

# Scalable Parallel Generation of Partitioned, Unstructured Meshes

D.C. Hodgson, P.K. Jimack, P. Selwood and M. Berzins

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

In this paper we are concerned with the parallel generation of unstructured meshes for use in the finite element solution of computational fluid dynamics problems on parallel distributed memory computers. The use of unstructured meshes allows the straightforward representation of geometrically complicated domains and is ideally suited for adaptive solution techniques provided the meshes are sensibly distributed across the processors. We describe an algorithm which generates well-partitioned unstructured grids in parallel and then discuss the quality of this mesh and its partition, and how this quality can be maintained as the mesh is modified adaptively.

## 1. INTRODUCTION

The usual approach to solving finite element (or finite volume) problems in parallel on a distributed memory machine is to decompose the mesh into a number of subdomains and to allocate each of these subdomains to a processor. This decomposition of the elements of the mesh should have two main features. Each subdomain should contain approximately the same number of node points (or elements for cell-centered finite volumes), so as to achieve “load-balancing”. Also, the number of node points (or edges for cell centered finite volumes) which lie on the boundary between different subdomains (we will refer to such points as “interpartition boundary vertices/edges”) should be kept to a minimum since the amount of interprocessor communication will depend upon this number.

There has been a considerable amount of research into the problem of partitioning an existing mesh across distributed memory in a manner compatible with the above (see for example [4,9] and references therein). However these methods assume that the mesh is not held in a distributed manner across a multi-processor machine but is stored in one place. This is clearly prohibitive since the size of the mesh is constrained by the memory available on the single processor on which it is stored. In addition, if we wish to solve very large problems in parallel we do not want the mesh generation and partitioning to be a serial bottleneck. For these reasons the main contribution of this paper is to illustrate a technique for generating an automatically *partitioned* mesh in *parallel*. This technique is outlined in the next section and its performance is analyzed and discussed in section 3.

In section 4, a number of extensions of the work are considered, including its application to time-dependent problems using adaptivity through local *h*-refinement and derefinement. Here being able to generate a well-partitioned initial mesh in parallel is not sufficient since the refinement process will destroy the load-balance as time progresses. The issue of dynamically modifying the existing partition is therefore addressed.

## 2. THE PARALLEL MESH GENERATOR

In order to create a mesh in parallel, the domain must be split up into subregions which can then be meshed simultaneously and independently. Our method does this by producing in serial an initial coarse, or background, triangulation (tetrahedralization in 3-d) of the domain, using a Delaunay algorithm as described in [14], and then distributing this background grid across the processors in an intelligent manner before meshing begins.

This distribution of the background grid is performed so as to ensure that each processor will generate a mesh of about the same size *and* the number of interpartition boundary vertices will be very low. In order to achieve this, a weighted dual graph of the background grid is first produced, as shown in figure 1. Here, the weight of each graph vertex is equal to the number of nodes that it is estimated will be generated within the background element to which that vertex corresponds ( $W_{v(\ell)}$  say), and the weight of each graph edge is equal to the number of nodes that it is estimated will be generated along each background edge (face in 3-d) to which that graph edge corresponds ( $W_{e(n,m)}$  say). This weighted dual graph is then partitioned using a recursive spectral bisection algorithm, such as described in ([3,9]). Spectral algorithms seek to partition the graph so that the number of boundary vertices is minimized subject to the constraint of maintaining load-balance. This means that, provided our estimates,  $W_{v(\ell)}$  and  $W_{e(n,m)}$ , are reliable, the partition of the background grid should be very well-suited for the parallel generation of our unstructured mesh.

