A Comparison of Some Dynamic Load-Balancing Algorithms for a Parallel Adaptive Flow Solver

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

Abstract

In this paper we contrast the performance of three different parallel dynamic load-balancing algorithms when used in conjunction with a particular parallel, adaptive, time-dependent, 3-d flow solver that has recently been developed at Leeds. An overview of this adaptive solver is given along with a description of a new dynamic load-balancing algorithm. The effectiveness of this algorithm is then assessed when it is coupled with the solver to tackle a model 3-d flow problem in parallel. Two alternative parallel dynamic load-balancing algorithms are also described and tested on the same flow problem.

1 Introduction

The use of distributed memory parallel computers for the solution of large, complex computational mechanics 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 solution 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 in conjunction with a parallel solver without causing a major serial bottleneck and 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 solver itself. This adaptivity should allow both the addition and deletion of degrees of freedom across the solution domain in a distributed manner, without ever requiring the entire mesh to be held on a single processor – see [10] for a discussion of some examples of such techniques.

In order for the parallel solver to perform efficiently however it is necessary that, at each stage of the solution process, the work load of each processor should be about equal (or proportional to its computational power in the case of an heterogeneous system). If this equality of load is initially achieved through appropriately partitioning the original finite element/volume mesh across the processors then it is clear that the use of parallel adaptivity will eventually cause the quality of the partition to deteriorate. For the same reasons that it is undesirable to perform mesh adaptivity on a single processor it is also undesirable to re-partition the mesh using just one processor: it would carry a large communication overhead, become a serial bottleneck and would be constrained by the amount of memory available to just one processor. Hence a parallel load-balancing technique is required which is capable of modifying an existing partition in a distributed manner so as to improve the quality of the partition (see Section 3) whilst keeping the amount of data relocation as small as possible.

In this work we consider the dynamic load balancing problem which arises in the adaptive solution of time-dependent partial differential equations using a particular parallel adaptive algorithm based upon hierarchical mesh refinement. This algorithm is applicable to problems in *three* space dimensions of the form

$$\frac{\partial \underline{u}}{\partial t}(\underline{x}, t) = \mathcal{L}(\underline{u}(\underline{x}, t)) \qquad \text{for } (\underline{x}, t) \in \Omega \times (0, T], \quad (1)$$

where $\Omega \in \Re^3$ and $\mathcal{L}$ is some spatial operator. It is based upon the adaptive refinement of a coarse root mesh, $\mathcal{T}_0$ say, of tetrahedra which covers the spatial domain Ω. The flexibility of the data structures held within the adaptivity code (see 2.1 below) means that the exact nature of the parallel solver may vary (e.g. finite element or finite volume) provided it uses a tetrahedral mesh and is able to work with a partition of the elements of this mesh.

In the following section an overview of this parallel adaptive algorithm is given, along with a brief description of a particular parallel solver based upon a cell-centred finite volume scheme. Section 3 then discusses the dynamic load-balancing problem in more detail and introduces two very recent software tools for tackling this distributed problem in parallel. A third dynamic load-balancing algorithm is introduced in Section 4, where its implementation is also outlined. The paper then concludes by reporting the results of a number of numerical tests which are used to contrast the three load-balancing algorithms for this particular adaptive solver.

To conclude this introductory section we observe that the parallel dynamic load-balancing problem addressed in this paper can arise in numerous other contexts in parallel computational mechanics. As well as the use of local h-refinement, other algorithms which permit the distri-

bution of the computational load across the domain Ω to vary as the simulation proceeds will require a dynamic load-balancing strategy. This includes algorithms based upon p-refinement (see [3, 4] for example) or those for solving systems, such as those arising in phase-change problems for example (e.g. [1]), in which the computational nature of the solution can change with time at each point in Ω. Another important situation in which dynamic load-balancing is required often arises when a parallel code is executed on an heterogeneous network (such as a cluster of workstations for example), in which the performance of each processor may vary with time as its load increases or decreases. All of these situations may be treated by the algorithms discussed in Sections 3 and 4 below however, for the sake of clarity, we now restrict all further discussion, examples and comparisons to the three-dimensional h-refinement code described in the following section.

