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

N. Touheed, P. Selwood, P.K. Jimack [1] and M. Berzins

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

Abstract

In this paper we contrast the performance of a number of different parallel dynamic load-balancing algorithms when used in conjunction with a particular parallel, adaptive, time-dependent, 3-d flow solver. An overview of this solver is given along with a description of the dynamic load-balancing problem that results from its use. Two recently published parallel dynamic load-balancing software tools are then briefly described and a number of recursive parallel dynamic load-balancing techniques are also outlined. The effectiveness of each of these algorithms is then assessed when they are coupled with the parallel adaptive solver and used to tackle a model 3-d flow problem.

1 Introduction

In this work we consider the dynamic load-balancing problem which arises in the adaptive solution of time-dependent partial differential equations (PDEs) using a particular parallel adaptive algorithm based upon hierarchical mesh refinement. This algorithm is applicable to hyperbolic and parabolic problems and is based upon the adaptive refinement of a coarse root mesh, $\mathcal{T}_0$ say, of tetrahedra which covers the domain and is never coarsened. The flexibility of the data structures held within the adaptivity code 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 this paper however we restrict our numerical experiments to a cell-centred finite volume solver. This uses a simple explicit conservative scheme for solving three-dimensional Euler equations and is described in detail in [14,16,18].

[1] Corresponding author. E-mail: pkj@scs.leeds.ac.uk.

In the following section an overview of the parallel adaptive algorithm is given. Section 3 then discusses the dynamic load-balancing problem that arises when using parallel adaptivity. This is followed, in Section 4, by a brief description of four different families of parallel dynamic load-balancing algorithm: two implemented in publicly available software tools ([9,24]), and two more which we have developed from existing published work ([17,21]). The paper concludes by reporting and discussing the results of a number of numerical tests which are used to contrast the load-balancing algorithms for this particular solver.

2 A Parallel Adaptive Algorithm

The PTETRAD software outlined in this section is a parallel implementation of a general-purpose serial code, TETRAD (TETRahedral ADaptivity), for the adaptation of unstructured tetrahedral meshes [18]. The technique used is that of local refinements/derefinements of the mesh to ensure sufficient density of the approximation space throughout the spatial domain at all times. A complete discussion of the parallel algorithms and data structures may be found in [14–16]. Briefly, however, a tree-based hierarchical mesh structure is used, with a fairly rich interconnection between mesh objects, requiring two main data-structure issues to be addressed.

(1) The partitioning of the hierarchical mesh is undertaken at the coarsest level, $\mathcal{T}_0$. This way the mesh hierarchy is such that all parent/child interactions (such as refinement/derefinement) are local to a processor. Also, the partitioning cost will be relatively low since the coarse mesh is generally much smaller than the computational mesh. It may, however, be difficult to obtain a good partition for comparatively small coarse meshes with large amounts of refinement.

(2) Given a partitioned mesh, data-structures are needed to support inter-processor communication and to ensure data consistency. The latter is handled by assigning a processor to own all mesh objects (elements, faces, edges and nodes), whilst communication is supported through the use of halo objects (e.g. copies of inter-processor boundary elements and their associated data-structures). If a mesh object shares a boundary with many processors, it will have a halo copy on each of these. The main advantage of maintaining such a rich set of distributed data structures is that the adaptive code may be used with a variety of different parallel solvers. The disadvantage however is that, compared to more focused algorithms (such as [13] for example), the amount of data that needs to be stored and partitioned between the processors is significantly greater.

Both TETRAD ([18]) and PTETRAD ([14–16]), use a similar strategy to that outlined in [11] to perform mesh adaptation. Edges are first marked for

refinement/derefinement (or neither) according to some estimate or indicator (provided as part of the parallel solver). 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. Green elements may not be further refined, as this may adversely affect mesh quality, but are first removed and then uniform refinement is applied to the parent element.

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.

3 Dynamic Load-Balancing

As explained in Section 2 above PTETRAD requires the coarse root mesh, $\mathcal{T}_0$, to be partitioned into subdomains. It is common to express the requirements of such a partitioning in terms of the weighted dual graph of the 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. For a homogeneous network of processors, the requirement is to partition this graph so that:

(1) the total vertex weight in each subgraph is approximately equal,

hence the computational load per processor will be about the same when the solver is applied. In addition to this, the number of halo elements should be kept as low as possible so as to minimize the communication overhead of the updates that are required at each time step and the computational overhead of the halo calculations. Consequently,

(2) the total cut-weight of the partition should be kept to a minimum.

(The 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 subgraphs within the partition.)

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 goal of a dynamic load-balancing algorithm is to modify an unbalanced partition of the dual graph so as to meet objectives 1 and 2 above in such a way that:

(3) there is a minimal amount of migration of data between subgraphs,

so that the communication overhead associated with moving data between processors does not nullify the computational advantages of obtaining an improved partition. A similar argument motivates the further requirement that:

(4) the load-balancing should be completed in parallel,

as otherwise the resulting sequential bottleneck could seriously reduce the overall efficiency and performance of the adaptive parallel solver.

It should be noted that there is no reason to expect that 1 to 4 above represent a consistent set of requirements. It is perhaps not surprising therefore that the vast majority of published dynamic load-balancing algorithms are based heavily on heuristics. In the next section we introduce two such software tools, called "Metis" ([8,9]) and "Jostle" ([23,24]) respectively. We also briefly describe some further heuristics, based upon [17,21], which are also used in our comparisons.

4 Some Parallel Dynamic Load-Balancing Algorithms

This brief discussion of dynamic load-balancing heuristics will first provide an overview of the multilevel approach that is available in both packages [9] and [24] (in which the existing partition is coarsened before rebalancing takes place), along with the other main features of [9]. In Subsection 4.2 some further approaches, based upon recursive bisection, are then described. In addition to these algorithms a number of other dynamic load-balancing heuristics have been suggested in recent years (see [2,3,6,7,22] or many of the references in [13] for some typical examples) although none of these are included in this investigation. One approach that has been successfully applied to the parallel load-balancing of adaptive unstructured meshes in 3-d is described by Oliker and Biswas in [13]. Their paper provides more than just another dynamic load-balancing heuristic however since it describes an entire strategy for load-balancing, based upon carefully constructed workload and data distribution models for their specific edge-based solver and mesh data structures. Not

only is an attempt made to quantify the gains that would be made from repartitioning the data against the cost of undertaking such a remapping but these gains are estimated, and any resulting repartitioning undertaken, before the mesh refinement stage so as to minimize the amount of data migration.