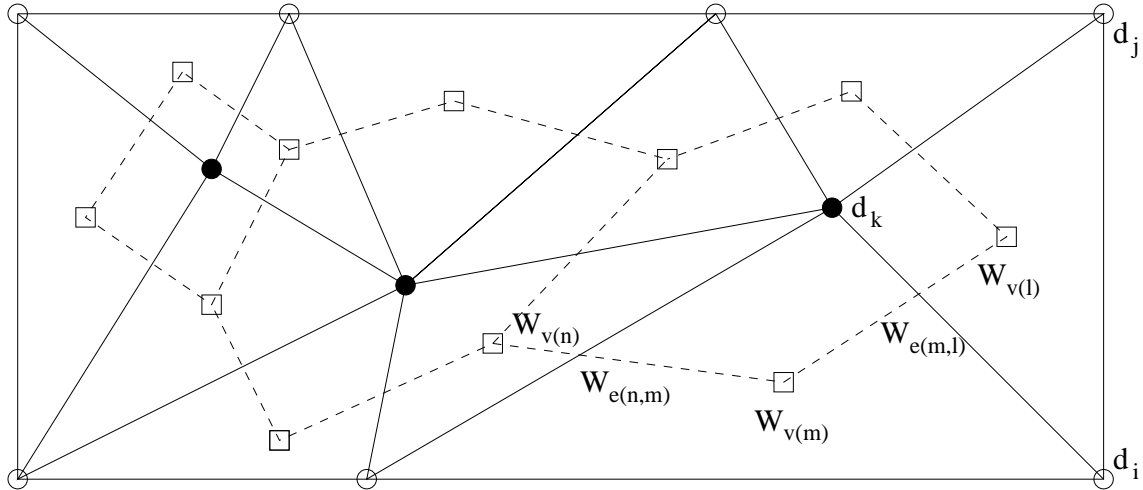


Figure 1. An example of a coarse background grid in 2-d along with its weighted dual graph (with vertex weights  $W_{v(\cdot)}$  and edge weights  $W_{e(\cdot,\cdot)}$ ). Also shown are some typical point distribution values ( $d_i$ ,  $d_j$  and  $d_k$ ).

The mesh generation itself also uses the same Delaunay algorithm [14], this time in parallel on each processor. Two options are available: either each processor can Delaunay mesh the union of those background elements which it has been allocated, or it can mesh

each of its background elements separately. In either case, the density of the generated mesh is governed locally through the use of point distribution values at each coarse grid vertex. These allow different target mesh spacings to be specified throughout the domain, as described in [14]. An additional, and equally important, use for these point distribution values is to allow the estimates  $W_{v(\ell)}$  and  $W_{e(n,m)}$  to be formed very cheaply using straightforward geometric formulae. For example, in two dimensions, the simple estimate

$$W_{v(\ell)} = \frac{\text{Area of Element } \ell}{\left(\frac{d_i+d_j+d_k}{3}\right)^2}$$

proves to be surprisingly reliable, where the point distribution values,  $d_i$ ,  $d_j$  and  $d_k$ , are shown in figure 1 (see [5] for further details).

This mesh generation procedure appears to be both well load-balanced and highly scalable. The load balance comes from the fact that each processor is generating a mesh contribution of about the same size, even on a highly irregular mesh, whilst the scalability comes from the fact that the only sequential steps are applied to the coarse background grid rather than the final mesh itself. Moreover, once the background grid has been partitioned there is no need for any inter-processor communication to take place (providing consistent algorithms are used to mesh subdomain boundaries so as to ensure that they match-up on neighbouring processors). In contrast with this, the methods described in [1] and [6] both farm out subregions to processors for meshing, without distributing them in a considered manner. This means that the generated mesh may not be very well balanced across the processors and that there is no guarantee that subregions sharing a processor will be connected. It is therefore always necessary to repartition these meshes once they have been generated in parallel. On the other hand, the parallel generator suggested in [7] *is* designed to produce meshes that are already well partitioned, using a “wavefront” approach to split up a background grid. However, no attention is explicitly paid to keeping the number of interpartition boundary vertices low, so the quality of the partitions produced is likely to be affected by this.

### 3. PERFORMANCE

The algorithm described in the previous section has been implemented using MPI ([8]) so as to ensure portability. It has been run successfully on a variety of platforms, including a distributed memory computer (a 64 node Intel hypercube), a shared memory computer (an 8 processor SGI Power Challenge) and also on a cluster of 16 SGI Indy workstations.