2 A Parallel Adaptive Flow Solver

2.1 A parallel adaptive algorithm

The software outlined in this subsection is based upon a parallel implementation of a general purpose serial code, TETRAD (TETRahedral ADaptivity), for the adaptation of unstructured tetrahedral meshes [20]. The technique used is that of local refinements/derefinements of the mesh to ensure sufficient density of the approximation space throughout the spatial domain, Ω, at all times. A more complete discussion of the parallel algorithms and data structures may be found in [17, 18, 19].

Data structures

One of the major issues involved in parallelising an adaptive code such as TETRAD is how to treat the existing data-structures. TETRAD utilises a complex tree-based hierarchical mesh structure, with a rich interconnection between mesh objects. Figure 1 indicates the mesh object structures used in TETRAD. In particular, note that the main connectivity information used is 'node to element' and that a complete mesh hierarchy is maintained by both element and edge trees. Furthermore, as the meshes are unstructured, there is no way of knowing *a-priori* how many elements share any given edge or node.

For parallelisation of TETRAD, there are two main data-structure issues. The first is how to partition a hierarchical mesh, the second is that we require new data-structures to support parallel partitioning of the mesh.

1. There are two main options for partitioning a hierarchical mesh. The first is to partition the grid at the root or coarsest level, $\mathcal{T}_0$. This has a number of advantages. The local hierarchy is maintained on a processor and thus all parent/child interactions (such as refinement/derefinement) are local to a processor. The partitioning cost will also be low, as the coarse mesh is generally quite small. The main disadvantage of this approach however is that, for comparatively small coarse meshes with large amounts of refinement, it may be difficult to get a good partitioning, both in terms of load balance and communication requirements.

   The other main approach is to partition the leaf-level mesh, i.e. the actual computational grid. The pros and cons of this approach are the opposite of those with the coarse level partitioning. In particular, the quality in terms of load balance and cut-weight of the partition is likely to be better, albeit at the expense of a longer partitioning time. However, the data-structures have to be more complicated as hierarchical operations, such as multigrid V-cycles and derefinement for example, are no longer necessarily local to a processor (and are therefore likely to be slower).

   The approach taken for parallelising TETRAD is that of partitioning the coarse mesh. The only disadvantage of this, that of possible suboptimal partition quality, can be avoided if the initial, coarse mesh is scaled as one adds more processors.

2. Given a partitioned mesh, we also need new data-structures in order to support inter-processor communication and to ensure data consistency. Data consistency is handled by assigning ownership of mesh objects (elements, faces, edges and nodes). As is common in many solvers such as those used by [2] we use halo elements, a copy of inter-processor boundary elements (with their associated data) used to reduce communication overheads. In order to have complete data-structures (e.g. elements have locally held nodes) on each processor, we also have halo copies of edge, node and face objects. If a mesh ob-

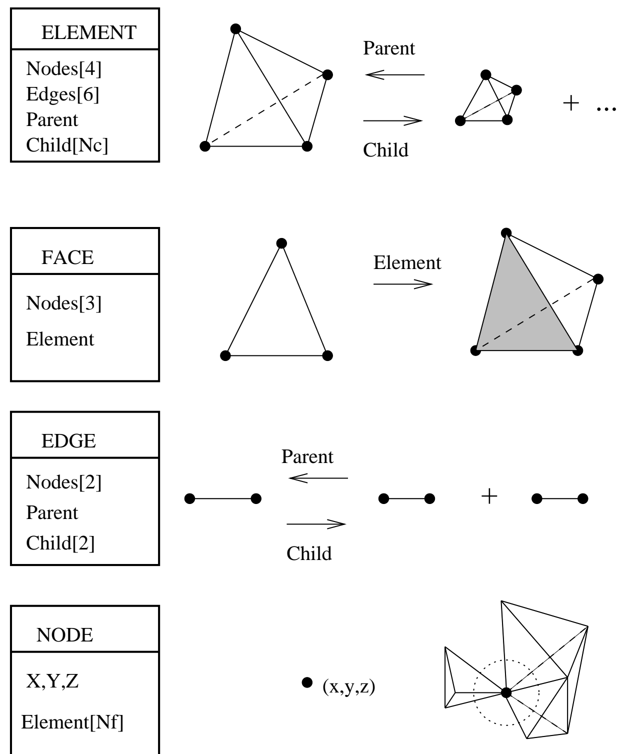

Figure 1: Mesh Data-Structures in TETRAD

ject shares a boundary with many processors, it may have a halo copy on each of these. All halos have the same owner as the original mesh object. In situations where halos may have different data than the original, the original is used to overwrite the halo copies and thus is definitive. This is used to help prevent inconsistency between the various copies of data held.