4.1 Metis and Jostle multilevel software tools

Both Metis ([8,9]) and Jostle ([23,24]) use multilevel partitioning algorithms which 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 at the previous level), followed by a careful repartition of the coarsest graph. This new partition is then projected onto the graph at the previous level and modified using a local algorithm (such as [4,10]) 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. Full details of each tool may be found in [8,9] and [23,24] respectively.

Parallel Metis provides four different options for the rebalancing of a partitioned weighted graph, the first two of which attempt to modify the existing partition incrementally (based upon diffusion algorithms; see, for example, [2,6]) and the second two of which compute entirely new partitions, using the existing partition to determine an efficient mapping between the old and new partitions.

(1) PMetisG refers to a multilevel diffusion algorithm that makes use of load-imbalance information that is obtained globally in order to update the partition incrementally.
(2) PMetisL refers to another diffusion algorithm which only makes use of local load-imbalance information in order to update the partition incrementally.
(3) PMetisM refers to a multilevel remapping algorithm which computes an entirely new p-way partition at the coarsest level and maps this to the original partition, refining locally at each finer level.
(4) PMetisR refers to another p-way remapping algorithm that computes an entirely new p-way partition directly and then maps this to the original partition.

Parallel Jostle only uses a multilevel remapping approach (although a type of local diffusion algorithm, [7], is automatically built into it). The user is able to control the amount of coarsening that takes place when forming the hierarchy of weighted graphs via a "graph reduction threshold parameter" that must be set. This gives Jostle an indication of how small the lowest-level

weighted graph should be allowed to become during the coarsening process. Two different choices of this parameter have been used in this work.

(1) PJostle20 here refers to Parallel Jostle with the default value of 20 vertices for this threshold parameter.
(2) PJostle300 here refers to Parallel Jostle with this value set to 300 vertices.

4.2 Five recursive load-balancing algorithms

In this subsection we briefly describe five more dynamic load-balancing algorithms which are all based upon the use of recursive bisection. The first two of these are based upon recursive coordinate bisection (RCB), [17], and the other three are based upon the algorithm of Vidwans *et al.* in [21].

Algorithm RCB0

This algorithm is the simplest of all of those considered here: recursive coordinate bisection, [17]. Although it is not strictly a dynamic algorithm it may be used in such a context, as discussed in [3] for example. The basic idea at each level of this approach is to cut the domain perpendicular to the coordinate direction in which the subdomain is longest and in such a way that there is an approximately equal load on either side of the cut. This bisection process is then repeated recursively on each subdomain until the required number of subdomains is obtained (with some minor modifications if this number is not a power of two). In our simple implementation each cut is determined sequentially on a single processor by estimating the median element in the longest direction (this is an element such that the total weight of those elements above and below it (in the longest direction) are approximately equal).

Algorithm RCB1

This is a variant of the RCB algorithm in which the sequence of cut directions is left unchanged from that used in the first repartition. The idea behind this approach is that the cutting plane should only move a small amount from one partition to the next, thus making this version of RCB more incremental than the original and ensuring that the majority of elements remain in the same subdomain. For the first repartition this algorithm is identical to RCB0.

This is also a relatively simple algorithm, proposed in [21]. It works by dividing the processors into equally-sized processor groups (or groups with a size difference of one when there is an odd number of processors) based upon the rank (in the MPI sense, [12]) of each processor. Data is then passed from one group to the other until the total workload for each group is approximately equal. This idea is then repeated recursively on each of the processor groups until there are p equally-weighted groups, each consisting of a single processor. For the particular application being considered here each data item to be transferred takes the form of an entire element of the coarse mesh $\mathcal{T}_0$, along with the locally refined mesh beneath it (plus associated data structures). Since these coarse elements have predetermined weights (dependent upon their current level of refinement) it will not generally be possible to get an exact load-balance between the groups at each stage. Nevertheless, the difference in the total weight of the processors in each group allows us to define a target for the required load to be transferred. We also define the group with the higher load to be the "Sender" group and the other group to be the "Receiver".

Having established the target 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. In order to do this we introduce the concept of "candidate processors". Processors in each group that contain elements of $\mathcal{T}_0$ that are face-adjacent to an element of $\mathcal{T}_0$ owned by a processor in the other group are called candidate processors. Where possible, only the candidate processors are allowed to be involved in the actual migration of data from Sender to Receiver. In order to determine precisely *which* elements from the candidate processors in the Sender group are migrated a simple grid-connectivity-based approach is then used. This starts by moving cells which are face adjacent to a cell in the Receiver group and then migrating layers of elements of $\mathcal{T}_0$ (based upon its connectivity) until the required load has been transferred. Such an approach is acknowledged in [21] to frequently lead to the formation of jagged partition boundaries and so in both of the variants below a different heuristic is used to decide which items of data to migrate, and to where. This is based upon attempting to send an equal share of the data from each of the candidate processors in Sender and attempting to keep the cut-weight between the two groups as small as possible, using the notion of "gain" that is defined in [10]. Further details of this heuristic may found in [19,20], along with more extensive details of these two extensions of the algorithm.

