Parallel Computational Fluid Dynamics
Recent Developments and Advances Using Parallel Computers
D.R. Emerson, A. Ecer, J. Periaux, N. Satofuka and P. Fox (editors)

Parallel Dynamic Load-Balancing for the Solution of Transient CFD Problems Using Adaptive Tetrahedral Meshes

N. Touheed*, P. Selwood†, P.K. Jimack‡, M. Berzins and P.M. Dew [a]

[a]Computational PDE Unit,School of Computer Studies,
University of Leeds, Leeds LS2 9JT, UK

This paper considers a new parallel dynamic load-balancing algorithm which has been developed for use in conjunction with an unstructured tetrahedral parallel adaptive solver for transient flow problems. A brief description of the 3-d adaptivity algorithm is then followed by a discussion of the load-balancing problem. The practical load-balancing algorithm is then explained, followed by a discussion of its parallel implementation and an assessment of its performance on a transient shock problem.

1. INTRODUCTION

The use of distributed memory parallel computers for the solution of large, complex CFD problems has great potential for both significant increases in mesh sizes and the significant reduction of solution times. For transient problems accuracy and efficiency constraints also require the use of mesh adaptation since flow features on different length scales are likely to evolve. Significantly, the meshes that are generally used for these problems on parallel machines are typically too large for serial adaptivity to be viable: since this would cause a major serial bottleneck and would introduce a large communication overhead. In addition the size of the final mesh would be artificially constrained by the amount of memory available to a single processor. There is therefore a clear need for parallel adaptivity procedures to be supplied in addition to the parallel CFD solver. In this paper we focus on the production of such parallel adaptivity procedures, with particular emphasis on practical algorithms and routines for parallel dynamic load-balancing.

In Section 2 we present a brief overview of a parallel algorithm for the refinement and de-refinement of tetrahedral meshes which are distributed across the memory of a parallel architecture. In particular those features of the parallel adaptivity algorithm and its distributed data structures which are most relevant to the problem of ensuring load-balance in the parallel solver are described. Based upon this knowledge it is then possible to identify the desirable features of a mesh partitioning algorithm so as to ensure maximum efficiency of the parallel solver. This is done in Section 3 and in Section 4 the details of such an algorithm are described. The paper then finishes with a numerical example which is used to evaluate the algorithms that have been presented.

*Supported by the UK and Pakistan governments in the form of ORS and COTS scholarships respectively.
†Supported by EPSRC through research grant GR/J84915.
‡Use of the Cray T3D facility at the Edinburgh Parallel Computing Centre is gratefully acknowledged.

2. PARALLEL ADAPTIVITY IN 3-D

In this section we briefly describe a parallel implementation of the 3-d adaptivity code TETRAD (TETRahedral ADaptivity) described in [9]. This software is intended for use in the solution of time-dependent problems and is based upon the adaptive hierarchical refinement of an initial root mesh. A more detailed description of the parallel implementation using MPI ([7]) may be found in [8].

As a computed solution evolves with time the error on the current mesh, which will be a particular hierarchical refinement of the root mesh, will evolve too. Based upon some indication or estimate of this error some edges of the current mesh may be marked for either refinement or de-refinement (coarsening) at the end of a time-step. Elements with all edges marked for refinement are refined regularly into eight children (see Figure 1). Elements with between one and five refined edges are dealt with using so-called "green refinement", in which an extra node is created at the centroid of the element and joined to each of the surrounding nodes (the number of new elements created therefore depends upon the number of refined edges: see Figure 2 for an example with just one refined edge). Green elements are only used to create a link between regular elements of differing levels of refinement and may not themselves be refined (their parent element must be refined regularly instead). Coarsening of the mesh may only take place locally provided every edge in an entire family of sibling elements has been marked for de-refinement.

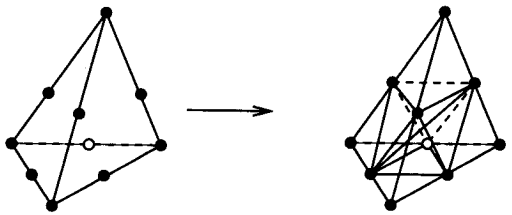

Figure 1. Regular refinement of a tetrahedron into eight children.