Adaptivity algorithms

Both TETRAD ([20]) and its parallel implementation, PTETRAD ([17]), use a similar strategy to that outlined in [14] to perform adaptivity. Edges are first marked for refinement/derefinement (or neither) according to some estimate or indicator (provided as part of the parallel solver: see 2.2 below for example). Elements with all edges marked for refinement may then be refined regularly into eight children. To deal with the remaining elements which have one or more edge to be refined we use so-called "green" refinement. This places an extra node at the centroid of each element and is used to provide a link between regular elements of differing levels of refinement. The types of refinement are illustrated in Figures 2 and 3. An important restriction that is made is that green elements may not be further refined as this may adversely affect mesh quality ([16]). Instead, they are first removed and then uniform refinement applied to the parent element.

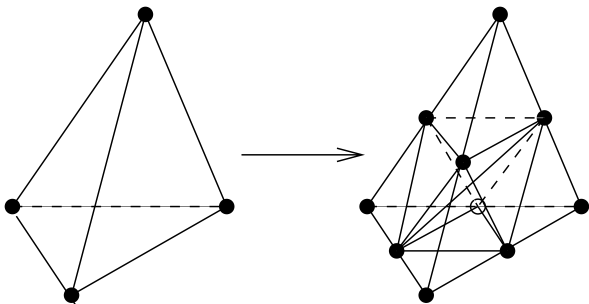

Figure 2: Regular Refinement dissecting interior diagonal

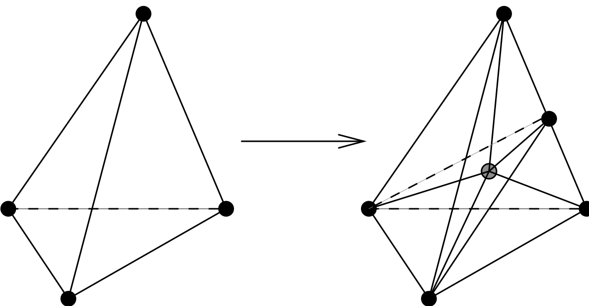

Figure 3: Green Refinement by the addition of an interior node

Immediately before the refinement of a mesh, the derefinement stage occurs. This may only take place when all edges of all children of an element are marked for derefinement and when none of the neighbours of an element to be deleted are green elements or have edges which have been marked for refinement. This is to prevent the deleted elements immediately being generated again at the refinement stage which follows. A further necessary constraint is that no edges or elements at the coarsest level, $\mathcal{T}_0$, may be derefined.

For further details of the implementation of these adaptive algorithms using MPI ([15]) please refer to [17]. This paper discusses important issues such as performing parallel searches in order to allow refinement of edges of green elements (which requires coarsening followed by regular refinement), maintaining mesh consistency and dealing with halo data in parallel.

2.2 A parallel finite volume solver

In order to apply the above adaptive algorithm to systems of PDEs of the form (1) a parallel solver is also required. The data structures supported by TETRAD have been used with both finite element and finite volume solvers (cell-centred and cell-vertex), however in this paper we restrict our numerical experiments to a cell-centred finite volume scheme.

The scheme that we use is applicable when (1) represents a system of hyperbolic conservation laws of the form

$$\frac{\partial \underline{u}}{\partial t} + \frac{\partial \underline{F}(\underline{u})}{\partial x} + \frac{\partial \underline{G}(\underline{u})}{\partial y} + \frac{\partial \underline{H}(\underline{u})}{\partial z} = 0 \,, \qquad (2)$$

such as the three-dimensional Euler equations for example, and is a parallel version of the algorithm described in detail [20]. This is a conservative cell-centred scheme which is a second-order extension of Gudunov's Riemann problem-based scheme ([6]), using MUSCL-type piecewise linear reconstructions of the primitive variables within each element ([13]). Although the time-stepping is explicit it is executed in two distinct phases: a non-conservative predictor-type update (referred to in [13] as the "Hancock step") followed by a second half-time-step based upon the application of the underlying conservation law. Implicit in this numerical method is the need to solve a Riemann problem at each element interface at each time-step – although this is only done approximately using a modified form of the approximate solver described in [7].