The outcomes of some typical computations are shown in Figure 2 and Table 1. Here, an unstructured mesh has been generated around a NACA0012 aerofoil using a point distribution function which is suitable for a supersonic flow (free stream Mach number = 2.0) with a moderate Reynolds number ( $Re = 500$ ). The coarse background grid contains 1239 elements and the mesh that is generated contains almost half a million elements. The generation times for this mesh were 62.2 seconds on the Intel i860 (using 8 processors) and 40.2 seconds on a cluster of 8 Indy workstations. As can be seen, the mesh density varies enormously throughout the domain, yet each partition is of a very similar size. Demonstrating that the number of interpartition boundary vertices is also low is quite hard to do quantitatively for an example of this size, however one can see from the shaded coarse mesh in Figure 2 that the subdomains are all connected and have compact shapes.

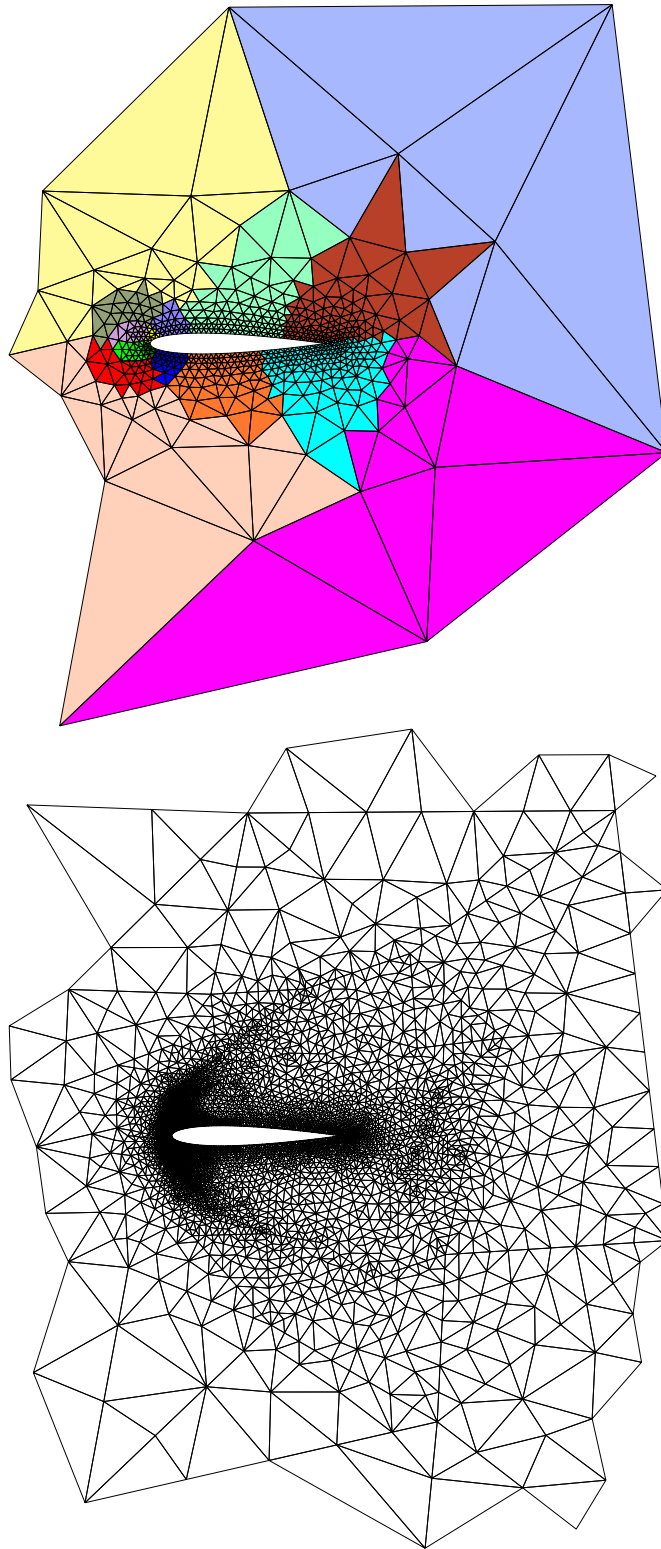


Figure 2. The coarse background mesh (with shading to illustrate how it has been partitioned into 8 subregions) and the final mesh generated around a NACA0012 aerofoil.