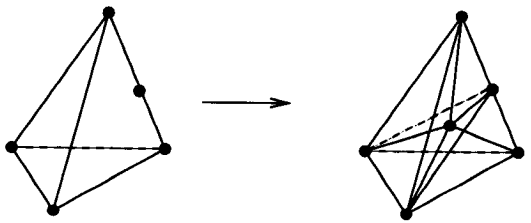

Figure 2. Green refinement by the addition of an interior node.

Full details of the parallel TETRAD algorithm and the parallel data structures that it uses are given in [8]. When focusing on the particular issue of load-balancing across a distributed memory parallel computer however only some of these details are important.

As indicated in Section 1, it is necessary to partition the hierarchically refined mesh across the processors. There are two main options for achieving this: either to partition at the root level and ensure that all non-root elements are in the same partition as their parent, or to partition the leaf mesh (which is the actual grid used for computation at any particular time). The advantage of the former is that all hierarchical operations (such as mesh de-refinement or multigrid V-cycles, for example) may be performed without communication since the entire mesh hierarchy for a given root element is stored on a single processor. This is the approach considered here, despite the fact that the partition quality (see Section 3 below) that one can achieve may be a little lower than that which is possible if the leaf mesh is partitioned directly.

The other key data concept that is used in the parallel TETRAD code is that of "halo" data. This is a common device used in parallel solvers (see [1] for example) in which a processor stores a copy of those elements (and other data) which are neighbours of one or more of its own elements but belong to another processor: the purpose being to minimise the amount of inter-processor communication that is required at each time-step. In the parallel adaptivity itself, the halo data also provides a mechanism for ensuring the consistency of the mesh across processor boundaries.

3. MESH PARTITIONING

Having briefly described the way in which a typical parallel adaptivity algorithm is implemented we now consider the details of how the mesh of tetrahedra should be partitioned at any time in the solution process. Note that in Section 2 it was decided that the mesh should be partitioned in such a way that all of the descendents of a root element should be located on the same processor as that element. Hence, in order to achieve load-balance in the parallel solver (assuming that the same number of degrees of freedom are present on each tetrahedron throughout the domain), the root mesh must be partitioned so that the sum of the weights of the root elements on each processor is approximately constant. (Here, the weight of a root element is defined to be the number of leaf elements contained within it.) In addition, it is highly desirable that the number of halo elements that each processor has to work with should be as small as possible (so as to minimise communication and parallel overheads). Thus the mesh partition must be such that the number of elements on the boundary between processors is minimised.

Partitioning a mesh subject to the above two constraints is a well-known problem; often referred to as the static load-balancing problem. For our time-dependent solver two additional constraints should also be respected however. These stem from the fact that when refinement takes place the mesh is already partitioned and so the load-balancing problem is in fact dynamic since it is desirable to respect the initial location of each root element when deciding upon a new partition of the mesh. In particular we would like to both keep the number of root elements migrated between processors as small as possible and to implement what migration is required in parallel so as to minimise the overhead associated with maintaining a good partition.

In summary, an ideal dynamic load-balancing algorithm for updating the partition of the root mesh after adaptivity has occurred should:

I. Equally distribute the number of leaf elements on each processor.

II. Minimise the number of elements on the inter-processor partition boundary.

III. Maintain locality of each of the root elements as much as possible.

IV. Have an efficient parallel implementation.

In the next section these properties are considered in the context of dynamically partitioning the weighted dual graph of the root mesh. In this graph the weight of node k, w_{N_k} say, is equal to the weight of the root element to which it corresponds, and the weight of edge (k, ℓ), $w_{E_{k\ell}}$ say, is equal to the number of leaf element faces which are shared by root elements k and ℓ: the nodes at the end of the edge.

4. PARALLEL DYNAMIC LOAD-BALANCING

It is clear that the four requirements enumerated in the last section are not always self-consistent. It is perhaps for this reason that quite a large number of dynamic load-balancing heuristics have been suggested in recent years ([2,5,10,11] for example), each of which appear to put a slightly different emphasis on the relative importance of the four properties. The algorithm described in this section explicitly attempts to respect *all* of these requirements; however when conflicts do arise it is items III and IV, which relate more to the parallel overhead than the partition quality, which are the first to be relaxed. The motivation behind this is our decision that, when one is forced to choose between the two, robustness is more important than parallel efficiency in a parallel dynamic load-balancing algorithm.