The parallel version of the solver is straightforward to implement due to the face data structure that exists within the adaptivity software (see Figure 1 for example). To avoid any conflicts at the boundary between two subdomains a standard "owner computes" rule is used for each of the faces when solving the approximate Riemann problems to determine fluxes. The use of halo elements ensures that the owner of each face has a copy of all of the data required to complete these flux calculations provided the halo data is updated twice for each time-step (i.e. immediately before the Hancock step and then again before the second half-time-step).

3 Dynamic Load-Balancing

As explained in Section 2.1 above parallel solvers such as PTETRAD require the computational domain to be

partitioned into subdomains. In the case of parallel TETRAD this partition should be applied to the coarse root mesh $\mathcal{T}_0$. It is usual to express the requirements of such a partition in terms of the weighted dual graph of this mesh. For each element, i, of the root mesh define a corresponding vertex of the dual graph and let this vertex have weight v_i, where v_i is the number of leaf-level elements of the current mesh which lie within root element i. For each pair of face adjacent elements in the root mesh define an edge, j, of the dual graph and let this edge have weight e_j, where e_j is the number of pairs of leaf-level elements in the current mesh which meet along face j. We may now observe that, for a homogeneous network of processors, we would like to be able to partition this graph so that at all times

1. the total vertex weight in each sub-graph is approximately equal.

This will ensure that the computational load will be about the same on each processor when the parallel finite volume solver is applied. In addition to this however, we would like the number of halo elements to be kept to as low as possible so as to minimise the overhead of the halo updates that are required twice per time-step. This means that we also need to partition the weighted dual graph so that at all times

2. the total cut-weight of the partition is kept to a minimum.

Here, the term cut-weight is defined to be the sum of the edge weights of those edges which link two vertices of the dual graph belonging to different sub-graphs within the partition. (Note that an alternative requirement could be to minimise the maximum cut-weight for any sub-graph as opposed to the total cut-weight, but we will not consider this problem here.)

Note that both of the above constraints on the partition of $\mathcal{T}_0$ (or its dual graph) should hold at each time-step. However, when parallel adaptivity occurs it is likely that the weights v_i and e_j will change. In particular, changes in the vertex weights v_i are liable to cause an existing well-balanced partition of $\mathcal{T}_0$ to become unbalanced. The objective of a dynamic load-balancing algorithm is to modify an existing, inadequate, partition of the dual graph so as to meet objectives 1 and 2 above but in such a way that

3. there is a minimal amount of migration of data between sub-graphs.

The motivation behind this requirement is simply that there is a significant communication overhead associated with moving data between processors and this overhead should not be allowed to nullify the computational advantages of obtaining an improved partition. A similar argument motivates yet another requirement that:

4. the load-balancing should be completed in parallel.

If this were not to be the case then there would be a sequential bottleneck in the whole solution procedure at the load-balancing stage which could seriously reduce the overall efficiency and performance of the adaptive parallel solver.

Before moving on to briefly discuss some parallel dynamic load-balancing algorithms it should be noted that there is no reason to expect that 1 to 4 above represent a consistent set of requirements. For example, an existing partition could be so far from optimal that there is no way that a near-optimal partition (in terms of 1 and 2) can be reached by only migrating a small proportion of the vertex weights. It is perhaps not surprising therefore that the vast majority of published dynamic load-balancing algorithms appear to be based heavily on heuristics. A recent survey of such algorithms may be found in [9] however in this section we will describe just two of these, called "Metis" ([11]) and "Jostle" ([25]) respectively. These have been selected since, at the time of writing, they are the only two parallel dynamic load-balancing algorithms which have public domain implementations: in both cases using MPI [15].

