Balancing Algorithm for Three_Dimensional Adaptive Unstructured Grids"*, AIAA Journal, Vol.32, No.3, pp. 497–505, 1994.

[17] C. Walshaw, M. Cross and M.G. Everett, *"Dynamic Load-Balancing for Parallel Adaptive Unstructured Meshes"*, Proceedings of Eighth SIAM Conference on Parallel Processing for Scientific Computing, SIAM, Philadelphia, 1997.

[15] N. Touheed and P.K. Jimack, *"Dynamic Load-Balancing for Adaptive PDE Solvers with Hierarchical Meshes"*, Proceedings of Eighth SIAM Conference on Parallel Processing for Scientific Computing, SIAM, Philadelphia, 1997.