4.1. Processor groups

The first stage of our dynamic load-balancing algorithm is to split the processors into two groups based upon the initial partition of the latest mesh. This is achieved by partitioning a processor graph, frequently referred to as the weighted partition communication graph (WPCG) (see [10] for example). This graph has one vertex for each processor, whose weight is equal to the sum of the weights of all root elements on that processor, and an edge connecting two nodes if they are face adjacent (i.e. the corresponding processors share part of the partition boundary). The weight of the edge connecting nodes i and j is equal to the total number of leaf element faces shared by these two processors.

When splitting the WCPG into two groups it is desirable to take into account constraints 1 and 2 from the previous section. In particular, we should ensure that the two groups have a similar total weight and that the total weight of edges of the WPCG which are cut by the division is as low as possible. This is perhaps best achieved through using a spectral algorithm and is the approach followed here. This is inexpensive because the size of the WPCG is only equal to the number of processors being used (details of a version of the algorithm suitable for weighted graphs may be found in [4]). The spectral algorithm provides an ordering of the nodes of the WPCG (i.e. the processors) and this list is then split at the point which comes closest to balancing the weight of the two groups.

At this stage there are two processor groups of approximately equal total weight, with a relatively short boundary between them. If the initial partition of the weighted dual graph of the root mesh is well load-balanced then the number of processors in each of the two groups will be equal. If the initial load-balance is poor however then it is likely

that there will be a different number of processors in each group and therefore that the average weight per processor in each group will be different. Hence, the next phase of the algorithm is to use local migration of root elements in order to balance the average weight per processor for the two groups.

4.2. Balancing the groups with local migration

We will refer to the group with the larger average weight per processor as the "Sender" group and the other as the "Receiver" group. If the difference in the average weights is greater than some tolerance then this stage of the algorithm requires the groups to be balanced by migrating nodes of the weighted dual graph (i.e. root elements). In order to decide which nodes to migrate we apply the ideas of Fiduccia and Mattheyses ([3]) whose algorithm is, in turn, developed from the well-known local migration strategy of Kernighan and Lin ([6]).

We first calculate the total weight, Mg_{tot} say, of all of the nodes to be migrated from the Sender to the Receiver. Let N_S and N_R be the number of processors in each of these groups and let their average weights per processor be denoted by Av_S and Av_R respectively. Then $\mathrm{Mg}_{tot} = N_S \times (\mathrm{Av}_S - \mathrm{Av})$, where Av is the overall average weight per processor in the WPCG.

Having established the required load to be transferred, the next issue to address is that of how many nodes each processor in the Sender group should actually send and which processors in the receiver group they should be sent to. Here we follow the approach taken in [10], by introducing the concept of candidate processors: these are processors in either group which are face-adjacent to at least one processor in the other. Where possible we only permit these processors to be involved in the migration of nodes, and if the i^{th} candidate processor in the Sender group is face adjacent to more than one candidate processor in the Receiver group then we only migrate nodes to that candidate processor which has the "longest" common boundary (i.e. which shares the most leaf element faces with the i^{th} candidate processor in the Sender group). Let N_{tot} be the total weight on all candidate processors of the Sender group. Then the target weight to be transferred from the i^{th} candidate processor in Sender group, Mg_i say, is given by

$$\mathrm{Mg}_i = \left(\frac{N_i}{N_{tot}}\right) \times \mathrm{Mg}_{tot} \ ,$$

where N_i is the total weight on the i^{th} processor.

We are now able to determine what weight, if any, should be transferred between each pair of processors. The final step in this phase of the algorithm therefore is to decide precisely which root elements should be migrated; our aim being to transfer those elements which lead to the smallest possible number of faces on the final partition boundary. In order to achieve this we define the concept of the "gain" associated with transferring a particular node in the weighted dual graph of the root mesh, node k say, from its current subgraph to another (from node i of the WPCG to node j say). This is given by

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

where the summation is over all edges, (k, ℓ), of the weighted dual graph which radiate from node k. A slight extension of this definition is the "gain density" of a node which is defined to be equal to $\text{gain}(k)/w_{N_k}$. In order to keep the number of leaf elements on the final inter-partition boundary as small as possible we always select the nodes with the highest gain density as the ones to migrate. (After each root element is migrated the gain of each of its neighbours may easily be updated so this selection should be sequential.)