Algorithm VKV1

This is a variant of the algorithm of Vidwans *et al.* which involves a slightly more sophisticated mechanism for dividing the processors into two equally-sized groups than simply using the processor ranks. This is achieved by considering the weighted partition communication graph (WPCG) of the initial partition of the mesh $\mathcal{T}_0$. The WPCG is obtained by having one vertex for every processor and an edge between two vertices if and only if the corresponding processors are face adjacent to each other (i.e. there is an element of $\mathcal{T}_0$ on one processor which is face-adjacent to an element of $\mathcal{T}_0$ on the other processor). The weight w_{N_i} of the i^{th} vertex of the WPCG is equal to the sum of weights of all coarse elements on the i^{th} processor and the weight $w_{E_{ij}}$ of the edge connecting the i^{th} and j^{th} processors is equal to the sum of weights of all coarse element faces on the partition boundary between the two processors. We now use a weighted version of the spectral bisection algorithm (see, for example, [22]) to order the processors (as opposed to just using their rank) before dividing them into two groups of equal size. The algorithm then proceeds as in VKV0 above (but with the modified scheme for selecting which cells to migrate at each level of the recursion).

Algorithm VKV2

This is a second variant of the algorithm in [21] which also applies weighted spectral bisection to the WPCG, but this time at each level of the recursion rather than as a pre-ordering mechanism. Moreover, when dividing the processors into two groups based upon this ordering we do not select groups of equal size but instead choose to form groups of (approximately) equal total weight. If the coarse mesh $\mathcal{T}_0$ is much more heavily refined in some parts than others then these two groups could have very different numbers of processors in them. When attempting to balance the load in each group it is now necessary to define the Sender to be the group with the highest average load per processor and to send from it enough data to ensure that the average load per processor in each group is approximately equal after rebalancing. We again achieve this data migration through the use of candidate processors and make use of the notion of gain to try to keep the cut-weight between the two groups as small as possible (as described in [19,20]).

It should be emphasized that, as with the multilevel and diffusion algorithms of the previous subsection, when applying all five of these recursive bisection algorithms to the problem outlined in Section 2 we only actually work with the weighted dual graph of $\mathcal{T}_0$ that was defined in Section 3. Only when the final location of each coarse element in $\mathcal{T}_0$ has been determined are the full mesh data structures transferred.

5 Some Computational Comparisons

In this section we contrast the different dynamic load-balancing algorithms briefly described in Section 4 when used in conjunction with the parallel adaptive solver outlined in Section 2. 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. However, when there is heavy local refinement in some regions of the spatial domain (as in the examples below) the dual graph of $\mathcal{T}_0$ has 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.

In each example we apply the parallel adaptive Euler solver of [14] to model a shock wave diffraction around the 3D right-angled corner formed between two cuboid regions (taken from [14,15,18]). The initial condition is of Rankine-Hugoniot shock data at the interface of the two cuboid regions with a shock speed of Mach 1.7. Fig. 1 illustrates how the mesh adapts to the solution as the shock progresses through the domain. Note that the smaller cube is situated *behind* the larger cube from the viewpoint selected for this figure. 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. All of the computations described were completed on a 32 processor SGI Origin2000 using the C binding of MPI ([12]) for communication and data redistribution. To obtain results and timings that are as reproducible as possible the calculations were performed using the Miser utility which allocates exclusive use of the processors and memory being used for the entire duration of each run.

5.1 Example one

For this example a root mesh, $\mathcal{T}_0$, containing 5184 elements is used and up to three levels of refinement are allowed. This leads to an initial fine mesh containing ≈ 80000 elements, with more elements appearing in this leaf-level mesh at later times. Note that throughout these calculations the adaptive mesh has a resolution equivalent to a mesh of $5184 \times 8^3 \approx 2.6$ million uniform, regular elements.

Table 1 presents a comparison of some partition-quality metrics when the different load-balancing algorithms are applied using 8, 16 and 32 processors. (Similar results were also collected for 2 and 4 processors, however these are not included as they show only small variations in performance for the differ-

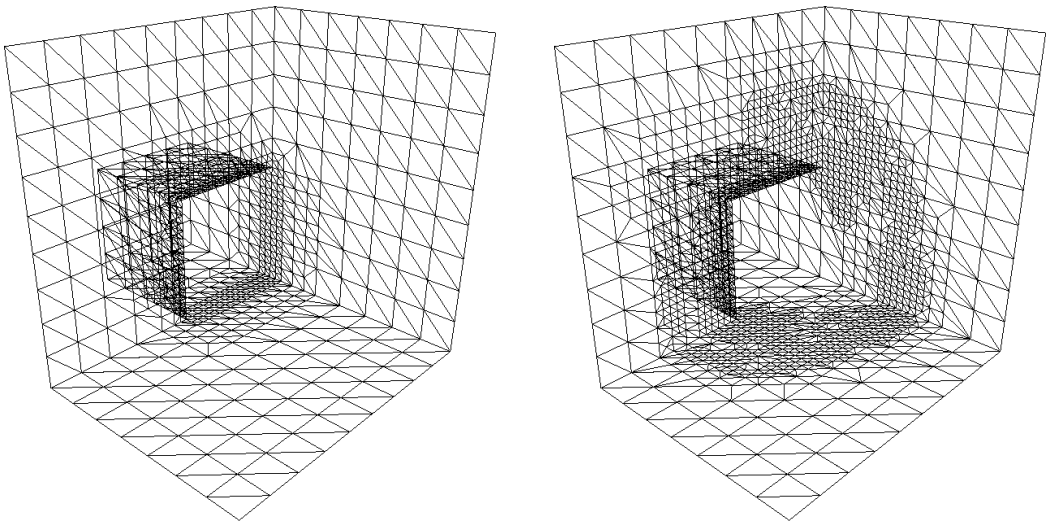

Fig. 1. An initial mesh (left) and an adapted mesh after a number of time steps (right).

ent algorithms and appear to yield little information that cannot be derived from the 8 processor examples.) In each case the initial partition has a maximum imbalance (MaxImb) of over 30% (this is the percentage by which the total vertex weight of the most heavily-weighted subgraph exceeds the average weight of the subgraphs) and the cut-weight (CutWt) is given. The solution times (SolT) quoted represent the wall-clock time (in seconds) taken by the parallel finite volume solver for the next thirty 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 (MigTot). (Note that, with 32 processors, two of the algorithms failed to produce partitions that could be used by our solver in this example, which is why there are some blank entries in this table, and the one that follows. The reasons for this are discussed in Subsection 5.3.)