Both Metis ([11]) and Jostle ([25]) are examples of multilevel partitioning algorithms. The general idea behind these techniques is to produce a hierarchy of coarsenings of the original weighted graph (where each level in the hierarchy is produced by merging together groups of neighbouring vertices of the graph (usually from the same subgraph) at the previous level), and then to perform a quite careful re-partition of the coarsest graph. This new partition is then projected onto the graph at the previous level and then modified using a local algorithm (such as [5, 12]) in order to improve the partition quality. This step of projection onto the previous level followed by local improvement is repeated until the original graph has been recovered, when the algorithm terminates. Further details of each of these algorithms may be found in the papers cited.

4 An Alternative Dynamic Load-Balancing Algorithm

In this section we introduce a third dynamic load-balancing algorithm in some detail. This was first described in [21, 22] although some minor modifications have since been made for its application to the three-dimensional problems considered here.

4.1 Group balancing

Following Vidwans *et al.* [23], 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 face 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 faces on the interpartition boundary between the two processors.

We now divide the WPCG into two subgroups denoted by Group1 and Group2. Unlike in [23] however we use a weighted version of the spectral bisection method 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 [8, 24]. 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 $S = D^T LD$ is then found. Finally a partitioning vector, $\underline{\mathrm{x}}$, is defined by $x_i = u_{2_i}/\sqrt{w_{N_i}}$ 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

$$\left| \sum_{i=1}^{n} w_{N_i}^{'} - \sum_{i=n+1}^{p} w_{N_i}^{'} \right| \tag{3}$$

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

If the latest leaf-level 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 this mesh is the result of heavy local refinement in just some regions of Ω 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.

4.2 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. elements of the coarse mesh $\mathcal{T}_0$ and their entire sub-mesh hierarchies) 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 ([12]), we apply the ideas of Fiduccia and Mattheyses ([5]) 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 4 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.

```
if(Ave_1 ≤ Ave_2){
    Sender = Group2;
    Receiver = Group1;
    Mig_tot = N_2 * (Ave_2 − Ave);
}
else{
    Sender = Group1;
    Receiver = Group2;
    Mig_tot = N_1 * (Ave_1 − Ave);
}
```

Figure 4: Calculation of Sender, Receiver and Mig_{tot}.

Having established the required load to be transferred, the next issue to address is that of how many nodes (i.e. elements of $\mathcal{T}_0$) 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* [23], 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 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,

$$\text{Mig}_i = \left(\frac{N_i}{N_{tot}}\right) * \text{Mig}_{tot}, \tag{4}$$

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 root mesh $\mathcal{T}_0$ 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 5.

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

Figure 5: The calculation of gain.

The gain density of a node of a weighted graph 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 four neighbours when we are working with tetrahedralisations of three-dimensional domains) using the algorithm outlined in Figure 6. 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 n_k which is a neighbour of the node k {
   Let p_k be the processor to which n_k belongs;
   if (p_k == j) then
      decrement gain(n_k) by 2*w_{E_{n_k k}};
   else if(p_k == i) then
      increment gain(n_k) by 2*w_{E_{n_k k}};
}
```

Figure 6: Updating the gains.

4.3 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. In practice the communication of the full coarse element hierarchies is left until the end of the load-balancing process, with much smaller tokens being passed instead at this stage.

The implementation of this load-balancing algorithm that is used for the numerical experiments described in Section 5 was completed using the MPI library ([15]). 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. An overview of the whole algorithm is given in Figure 7.

5 Some Computational Comparisons

In this section we contrast the two dynamic load-balancing algorithms introduced in Section 3 and the new algorithm introduced in Section 4 when used in conjunction with the parallel adaptive flow solver outlined in Section 2. It should be noted that this flow solver requires a partition of the root mesh, $\mathcal{T}_0$, such that the total number of leaf-level elements on each processor is approximately equal. When there is heavy local refinement in some regions of the spatial domain Ω (as in the examples below) it follows that the dual graph of $\mathcal{T}_0$ will have highly disparate weights. Hence, in this paper we are only testing the performance of the dynamic load-balancing algorithms for one specific class of problem: the repartitioning of highly non-uniformly weighted graphs.

5.1 Examples

For these examples we apply our parallel adaptive Euler solver to model a shock wave diffraction around the 3D right-angled corner formed between two cuboid mesh regions (taken from [17, 20]). The initial condition is of

```
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 7: Parallel post-processing algorithm.