4.3. Recursion and parallel implementation

The previous two subsections describe how two groups of processors may be created and then balanced in such a way as to keep the boundary between them as short as possible. The final stage of the algorithm is to apply the same steps again, recursively, on each of these two processor groups. The termination criterion for the recursion is when a group consists of a single processor. At this point the algorithm will have ensured that the total weight on each processor is approximately equal.

Clearly, as the depth of recursion increases the level of parallelism in the algorithm increases too. In order to further increase the parallel efficiency however, we do not transfer the complete mesh data at each level of the algorithm, but transfer only enough information about the weighted graph to allow the new weights and gains to be calculated so that the algorithm may proceed. The full transfer of mesh data may then be carried out concurrently at the end, once the final destination of each root element is known. This has the additional advantage of reducing the overall amount of communication that is required since elements do not need to be sent to intermediate processors as they might otherwise be if the partition were to be fully updated after each level of the recursion.

Before finishing this section on the dynamic load-balancing algorithm it is worth considering how it is likely to behave in one or two particular cases. If the mesh is already well load-balanced then it will not be altered at all because the value of Mg_{tot} will always be less than the tolerance that has been set. If the load-imbalance is small and uniformly distributed then only a small amount of migration will take place at each level – improving the load-balance whilst maintaining most of the data locality (moreover those elements which are transferred will always be the ones which keep the partition boundary as short as possible). In the case where we start with a very heavy imbalance however, it is impossible to maintain so much data locality. Hence the above algorithm will require a significantly greater number of elements to be transferred at each level. Moreover, the recursion tree will become quite highly imbalanced due to the groups at each level having large differences in the number of processors that they contain. Hence the parallel efficiency within the load-balancing is likely to decrease for such an extreme case. Nevertheless, it is to be expected that the quality of the new partition will be a great improvement over that of the original, thus making this extra overhead very worthwhile.

5. COMPUTATIONAL RESULTS

This section illustrates the performance of this dynamic load-balancing algorithm when used in conjunction with the parallel adaptive refinement algorithm described in Section 2. The implementation has been completed using MPI [7]: with significant use of the group operations that it supports. At each level, the spectral ordering of the processors in each group is found sequentially using just one of the processors within the group.

The test problem used is the time-dependent gas dynamics problem studied in [9]. This is an inviscid Euler flow calculation with ideal gas equations of state modelling a shock wave diffraction around the 3D right-angled corner formed between two cuboid mesh regions. The initial condition is of Rankine-Hugoniot shock data at the interface of the two cuboid regions with and a shock speed of Mach 1.7.

The solution is computed with a cell-centred, Riemann problem based, finite volume scheme of the MUSCL type, employing an HLLC style approximate Riemann solver. For full details of the scheme, see [9]. The parallel version of the solver uses the standard 'owner computes' rule (e.g. see Caballo [1]) and halo data updates occur after both the Hancock and full time-steps are completed.

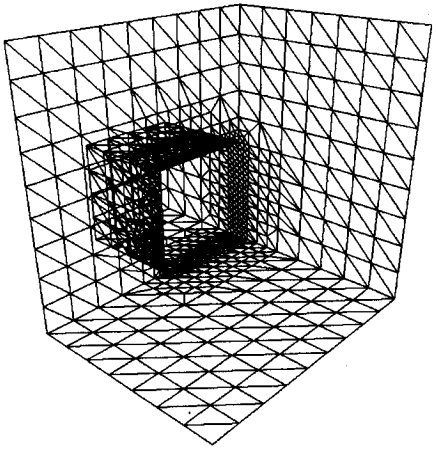

Figure 3. Coarse mesh of 5,184 elements adapted to initial shock condition.