An alternative form of comparison between the load-balancing algorithms is provided by Table 2. These results are intended to provide a more complete picture of each load-balancer's performance, and are obtained using sequences of 300 time steps with ten separate mesh adaptations (after every 30 time steps). Whenever the maximum imbalance exceeds a prescribed tolerance of 10% after adaptivity has occurred the load-balancing algorithm is called. The solution times (SolT) quoted are the total times for the finite volume solver to complete all 300 time steps *excluding* the repartitioning times, which are given separately. These latter times have themselves been split into the cost of calculating which coarse elements must be transferred (RepT) and the cost of actually transferring them (RedT), thus giving a true indication of the total overhead of each load-balancing algorithm on a particular architecture. As additional, architecture independent, indicators of this overhead the table

also shows the total weight of all of the root elements that were migrated throughout the 300 time steps (MigTot), the number of times that repartitioning needed to be undertaken (Mig#), and the average amount of data migrated each time repartitioning took place (*excluding* the first repartition) (MigAve).

5.2 Examples two and three

These examples are similar to the previous one except that now the root mesh, $\mathcal{T}_0$, contains 34560 elements, with a maximum of three and four levels of refinement respectively. When up to three levels of refinement are allowed the initial fine mesh contains $\approx$ 291000 elements, with more elements appearing at later times, and with up to four levels of refinement the final leaf-level mesh contains over a million elements. Note that throughout these calculations the three-level adaptive mesh has a resolution equivalent to a mesh of $34560 \times 8^3 \approx 17.7$ million uniform, regular elements, and the four-level mesh has a resolution equivalent to over 140 million such elements. Tables 3 and 4 present corresponding results to those given in Tables 1 and 2 respectively for the three-level example and Table 5 presents results over 300 time steps for the four-level example. Both of these examples made use of the same initial mesh and so no table is provided for the rebalancing of the initial partition in the latter case since the results are identical to those in Table 3.

5.3 Discussion

Before discussing the specific details of the figures presented in Tables 1 to 5 there are a few general observations that should be made. We emphasize that these results are for a particular parallel adaptive solver and refinement algorithm applied to a particular system of partial differential equations. Great care should therefore be taken not to attempt to extrapolate from this data more than is reasonable. Two particularly important features of the parallel adaptive algorithm used here are that it requires a partition of the coarse root mesh rather than the actual computational mesh at each stage, and that a significant computational overhead is incurred when the cut-weight is large (due to the additional calculations required on halo elements). These factors clearly have a significant effect on the outcome of all of the load-balancing experiments undertaken. We also remark that it is generally accepted that remapping schemes (such as PMetisM and PMetisR) tend to produce better cut-weights in the new partition than diffusive schemes (such as PMetisG and PMetisL) but at the cost of a greater amount of data redistribution. Furthermore, it should be noted that the implementation of RCB used here

is not entirely parallel (since it produces the bisection of each subdomain sequentially) and so the timings for this (RepT) could possibly be improved somewhat if a fully parallel algorithm were to be implemented.

The selection of a tolerance of 10% for Tables 2, 4 and 5 is not particularly significant. Similar data was also collected for tolerances of 5% and 15% with similar qualitative outcomes. We have chosen to present these specific figures mainly because the overall solution times were marginally lower for this choice of tolerance than for the other two. The advantages of basing the decision about whether or not to repartition upon a tolerance such as this are its simplicity and generality. It seems unlikely however that this strategy will be as efficient as that described in [13], nevertheless we find that our simple strategy is surprisingly robust and, since our adaptive software is intended for use with a variety of different parallel solvers based upon different data structures, it saves the work of developing reliable cost models for each of these. Finally, we note that the migration patterns themselves will have a significant effect on the total repartitioning time. For example, if nearly all of the data being migrated is going to or from one processor then this may well take significantly longer than if the same amount of data is being migrated equally between all p processors. This is likely to be an important issue in most practical applications, where refinement will usually be quite localized — such as when following a shock wave, as in the examples considered here.

Example one

The first observation that must be made about the results in Table 1 is that for the 32 processor case two of the algorithms (VKV0 and PMetisL) produced new partitions with an empty subdomain. This causes our parallel solver to fail and so results are not included for these runs. An explanation for this behaviour comes from the observation that in this example we are attempting to repartition an extremely non-uniformly weighted graph which is initially very poorly balanced. The basic problem is that the coarse mesh for this problem has an insufficient number of elements to allow the leaf mesh to be easily load-balanced. In fact, with the exception of PMetisG, all of the algorithms have great difficulty in obtaining a good final partition (as an indication of the difficulties associated with finding such a partition we note that the output from PMetisG gives just 5 (highly-refined) coarse elements on one processor and as many as 1705 on another). Inspection of Table 1 however shows that these poor load-balances appear to have little effect on the solution times. This is because these small examples contain a relatively high halo element ratio which means that the cut-weight appears to be a much more significant statistic. It is the improved cut-weight per processor which appears to be the cause of the surprisingly small solution times, after repartitioning, when 32 processors are used.

	MaxImb	CutWt	SolT	MigTot	Procs
Initial	33%	3064	32.6	0	
RCB0	4%	6474	25.7	73474	
RCB1	4%	6474	25.7	73474	
VKV0	3%	5634	25.6	26569	
VKV1	0%	4942	25.4	30877	
VKV2	0%	4905	25.0	25182	
PMetisG	5%	3719	25.0	18025	8
PMetisL	5%	4509	25.5	19094	
PMetisM	5%	3496	25.2	49800	
PMetisR	3%	3202	24.1	48293	
PJostle20	2%	4171	25.1	82376	
PJostle300	2%	3842	25.2	54832	
Initial	55%	4760	17.1	0	
RCB0	9%	9197	13.1	81465	
RCB1	9%	9197	13.1	81465	
VKV0	2%	8010	13.3	31507	
VKV1	5%	7316	13.0	30058	
VKV2	4%	7510	12.4	30424	
PMetisG	5%	6402	12.7	30185	16
PMetisL	18%	5964	13.2	33579	
PMetisM	6%	4824	12.2	45858	
PMetisR	9%	4872	12.5	48970	
PJostle20	6%	5840	13.1	66119	
PJostle300	7%	5501	13.4	55707	
Initial	143%	5616	11.2	0	
RCB0	28%	10490	6.4	69921	
RCB1	28%	10490	6.4	69921	
VKV0	—	—	—	—	
VKV1	24%	9320	6.1	38131	
VKV2	24%	8959	6.0	36714	
PMetisG	6%	8680	6.2	32744	32
PMetisL	—	—	—	—	
PMetisM	20%	6105	5.5	47213	
PMetisR	16%	6317	5.4	51886	
PJostle20	22%	7213	6.0	57841	
PJostle300	42%	7103	6.6	43884	

Table 1
Some partition-quality metrics for a single rebalancing step for Example one.