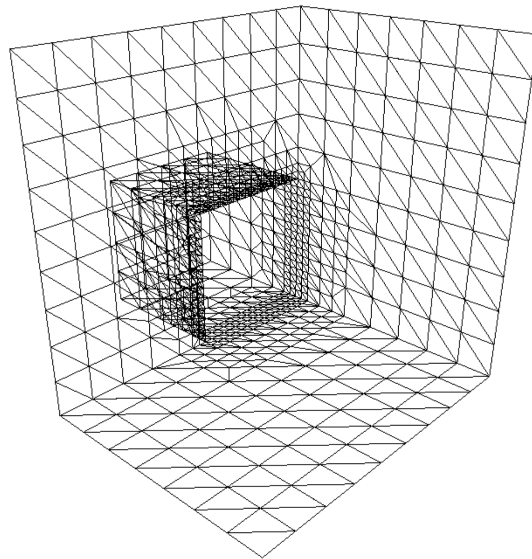

Figure 8: Coarse mesh refined to capture initial shock.

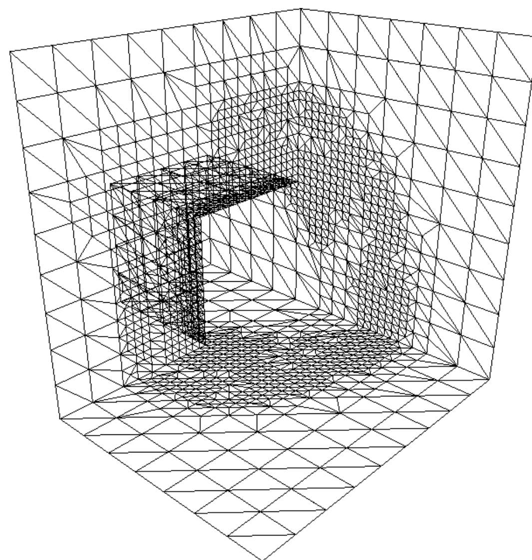

Figure 9: Adapted mesh after 240 time-steps.

Rankine-Hugoniot shock data at the interface of the two cuboid regions with a shock speed of Mach 1.7. Figures 8 and 9 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. In the computations whose results are tabulated below the root mesh, $\mathcal{T}_0$, contained $34,560$ elements (this is a little more than for the illustrative examples shown in Figures 8 and 9). Up to two levels of refinement are allowed which leads to an initial fine mesh containing $87,970$ elements with many more elements appearing in this leaf-level mesh at later times (e.g. $106,772$ elements after 60 time-steps). Note that throughout these calculations, the adaptive mesh has resolution equivalent to a mesh of 2.2 million uniform, regular elements.

Table 1 presents a comparison of some partition-quality metrics when the three different load-balancing algorithms are applied using 4, 8, 16 and 32 processors of a Cray T3D computer. In each case the initial partition has a maximum imbalance of over 20% (this is the percentage by which the total vertex weight of the most heavily-weighted sub-graph exceeds the average weight of the sub-graphs) and the cut-weight is given. The solution times quoted represent the wall-clock time (in seconds) taken by the parallel finite volume solver for the next three time-steps, either using the initial partition or using a new partition after application of one of the load-balancing algorithms. Finally, when load-balancing has been performed, the total weight of all of the root elements of $\mathcal{T}_0$ that have been migrated from one processor to another is quoted. It should be noted that for these, and all of the other calculations described here, Jostle was used with its graph reduction threshold parameter set to 300. This value appeared to give consistently better results than either of the other values tried: 20 (the default) and 50. All other parameters in both Jostle and Metis were left at their default values.