Table 1

Details of the parallel generation of a 447 403 element unstructured mesh around a NACA0012 aerofoil.

Subdomain	Vts.	%age diff.	i860	Indys
1	28639	+0.6	62.2	40.2
2	28061	-1.4	51.1	31.3
3	28928	+1.6	62.1	35.2
4	27925	-1.9	51.7	29.1
5	28552	+0.3	47.2	21.4
6	28215	-0.9	52.8	27.7
7	28763	+1.1	50.6	25.3
8	28613	+0.5	50.2	22.1

Experiments with larger meshes have shown that as the number of processors and the mesh size increase the generator can be shown to scale reasonably well. For example, using 16 nodes on the Intel i860 to generate a mesh in excess of a million elements takes little over 100 seconds. Also, the maximum difference in the size of each subregion always appears to be between  $-5\%$  and  $+5\%$  of the average size.

The generation of separate Delaunay meshes within each element of the background grid means that the final mesh is only locally Delaunay. This does not appear to affect the quality of this mesh adversely however since the number of coarse elements is always far smaller than the number of elements actually generated.

It is important to stress that, even though the number of vertices generated by each processor is about the same, the time taken by each processor varies more greatly. This is entirely due to the fact that the mesh being generated is of such a variable density, which causes some processors to be allocated many fewer background elements than others. It is these processors which are the last to finish since the sequential generation algorithm used, [14], is slower at generating a few large meshes than a large number of moderate meshes. Ideally therefore, the density of the coarse mesh should be made to reflect the required final mesh density everywhere. When this is done, the variation in meshing times between processors falls off drastically (thus leading to greater efficiency).

Another improvement that can be made to the algorithm as it is described above, is to attempt to reduce further the small (5%) variations in the mesh sizes on each processor. Such variations can still lead to a noticeable drop in the efficiency of a parallel solver, and so a small amount of post-processing of the partition may be worthwhile. One solution is to move a background element (and its sub-mesh) from one processor to another which contains fewer nodes, while keeping the total number of interpartition boundary vertices as low as possible.

#### 4. EXTENSION TO ADAPTIVE SOLUTION METHODS

Generation of the initial fine mesh is only one part of the solution process. The use of adaptive methods means that the initial mesh will have to be both refined and coarsened