Although each of the load-balancing algorithms performs sufficiently well in this case to ensure that the parallel adaptive solver scales far better than would otherwise be the case, there are some differences that are worth noting. As expected, the remapping schemes tend to lead to better cut-weights than the diffusion schemes which migrate less data. It is also apparent that for parallel Jostle a smaller graph reduction threshold parameter leads to a better load-

balance but at the cost of more data migration. Finally for Table 1 we note that the recursive bisection algorithms appear to be quite competitive in terms of the metrics that have been considered. The only exception to this being the high migration costs associated with RCB0 and RCB1. These are always very high at the start of an adaptive solve since the RCB algorithm cannot take into account the initial partition produced by the parallel adaptivity software.

The final column of Table 2 shows that, when the initial migration is excluded, the average amount of data migrated each time the RCB algorithms are used is significantly less than for the initial migration, as one would expect. Arguably, this gives a fairer reflection of these heuristics as dynamic load-balancers. Unfortunately however both RCB0 and RCB1 consistently fail to obtain load-imbalances of less than the 10% threshold and so repartitioning is attempted on a larger number of occasions than with the other methods. (The reason for this difficulty is that heavily-weighted elements appearing near the middle of the sorted list of elements make it impossible to split the list into equally-weighted halves: and the effects of this become more and more significant each time the bisection process is repeated.) Slightly different conclusions regarding the performance of the VKV algorithms to those drawn from Table 1 may also now be made. Over the sequence of time steps with numerous mesh refinements these algorithms tend to be slightly inferior to PMetis and PJostle. This suggests that there may be dangers in only using the "snap-shots" associated with a single repartitioning problem when attempting to assess the quality of a dynamic algorithm. Having made this observation however, we also note that the relatively small size of the coarse mesh for this problem still causes the same difficulties for 32 processors that were apparent in Table 1 and it again appears that PMetisG is the most robust algorithm for dealing with this severe situation.

For 8 and 16 processors the rebalancing problem is less extreme. We still see however that, for this small problem, the total savings that result from an efficient load-balancing strategy are not that large. Nevertheless, the figures for the performance of parallel Jostle in Table 2 do demonstrate that the ability to be able to control the graph reduction threshold parameter can be important. We also observe that, even solving the same problem with the same code, the best choice of repartitioning algorithm appears to change with p (PMetisR for $p = 16$ and PMetisG for $p = 32$ for example). This clearly illustrates the difficulties associated with attempting to rank the different algorithms.

Examples two and three

We now consider the statistics presented in Table 3. It is clear from this table that the coarse mesh of 34560 elements is sufficiently large to permit a leaf mesh with up to three levels of refinement to be partitioned across 32 pro-

	SolT	RepT	RedT	MigTot	Mig#	MigAvg	Procs
Initial	271.9	0.0	0.0	0	0	0	
RCB0	218.8	4.5	24.8	108130	5	8664	
RCB1	218.7	5.1	23.6	98649	5	6294	
VKV0	221.8	4.5	12.4	32477	3	2954	
VKV1	211.6	3.9	11.7	37206	3	3164	
VKV2	212.2	6.3	14.1	35592	4	3470	8
PMetisG	207.2	4.0	11.8	26676	4	2884	
PMetisL	208.9	3.3	14.2	28587	4	3164	
PMetisM	205.7	1.1	11.9	80765	2	30965	
PMetisR	204.1	2.4	15.2	114738	3	33222	
PJostle20	214.6	5.5	28.1	239822	4	52482	
PJostle300	215.7	4.8	23.6	166933	4	37367	
Initial	146.1	0.0	0.0	0	0	0	
RCB0	130.8	6.1	46.4	242178	9	20089	
RCB1	127.2	5.9	33.3	162483	9	10127	
VKV0	137.9	7.1	18.8	50173	5	4666	
VKV1	127.8	5.3	16.0	43518	5	3365	
VKV2	117.3	6.8	14.6	45665	5	3810	16
PMetisG	113.2	3.6	15.5	45892	6	3141	
PMetisL	112.7	3.7	15.6	52992	6	3883	
PMetisM	113.0	4.2	26.8	234071	6	37643	
PMetisR	107.1	2.5	18.5	179462	4	43497	
PJostle20	118.4	10.2	45.4	491945	8	60832	
PJostle300	119.8	6.6	32.8	301168	6	49092	
Initial	104.3	0.0	0.0	0	0	0	
RCB0	71.3	4.8	34.7	255368	10	20605	
RCB1	72.5	3.7	28.9	195904	10	13998	
VKV0	—	—	—	—	–	—	
VKV1	70.1	9.3	19.1	73578	8	5064	
VKV2	69.7	10.3	22.8	68871	10	3573	
PMetisG	64.8	3.1	17.1	64719	8	4568	32
PMetisL	—	—	—	—	–	—	
PMetisM	56.3	4.8	25.7	400204	10	39221	
PMetisR	57.8	5.0	26.2	406607	10	39413	
PJostle20	62.9	11.7	33.3	578775	10	57882	
PJostle300	64.2	4.8	27.7	251810	10	23103	

Table 2
Performance statistics over 300 time steps using a rebalancing tolerance of 10% for Example one.