An alternative form of comparison between the three load-balancing algorithms is provided by Tables 2 and 3. For these results sequences of thirty time-steps were taken with adaptivity taking place on ten occasions (after every three time-steps). Whenever the maximum imbalance exceeds a prescribed tolerance (2.5% for Table 2 and 10% for

	Initial	Metis	Jostle	Recursive
Processors	4			
Imbalance	26%	5%	0%	0%
Cut-Weight	1851	1736	1736	2429
Sol. Time	24.5	20.0	19.0	19.0
Migration	–	8965	13225	7606
Processors	8			
Imbalance	21%	5%	1%	0%
Cut-Weight	4075	3681	3141	5032
Sol. Time	12.5	10.9	10.1	10.5
Migration	–	7196	6345	10077
Processors	16			
Imbalance	20%	5%	2%	1%
Cut-Weight	5397	5331	4964	7024
Sol. Time	6.4	5.5	5.3	5.6
Migration	–	8042	28551	12317
Processors	32			
Imbalance	31%	6%	5%	2%
Cut-Weight	7432	7258	7638	9491
Sol. Time	3.6	2.9	2.9	3.0
Migration	–	13007	36153	16081

Table 1: Some partition-quality metrics immediately before and after a single re-balancing step.

Table 3) after mesh adaptivity has taken place the load-balancing algorithm is called. The solution times quoted are the total times for the finite volume solver to complete the 30 time-steps *excluding* the repartitioning times. This gives an indication of the quality of the load-balancing algorithms. In order to compare their overheads the tables also show the total weight of all of the root elements that were migrated throughout the thirty time-steps taken in each run.

	Metis	Jostle	Recursive
Processors	4		
Sol. Time	207.9	200.5	202.7
Migration	16015	34451	15576
Processors	8		
Sol. Time	108.7	102.8	108.2
Migration	47671	77054	27170
Processors	16		
Sol. Time	56.0	53.7	56.4
Migration	21339	111174	39162
Processors	32		
Sol. Time	29.2	28.5	31.3
Migration	48772	202385	44684

Table 2: Solution times and total migration weights for 30 time-steps using a re-balancing tolerance of 2.5%.

	Metis	Jostle	Recursive
Processors	4		
Sol. Time	211.2	210.0	211.3
Migration	14060	13225	13389
Processors	8		
Sol. Time	110.9	106.5	108.7
Migration	19864	28841	23019
Processors	16		
Sol. Time	56.4	55.3	57.8
Migration	39511	58621	33401
Processors	32		
Sol. Time	29.1	30.1	30.8
Migration	62682	179914	51875

Table 3: Solution times and total migration weights for 30 time-steps using a re-balancing tolerance of 10%.

5.2 Discussion

The results tabulated above represent a provisional comparison of the three load-balancing algorithms considered. It is clear that the recursive algorithm tends to migrate the least amount of data and lead to the best load-balance after re-partitioning, however this is always achieved at the expense of a larger cut-weight. Jostle would appear to provide the best quality partitions after load-balancing, in the sense that both the maximum load imbalance and the cut-weight are always low, however this is at the expense of requiring more data migration. Metis on the other hand requires less data migration than Jostle and also achieves a low cut-weight, but at the expense of a relatively high load imbalance!

When contrasting the algorithms over a significant number of time-steps and applications of the adaptivity algorithm the results in Tables 2 and 3 appear to be equally inconclusive. Using Jostle generally seems to lead to less time being required by the parallel solver but at the expense of the most data migration. The recursive algorithm on the other hand leads to the least amount of data migration but tends to require more time in the parallel solver – with Metis taking the middle ground. This means that, for a given application (of the form (1)) and computer system, when the time required to take a time-step is quite large compared to the time required to relocate a unit of data Jostle is likely to be the preferred load-balancing tool. On the other hand, if the problem and computer are such that the time required to take a time-step is quite small compared to the cost of moving data, then the recursive algorithm may be best.

It is clear that a much larger number of comparisons on a wide variety of problems and computer architectures will need to be completed if any one algorithm is to emerge as being more robust than the other two. Initial experiments suggest that the recursive algorithm introduced in Section 4 competes well with both Metis and Jostle for the type of load-balancing problem that arises when parallel adaptivity is undertaken. Further work is required however to explore the affects of different parameters, such as the size of the root mesh or the maximum depth of refinement, on the performance of these algorithms, and to assess whether any modifications and improvements can be made to them.

Acknowledgements

Our parallel computations were carried out on the Cray T3D computer at Edinburgh Parallel Computing Centre. NT would like to acknowledge the financial support of the UK and Pakistan governments in the form of ORS and COTS scholarships respectively. The work of PS was undertaken as part of EPSRC grant GR/J84919.