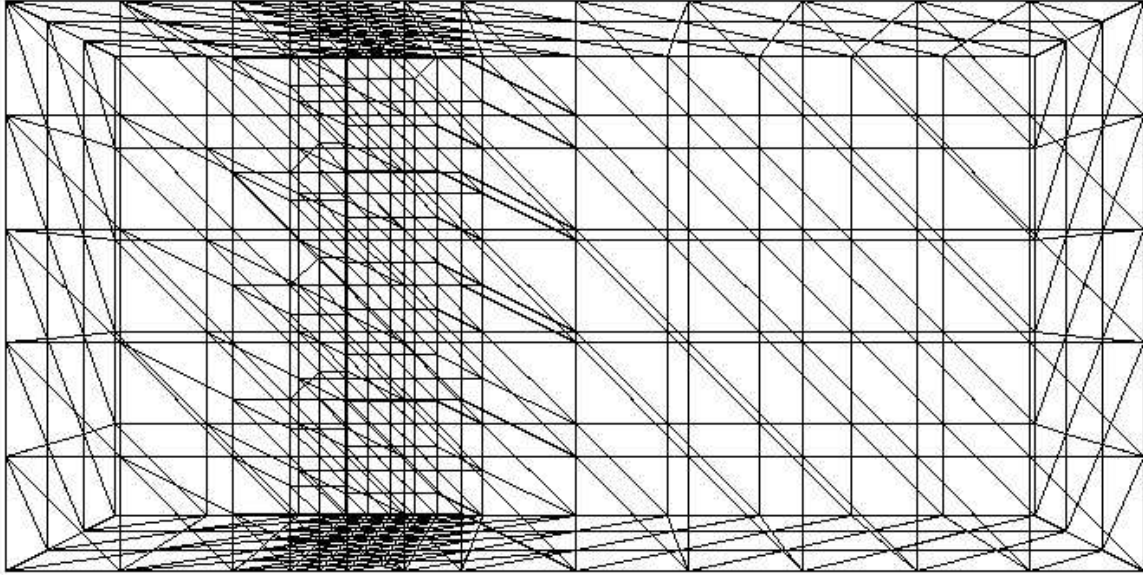


Figure 3. An unstructured 3-d mesh which accurately represents a steep front which is about to be advected from left to right.

in different areas on the basis of computed error estimates. This problem of dynamic load balancing during the adaptive solution of time-dependent problems has been considered by a number of authors, such as [2,11,12]. This problem is essentially the same as that already described above: where an existing partition has a low number of interpartition boundary vertices but a less than perfect load balance.

The problem is illustrated by Figure 3 which shows a three dimensional tetrahedral mesh, generated using the code of [10], that is used to represent a simple function with a shock. This is used as initial data for the linear advection equation, with the shock being advected to the right. In [10] an unstructured mesh adaption algorithm is described which is suitable for just this class of problem. As the front advects, the mesh is refined immediately ahead of it and coarsened immediately behind it. A key issue for any parallel solver is therefore to ensure that as the mesh changes, each processor maintains a roughly equal share of the elements and unknowns. One way to achieve this would be to generate an entirely new mesh every few time-steps, using the technique of Section 2. Alternatively, one could use conventional hierarchical refinement and derefinement but form a new partition of the mesh whenever adaptivity has occurred. Both of these approaches seem inefficient however, as they fail to make the best use of the existing partition.

The approach here is to make use of hierarchical refinement and derefinement of a coarse background grid, and to only consider altering the partition of this background grid after each adaptive step has occurred. As with the original mesh generation, this has the advantage that the partitioning problems considered are always much smaller than if one were to attempt to directly partition the mesh at its finest level. Moreover, it is possible to use repartitioning techniques such as those provided by the software package JOSTLE ([13]), to ensure that the partition before refinement is used as the basis for the

Total number of		% Imbalance	
Adaptivity cycles	Timesteps	Before Repartitioning	After Repartitioning
0	0	-	2.22
5	15	25.67	2.9
10	30	10.97	0.84
15	45	19.56	2.69
20	60	15.98	1.46

Table 2  
Partition imbalance for a mesh adapting to follow a shock

partition after refinement whenever possible. This has the further advantage that most of the background elements (and therefore the data for the sub-meshes within them) will remain on the same processor as they were before the adaptive step.

As an example of this, the mesh shown in Figure 3, which contains two levels of refinement beneath the background grid, is partitioned equally across 4 processors. As the solution evolves, the load balance of the initial partition is lost, as some of the coarse elements (to the right) refine further, whilst others derefine. Table 2 shows how the imbalance in the partitioning occurs as the solution evolves. Note that by using JOSTLE to repartition the weighted dual graph of the background grid, we can regain the balance, and hence the efficiency, of the partition. Moreover, the vast majority of the coarse elements (and their sub-mesh data) remain in the same memory locations as before the repartitioning.

This simple three-dimensional example again demonstrates that the approach of working mainly with the weighted dual graph of a background grid appears to have significant potential. There are still a number of issues associated with this which should be addressed, but the underlying approach seems to be both efficient and effective.