cessors without significant difficulty. It is also possible, once again, to observe the excellent scalability of the parallel solver as the number of processors is increased, and to see that the algorithms producing the lowest cut-weights (the Metis remapping algorithms) lead to the best solution times. Interestingly however, neither PMetisM or PMetisR migrate as much data as Jostle20,

	MaxImb	CutWt	SolT	MigTot	Procs
Initial	48%	6776	118.6	0	
RCB0	2%	18359	87.9	230335	
RCB1	2%	18359	87.9	230335	
VKV0	0%	15947	86.9	89811	
VKV1	0%	12314	85.1	86945	
VKV2	2%	11395	83.8	77261	
PMetisG	5%	8682	88.4	88151	8
PMetisL	5%	7862	85.8	92669	
PMetisM	4%	5511	83.4	137326	
PMetisR	3%	6001	81.4	174347	
PJostle20	1%	6919	84.9	225691	
PJostle300	2%	6778	81.8	192914	
Initial	91%	9183	71.4	0	
RCB0	3%	25577	47.5	276327	
RCB1	3%	25577	47.5	276327	
VKV0	0%	20490	44.2	115058	
VKV1	1%	16406	43.0	112618	
VKV2	1%	16027	43.2	117723	
PMetisG	6%	12045	43.7	101199	16
PMetisL	5%	11901	43.4	109214	
PMetisM	4%	8460	42.2	179626	
PMetisR	5%	8346	41.7	188889	
PJostle20	2%	10188	46.6	246636	
PJostle300	3%	10068	46.1	182617	
Initial	96%	12875	42.5	0	
RCB0	6%	28510	22.8	254311	
RCB1	6%	28510	22.8	254311	
VKV0	2%	28085	22.5	129931	
VKV1	3%	23457	22.1	129573	
VKV2	2%	22950	22.6	128408	
PMetisG	5%	18357	22.2	112400	32
PMetisL	5%	17209	22.6	132722	
PMetisM	5%	12239	21.0	158375	
PMetisR	5%	12278	20.8	208496	
PJostle20	4%	14009	21.8	235667	
PJostle300	9%	15009	22.9	176209	

Table 3
Some partition-quality metrics for a single rebalancing step for Example two.

which has almost the same migration total as RCB: although it is far more efficient than RCB. (Note that in Tables 1 and 3 RCB0 and RCB1 give identical results since they are identical algorithms when applied to the first partition.)

The next set of results presented is given in Table 4 where we again see that,

by considering numerous applications of each algorithm over the evolution of an entire problem, a much greater diversity in their relative performance is apparent than from a single application of each algorithm (as in Table 3). Table 5 presents similar results for a different run of the same problem but with a finer leaf mesh. In each case a major factor determining the overall parallel simulation time is the number of occasions on which repartitioning is necessary after adaptivity has occurred. This suggests that our approach of basing the repartitioning decision on whether or not the maximum imbalance exceeds some given tolerance may be a little simplistic (and an approach such as in [13], where remapping costs are estimated before deciding whether or not to repartition, may be worthwhile). Nevertheless, we feel that the simplicity and generality of our criterion mean that it is worthy of this investigation.

Perhaps the most striking statistics in both Tables 4 and 5 are the times taken by the recursive bisection algorithms to calculate what the new partitions should be (RepT). For the RCB algorithm this is clearly due, at least in part, to the use of a sequential bisection algorithm at each step. Nevertheless, there is a more significant difficulty than this, which is also apparent for the three VKV algorithms. This difficulty stems from the fact that in both example two and example three the coarse mesh contains 34560 elements, hence this is the size of the weighted graph that must be partitioned. The PMetis and PJostle algorithms coarsen the weighted dual graph of this mesh in order to perform repartitioning, whereas the RCB and VKV algorithms do not. Whilst all of these algorithms have a linear complexity, the use of graph coarsening clearly has an enormous effect on the cost of the algorithms as the root mesh grows in size (presumably because one is able to reduce the average amount of work per graph vertex through coarsening). Moreover, it is clear from the first example that in order to have any hope of balancing leaf meshes of the order of a million elements, which are of a non-uniform density across the domain, a root mesh of at least the magnitude used here is absolutely necessary.

When contrasting the performance of the different versions of PMetis on these examples Table 4 appears to verify the conclusions drawn above, from Table 3, that the remapping algorithms perform best. This is not evident from the results of Table 5 however, where PMetisL performs best on 8 and 16 processors and PMetisR does very badly in the 32 processor case. One observation that does seem to hold from these latter examples however is that PMetisG does not perform any better than the other versions in these cases, for which the repartitioning problem is less demanding than the rather extreme situation of example one. No conclusions can be drawn about the choice of graph reduction threshold parameter in PJostle on the basis of these experiments. Finally, we comment that the cumulative timings for each algorithm in Tables 2, 4 and 5 should be treated with great caution since if, for some other problem, the amount of work per time step were significantly greater than for the explicit scheme outlined in Section 2, the relative importance of the solution time over