References

[1] C. Bailey, P. Chow, M. Cross, Y. Fryer and K. Pericleous, *"Multiphysics Modelling of the Shape Casting Process"*, Proc. R. Soc. A, 452, 459–486, 1996.

[2] J. Cabello, *"Parallel Explicit Unstructured Grid Solvers on Distributed Memory Computers"*, Advances in Eng. Software, 23, 189–200, 1996.

[3] L. Demkowicz, J.T. Oden and W. Rachowicz, *"A New Finite Element Method for Solving Compressible Navier-Stokes Equations Based on an Operator Splitting Method and h-p Adaptivity"*, Computer Methods in Applied Mechanics and Engineering, 84, 275–326, 1990.

[4] K.D. Devine and J.E. Flaherty, *"Parallel Adaptive HP-Refinement Techniques for Conservation Laws"*, Applied Numerical Mathematics, 20, 367–386, 1996.

[5] 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.

[6] S.K. Godunov, *"A Finite Difference Method for the Numerical Computation of Discontinuous Solutions of the Equations of Fluid dynamics"*, Mat. Sb., 47, 357–393, 1959.

[7] A. Harten, P.D. Lax and B. van Leer, *"On Upstream Differencing and Godunov Type Schemes for Hyperbolic Conservation Laws"*, SIAM Rev., 25, 36–61, 1983.

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

[9] P.K. Jimack *"An Overview of Dynamic Load-Balancing for Parallel Adaptive Computational Mechanics Codes"*, Parallel and Distributed Processing for Computational Mechanics I (ed. B.H.V. Topping), Saxe-Coburg Publications, 1997.

[10] P.K. Jimack *"Techniques for Parallel Adaptivity"*, Parallel and Distributed Processing for Computational Mechanics II (ed. B.H.V. Topping), Saxe-Coburg Publications, 1998.

[11] G. Karypis and V. Kumar, *"A Coarse-Grain Parallel Formulation of Multilevel k-way Graph Partitioning Algorithm"*, Proc. of 8th SIAM Conf. on Parallel Proc. for Scientific Computing, SIAM, 1997.

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

[13] B. van Leer, *"On the Relation Between Upwind Difference Schemes"*, SIAM J. Sci. Stat. Comp., 5, 1–20, 1984.

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

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

[16] M.E.G. Ong, *"Uniform Refinement of Tetrahedron"*, SIAM J. Sci. Comp., 15, 1134–1144, 1994.

[17] P.M. Selwood, M. Berzins and P.M. Dew, *"3D Parallel Mesh Adaptivity: Data-Structures and Algorithms"*, Proc. of 8th SIAM Conf. on Parallel Proc. for Scientific Computing, SIAM, 1997.

[18] P. Selwood, N. Touheed, P.K. Jimack, M. Berzins and P.M. Dew, *"Parallel Dynamic Load-Balancing for the Solution of Transient CFD Problems Using Adaptive Tetrahedral Meshes"*, To appear in Proc. of Parallel CFD 97 Conference, May, 1997, Manchester, UK.

[19] P. Selwood, N.A. Verhoeven, J.M. Nash, M. Berzins, N.P. Weatherill, P.M. Dew and K. Morgan, *"Parallel Mesh Generation and Adaptivity : Partitioning and Analysis"*, Parallel CFD – Proc. of Parallel CFD 96 Conference (ed. A.Ecer, J.Periaux, N.Satufoka and P.Schiano), Elesvier Science BV, 1997.

[20] 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, 81–104, 1997.

[21] 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.

[22] N. Touheed and P.K. Jimack, *"Dynamic Load-Balancing for Adaptive PDE Solvers with Hierarchical Meshes"*, Proc. of 8th SIAM Conf. on Parallel Proc. for Scientific Computing, SIAM, 1997.

[23] 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.

[24] C. Walshaw and M. Berzins, *"Dynamic Load-Balancing For PDE Solvers On Adaptive Unstructured Meshes"*, Concurrency, 7, 17-28, 1995.

[25] C. Walshaw, M. Cross and M.G. Everett, *"Dynamic Load-Balancing for Parallel Adaptive Unstructured Meshes"*, Proc. of 8th SIAM Conf. on Parallel Proc. for Scientific Computing, SIAM, 1997.