There are two main difficulties which are currently being investigated further. Firstly, after a very large number of adaptive steps it may be necessary to discard the present partition altogether and repartition the problem from scratch. This is most likely to occur if the background grid is excessively coarse or if extremely high levels of refinement are being used in small, localized regions. The other issue is that of whether the local repartitioning of the coarse background mesh can itself be efficiently implemented in parallel which is desirable from a scalability point of view.

## ACKNOWLEDGEMENTS

We would like to thank Philip Capon, Peter Dew, Bill Speares, Nasir Touheed and Chris Walshaw whose contributions and suggestions have all been most valuable. We are also grateful to Shell Research for funding and supporting the development of the 3-d adaptive code. The work of the first and third authors is funded by the EPSRC (under a postgraduate studentship and grant GR/J84919 respectively), and those calculations performed on the Intel i860 made use of the machine and support provided by Daresbury

Laboratory, UK (under EPSRC grant GR/J27066).

## REFERENCES

1. T. Arthur, 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. P. Diniz, S. Plimpton, B. Hendrickson, R. Leland *Parallel Algorithms for Dynamically Partitioning Unstructured Grids*. In Proc. of the Seventh SIAM Conference on Parallel Processing for Scientific Computing, ed. D.H. Bailey *et al* (SIAM), pp. 615 – 620, 1995.
3. B. Hendrickson, R. Leland. *An Improved Spectral Graph Partitioning Algorithm for Mapping Parallel Computations*. SIAM Jour. on Sci. Comp., Vol. 16, No. 2, pp 452-469, 1993
4. D.C. Hodgson, P.K. Jimack. *Efficient Mesh Partitioning for Parallel P.D.E. Solvers on Distributed Memory Machines*. In Proc. of the Sixth SIAM Conference on Parallel Processing for Scientific Computing, ed. R.F. Sincovec *et al* (SIAM), pp. 962 – 970, 1993.
5. D.C. Hodgson, P.K. Jimack. *Parallel Generation of Partitioned, Unstructured Meshes*. In Advances in Parallel and Vector Processing for Structural Mechanics, ed. B.H.V. Topping and M. Papadrakakis (Civil-Comp Press), pp. 147–157, 1994.
6. A.I. Khan, B.H.V. Topping. *Parallel Adaptive Mesh Generation*. Computing Systems in Engineering, Vol. 2, No. 1, pp. 75-101, 1991.
7. R. Löhner, J. Camberos, M. Merriam. *Parallel Unstructured Grid Generation*. Computer Methods in Apl. Mech. Eng., 95, pp. 343-357, 1992.
8. Message Passing Interface Forum. *MPI: A Message Passing Interface standard*. Int. J. of Supercomputer Applications, 8, 1994.
9. H.D. Simon. *Partitioning of Unstructured Problems for Parallel Processing*. Computing Systems in Engineering, Vol. 2, No. 2/3, pp. 135-148, 1991.
10. W. Speares, M. Berzins. *A Fast 3-D Unstructured Mesh Adaption Algorithm with Time-Dependent Upwind Euler Shock Diffraction Calculations*. In Proceedings of the 6th International Symposium on Computational Fluid Dynamics, pp 1191-1188, 1995.
11. A. Vidwans, Y. Kallinderis, V. Venkatakrishnan. *Parallel Dynamic Load-Balancing Algorithm for Three-Dimensional Adaptive Unstructured Grids*. AIAA Journal, Vol. 32, No. 3, pp. 497-505, 1994.
12. C.H. Walshaw, M. Berzins. *Dynamic Load-Balancing for PDE Solvers on Adaptive Unstructured Meshes*. Concurrency: Practice & Experience
13. C.H. Walshaw, M. Cross, S. Johnson, M.G. Everett. *JOSTLE: Partitioning of Unstructured Meshes for Massively Parallel Machines*. To appear in Proceedings of Parallel CFD '94, Kyoto.
14. N.P. Weatherill, O. Hassan. *Compressible Flowfield Solutions with Unstructured Grids Generated by Delauney Triangulation*. AIAA Journal, Vol. 33 No. 7, pp 1196-1204, 1995.