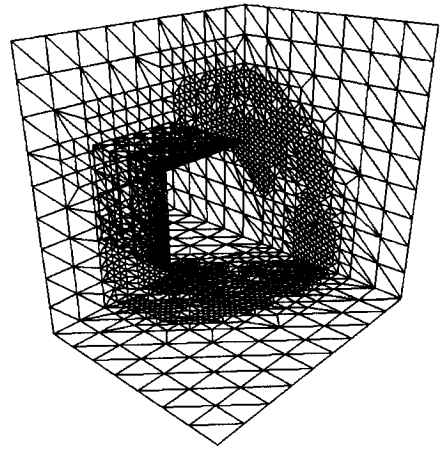

Figure 4. Adapted mesh after 240 time-steps

Figures 3 and 4 illustrate how the mesh adapts to the solution as the shock progresses through the domain. It is clear that although a partition of the mesh for the initial condition may be good, it is unlikely to remain so as the solution develops and thus dynamic load-balancing of the distributed data will be required.

For this example, we consider a finer initial mesh than the illustrative examples shown in Figures 3 and 4. Our coarse mesh has 34,560 elements which results in a mesh of 87,970 elements after adapting to the initial condition. Two levels of adaptation are used. As the shock progresses into the domain, the mesh adapts accordingly so that after 60 time-steps there are 106,772 elements. Note that throughout this calculation, the adaptive mesh has resolution equivalent to a mesh of 2.2 million uniform, regular elements.

An experiment using 8 processors of the Cray T3D has been conducted. The initial partitioning of the mesh uses approximate weights obtained from knowledge of the initial condition and is performed using Jostle V1.0 [11]. As the weights are only approximate, there is a 31% imbalance after the first adaptation stage. As the solution progresses

however, this improves to 17% before degrading again. After 60 time-steps, the imbalance has reached 37% and it is clear that repartitioning would be advantageous.

Repartitioning the mesh restores the balance to 9% although this comes at a slight cost to the cut weight which increases from 4303 to 5246 edges cut. The benefit however is quite clear. The 30 time-steps following the repartitioning take only 429.1 wall-clock seconds as opposed to the 501.5 seconds taken if repartitioning is not employed. This is a significant saving, and demonstrates the effectiveness of our repartitioning approach.

6. CONCLUSIONS

A new parallel load-balancing algorithm has been presented for use with unstructured adaptive mesh calculations of transient flows on distributed memory computers. An Euler flow example has been given which illustrates that the algorithm produces new partitions that significantly improve the efficiency of the solver.

REFERENCES

1. J. Cabello. *Parallel Explicit Unstructured Grid Solvers on Distributed Memory Computers.* Advances in Eng. Software, 23, 189 (1996).
2. G. Cybenko. *Dynamic Load Balancing for Distributed Memory Multiprocessors.*, J. of Parallel and Distributed Computing, 7, 279 (1989).
3. C.M. Fiduccia and R.M. Mattheyses. *A Linear Time Heuristic for Improving Network Partitions.* Proc. of the 19th IEEE Design Automation Conference, IEEE, p175 (1982).
4. D.C. Hodgson and P.K. Jimack. *Efficient Parallel Generation of Partitioned, Unstructured Meshes.* Advances in Eng. Software, 27, 59 (1996).
5. 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).
6. B. Kernighan and S. Lin. *An Efficient Heuristic Procedure for Partitioning Graphs.* Bell System Technical Journal, 29, 209 (1970).
7. Message passing Interface Forum. *MPI: A Message Passing Interface Standard.* Int. J. of Supercomputer Applications, 8, no. 3/4 (1994).
8. P.M. Selwood, M. Berzins and P.M. Dew. *3D Parallel Mesh Adaptivity: Data-Structures and Algorithms.* In proc. of Eighth SIAM Conf. on Parallel Proc. for Scientific Computing (1997), SIAM Philadelphia.
9. W. Speares and M. Berzins. *A 3-D Unstructured Mesh Adaptation Algorithm for Time-Dependent Shock Dominated Problems.* Int. J. Num. Meth. in Fluids, 25, 81 (1997).
10. A. Vidwans, Y. Kallinderis and V. Venkatakrishnan. *Parallel Dynamic Load-Balancing Algorithm for Three-Dimensional Adaptive Unstructured Grids.* AIAA Journal, 32, 497 (1994).
11. C. Walshaw, M. Cross and M.G. Everett. *Dynamic Load-Balancing for Parallel Adaptive Unstructured Meshes.* In proc. of Eighth SIAM Conf. on Parallel Proc. for Scientific Computing (1997), SIAM Philadelphia.