	SolT	RepT	RedT	MigTot	Mig#	MigAvg	Procs
Initial	1071.0	0.0	0.0	0	0	0	
RCB0	758.5	14.4	78.4	413297	4	60987	
RCB1	792.3	19.0	88.8	358331	5	31999	
VKV0	783.9	47.3	36.4	118507	3	14348	
VKV1	785.3	12.7	25.5	107001	2	20056	
VKV2	773.7	11.9	23.7	99744	2	22483	8
PMetisG	747.2	5.3	35.8	162429	2	74278	
PMetisL	742.1	4.4	20.2	101219	2	8550	
PMetisM	750.6	5.1	29.8	219518	2	82192	
PMetisR	727.1	4.8	34.4	276733	2	102386	
PJostle20	751.0	12.8	62.3	619664	3	196986	
PJostle300	735.8	7.3	32.8	305270	2	112356	
Initial	657.6	0.0	0.0	0	0	0	
RCB0	458.2	12.0	84.5	540378	5	66013	
RCB1	450.8	12.0	76.5	430972	5	38661	
VKV0	430.7	31.0	32.2	155247	4	13396	
VKV1	426.7	22.1	29.8	151544	4	12975	
VKV2	419.6	50.3	32.8	170581	4	17619	16
PMetisG	391.8	9.0	25.5	151765	4	16855	
PMetisL	393.4	10.3	30.2	155621	5	11602	
PMetisM	371.5	5.8	33.0	431692	3	126033	
PMetisR	371.6	5.9	33.9	443908	3	127510	
PJostle20	399.0	13.9	59.6	899040	4	217468	
PJostle300	402.3	8.2	41.0	518883	3	168133	
Initial	422.4	0.0	0.0	0	0	0	
RCB0	244.7	37.7	108.7	718023	10	51524	
RCB1	232.9	28.2	80.0	555659	10	33483	
VKV0	265.3	45.6	42.5	180421	6	10098	
VKV1	238.0	32.4	34.6	194244	6	12934	
VKV2	232.3	26.7	26.6	173236	5	11207	32
PMetisG	206.1	8.1	25.0	186297	6	14779	
PMetisL	204.8	7.2	26.0	211560	6	15768	
PMetisM	189.4	6.6	28.0	703927	5	136388	
PMetisR	197.2	11.6	46.8	1044486	8	119427	
PJostle20	207.0	14.2	40.4	1078759	5	210773	
PJostle300	211.5	17.3	60.5	1184563	7	168059	

Table 4
Performance statistics for 300 time steps using a rebalancing tolerance of 10% for Example two.

the repartitioning and migration times would increase. Hence, for 8 processors in Example 3, PMetisG would become more efficient than PMetisL for a sufficiently large amount of work per time step (although VKV2 would remain most efficient). Similarly, for $p = 32$, PMetisM and PMetisR would become more efficient than PJostle20 if the amount of work between each adaptation

	SolT	RepT	RedT	MigTot	Mig#	MigAvg	Procs
Initial	4698.7	0.0	0.0	0	0	0	
RCB0	3114.7	13.3	146.1	389363	2	159028	
RCB1	3114.7	13.3	146.1	389363	2	159028	
VKV0	3582.5	50.9	121.9	245974	2	156163	
VKV1	3162.5	20.5	82.7	233785	2	146840	
VKV2	2911.3	20.0	57.3	206932	2	129671	8
PMetisG	2925.1	29.9	102.5	255669	3	83759	
PMetisL	2956.4	11.8	54.5	169189	2	76520	
PMetisM	3073.5	13.9	83.4	531560	2	394234	
PMetisR	3014.9	13.2	120.2	843478	2	669131	
PJostle20	3074.6	18.8	123.5	1182777	2	957086	
PJostle300	2963.8	18.1	114.1	698202	2	505288	
Initial	3383.2	0.0	0.0	0	0	0	
RCB0	1757.9	19.0	226.4	808593	3	266133	
RCB1	1750.3	21.4	235.7	780810	3	252242	
VKV0	1706.6	52.5	121.8	434778	3	159860	
VKV1	1659.6	24.6	50.2	275744	2	163126	
VKV2	1795.1	45.3	58.9	319976	2	202253	16
PMetisG	1684.0	16.9	126.8	513631	3	206216	
PMetisL	1547.8	17.7	81.2	380822	3	135804	
PMetisM	1843.9	8.7	68.0	658804	2	479178	
PMetisR	1692.4	7.8	65.5	830448	2	641559	
PJostle20	1696.6	21.3	160.5	1966020	3	859692	
PJostle300	1607.4	10.8	73.4	1085660	2	903043	
Initial	1452.2	0.0	0.0	0	0	0	
RCB0	956.8	87.7	580.3	1705189	10	161209	
RCB1	932.6	61.9	399.3	1532089	10	141975	
VKV0	996.0	60.8	117.0	434747	4	101605	
VKV1	991.7	68.4	142.5	453395	5	80956	
VKV2	978.9	34.1	71.8	359088	3	115340	32
PMetisG	802.0	9.8	58.0	403526	3	145563	
PMetisL	797.3	9.9	53.5	458810	3	163044	
PMetisM	776.5	9.1	61.2	1247821	3	544723	
PMetisR	761.3	41.0	295.7	5109665	10	544574	
PJostle20	783.9	8.3	39.8	926074	2	690407	
PJostle300	827.3	28.2	189.2	3300839	5	781158	

Table 5
Performance statistics over 300 time steps using a rebalancing tolerance of 10% for Example three.

of the mesh were to go up by a sufficient amount.

6 Conclusions and Future Work

In this paper we have attempted to make some simple comparisons between a variety of different dynamic load-balancing algorithms with respect to one particular application. In such a series of tests one cannot hope to reach any *general* conclusions concerning these dynamic load-balancing heuristics since it is well-known that different techniques are likely to perform best for different problems. Nevertheless, it is possible to make a number of specific observations about the heuristics when restricted to the class of problem considered here: with moderate graph sizes (5000 to 35000 vertices) and highly variable vertex weights.

Any reasonable dynamic load-balancing algorithm yields a significant improvement in the performance of the solver and permits good scalability of the solver as the number of processors increases. For small numbers of processors (eight or less), the simple recursive bisection algorithms (especially VKV2) are generally quite competitive with PMetis and PJostle. Furthermore, the use of RCB1 rather than RCB0 significantly reduces the amount of data migration required for the two smaller problems but makes a less significant reduction for the largest problem considered. This reduction does not always result in a better overall solution time though, since, on some occasions, the solver time (SolT) is higher than for the partitions produced by RCB0.

The algorithm that is most robust in terms of always delivering reasonably well-balanced final partitions is PMetisG, which is based upon diffusion but still makes use of global information to ensure that a good partition is obtained. This is not usually the best algorithm in any given situation however. Although the Metis remapping algorithms (PMetisM and PMetisR) frequently produce the best solver times (SolT) (because they typically delivered the best new partitions), overall no one algorithm comes out as being better than the rest. (For each of VKV2, PMetisL, PMetisM, PMetisR and PJostle20 we encountered at least one example in which that version gave the smallest overall solution time.) As the number of levels of refinement increases the actual migration time (RedT) tends to become more significant than the time taken to calculate the new partitions (RepT).

It is more informative to assess the quality of the different load-balancers over an entire sequence of mesh refinements rather than just to consider simple metrics, such as cut-weight or migration volume, at a single repartitioning step. The number of repartitions required over a sequence of refinements appears to be the most important single factor in determining the overall performance of each algorithm. This is something that is very hard to predict in advance, however it is clear that algorithms which generally produce quite poorly-balanced loads will be penalized the most in this respect (e.g. RCB0 and RCB1 in these

examples).

Although no one algorithm comes out as being better than the rest, it is reasonable to conclude that both of the packages, PMetis and PJostle, performed better than the relatively simple recursive bisection codes that we implemented ourselves, especially with larger numbers of processors. For root meshes at the larger end of the spectrum considered here it is advisable that some form of multilevel algorithm, based upon graph coarsening, is used.

In view of the observation that the number of repartitions required appears to be the most important single factor, a key issue that must be addressed in any future work is that of determining precisely when repartitioning should take place. This issue is already beginning to receive significant attention, as in [13] for example. An essential ingredient required to make such decisions would appear to be the use of a metric which includes both migration costs and communication/halo costs for a given partition.

For example, [1], consider the partitioning problem after the $(n+1)$th remesh with just two processors. Let L^{n+1} be the weighted Laplacian matrix of the weighted dual graph of the coarsest level mesh after the $(n+1)$th remesh (see, for example, [5]) and let $\underline{x}^n$ be the latest partition vector ($x_i^n = \pm 1$ according to which subdomain coarse element i belongs to). The communication/halo overhead at the next step of the solver is therefore proportional to $(\underline{x}^n)^T L^{n+1} \underline{x}^n$. Conversely, if repartitioning were to take place (leading to a new partition vector $\underline{x}^{n+1}$) before the next step of the solver, the new communication/halo overhead plus the movement cost would be proportional to

$$(\underline{x}^{n+1})^T \; L^{n+1} \; \underline{x}^{n+1} + \lambda (\underline{x}^{n+1} - \underline{x}^n)^T (\underline{x}^{n+1} - \underline{x}^n)$$

for some constant λ (the ratio of moving cost to communications costs). With a suitable choice of this constant, this expression could be used to decide whether or not to accept a possible new partition, obtained using one of the algorithms considered above for example. Alternatively, this quadratic form could be minimized by solving the equations:

$$(L^{n+1} + \lambda I) \; \underline{x}^{n+1} = \; \lambda \; \underline{x}^n$$

for a new partition vector $\underline{x}^{n+1}$, thus yielding an alternative dynamic load-balancing heuristic. This approach therefore provides an explicit mechanism not only for deciding whether or not it is worthwhile migrating data to a possible new partition from the existing partition, but also for taking into account the migration costs in obtaining a possible new partition.

Acknowledgements

We thank Bruce Hendrickson and the anonymous referees for their valuable suggestions which have significantly influenced the final form of the paper. We are grateful to Jason Lander, Joanna Schmidt and Jasbinder Singh for their advice and assistance concerning the Origin 2000. 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] M. Berzins, Private Communication, 1999.

[2] G. Cybenko, *"Dynamic Load Balancing for Distributed Memory Multiprocessors"*, J. of Parallel and Distributed Computing, 7, 279–301, 1989.

[3] P. Diniz, S. Plimpton, B. Hendrickson and R. Leland, *"Parallel Algorithms for Dynamically Partitioning Unstructured Grids"*, Proc. of 7th SIAM Conf. on Parallel Proc. for Sci. Comp., SIAM, 1995.

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

[5] D.C. Hodgson and P.K. Jimack, *"A Domain Decomposition Preconditioner for a Parallel Finite Element Solver on Distributed Unstructured Grids"*, Parallel Computing, 23, 1157–1181, 1997.

[6] G. Horton, *"A Multi-Level Diffusion Method for Dynamic Load Balancing"*, Parallel Computing, 19, 209–218, 1993.

[7] 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 Lab., Cheshire WA4 4AD, UK, 1995.

[8] 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.

[9] G. Karypis, K. Schloegel and V. Kumar, *"ParMetis: Parallel Graph Partitioning and Sparse Matrix Ordering Library. Version 2.0"*, Department of Computer Science, University of Minnesota, 1998.

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

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

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

[13] L. Oliker and R. Biswas, *"PLUM: Parallel Load Balancing for Adaptive Unstructured Meshes"*, J. Parallel and Distributed Computing, 52, 150–177, 1998.

[14] P.M. Selwood and M. Berzins, *"Portable Parallel Adaptation of Unstructured Tetrahedral Meshes"*, accepted by Concurrency, 1999.

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

[16] P. Selwood, M. Berzins J. Nash and P.M. Dew, *"Portable Parallel Adaptation of Unstructured Tetrahedral Meshes"*, Solving Irregularly Structured Problems in Parallel – Proc. of Irregular 98 Conference (ed. A.Ferreira *et al.*), Springer Lecture Notes in Computer Science, 1457, 56–67, 1998.

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

[18] 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.

[19] N. Touheed, *"Parallel Dynamic Load-Balancing for Adaptive Distributed Memory PDE Solvers"*, Ph.D. Thesis, Univesity of Leeds, 1998.

[20] 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 Sci. Comp., SIAM, 1997.

[21] A. Vidwans, Y. Kallinderis and V. Venkatakrishnan, *"Parallel Dynamic Load-Balancing Algorithm for Three-Dimensional Adaptive Unstructured Grids"*, AIAA Journal, 32, 497–505, 1994.

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

[23] 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 Sci. Comp., SIAM, 1997.

[24] C. Walshaw, M. Cross and M.G. Everett, *"Parallel Dynamic Graph Partitioning for Adaptive Unstructured Meshes"*, J. Par. Dist. Comput., 47, 102-108, 1997.