Parallel Unstructured Tetrahedral Mesh Adaptation: Algorithms, Implementation and Scalability

P.M. Selwood & M. Berzins
Computational PDEs Unit, School of Computer Studies,
The University of Leeds, Leeds, LS2 9JT, UK

DRAFT PAPER submitted for publication.

Abstract

The use of unstructured adaptive tetrahedral meshes in the solution of transient flows poses a challenge for parallel computing due to the irregular and frequently changing nature of the data and its distribution. A parallel mesh adaptation algorithm, PTETRAD, for unstructured tetrahedral meshes (based on the serial code TETRAD) is described and analysed. The portable implementation of the parallel code in C with MPI is described and discussed. The scalability of the code is considered, analysed and illustrated by numerical experiments using a shock wave diffraction problem.

1 Introduction

Spatial mesh adaptation techniques are increasingly being used for the solution of transient partial differential equations (PDEs) e.g. [18]. The use of error indicators to carefully place fewer triangular, tetrahedral, quadrilateral or hexahedral elements than with a regular mesh makes it possible to solve problems more quickly, but without reducing accuracy and to gain robustness and confidence in the accuracy of the solution. The most commonly used mesh adaptation techniques are usually classified as h, p or r refinement, see [1, 12]. In h–adaptation an initial mesh that defines a geometry is successively refined and coarsened by the addition and removal of nodes until a mesh appropriate to the solution has been constructed. This approach can be used with both structured and unstructured meshes and in any number of spatial dimensions.

Despite the success of mesh adaptation in reducing the overall solution time, there is a need for ever larger problems to be solved with greater accuracy. Current serial computers cannot cope with the memory or computational demands of such problems and it is thus necessary to consider the use of parallel computers in order to achieve acceptable solution times. Combining a parallel solver with a serial adaptation procedure is clearly undesirable as the adaptation would be a bottleneck, preventing the efficient scaling of the solution process. Furthermore, meshes are often so large as not to fit on a single processor's memory. There is thus a clear need for a parallel mesh adaptation procedure. The main difficulties in using parallel methods are the complex data structures, the irregular nature of the mesh and the constantly evolving nature of the mesh as it is adapted to follow the evolution of the flow.

In this paper we discuss the issues arising in the parallelisation of the TETRAD (TETRahedral ADaptation) code of Speares and Berzins [18], a general purpose, three–dimensional,

unstructured tetrahedral mesh h-adaptation algorithm. The parallel version of TETRAD, PTETRAD, utilises a mesh partitioned and distributed over the processors of the parallel machine. The solution and adaptation algorithms proceed independently on each processor with communications occurring at intervals to ensure consistency of both the mesh and the data across the machine. The parallel solvers using PTETRAD employ data parallelism using the commonly used 'owner computes' rule (see [5] for example). This paper follows on from earlier work that considers static partitioning [15], dynamic repartitioning [22] and some algorithmic and data-structure issues [16] for adaptive meshes in parallel. A comparison between the algorithm discussed here with other approaches to parallel adaptation may be found in [8].

In order to address the parallelisation of the TETRAD code and consider the issues that arise with regard to portability and scalability, Section 2 will describe TETRAD, and give a description of its complex serial data-structures. A discussion of how these data structures are parallelised is given in Section 3. The issue of data consistency in parallel mesh refinement is addressed in Section 4. Section 5 considers the parallel implementation and portability issues while Section 6 describes how the load is balanced throughout the parallel computation, from the initial mesh partition to how such partitions are adjusted as the mesh adapts. The choice of partition is a very important factor determining the scalability of the entire solution process as is discussed in Section 7. Section 8 describes the test problem used to illustrate the scalability of the code and shows results illustrating both the success and weaknesses of the approach described. These results are used to consider how the scalability is affected by the partitioning, both for the solver and for the adaptation. This paper concludes with a summary of the present approach and by identifying future work.

2 TETRAD

2.1 Data Structures

TETRAD is a general-purpose tetrahedral-mesh adaptation code which can be used to support a variety of numerical schemes e.g. [18, 9]. In order to ease the complexity of using unstructured meshes and to support a range of solvers, a rich data-structure is utilised in TETRAD. Figure 1 shows that the nodes, edges, faces and elements in a tetrahedral mesh are all stored, with their natural connectivities. Each node makes use of a one-way linked list of surrounding elements to describe the connectivity within the irregular mesh. Nodes and faces are stored in a two-way linked list and elements are organised as a tree. The tree is not uniform in that a refined element can produce varying numbers of child elements, depending on how the refinement is done. Edges are stored in a series of two-way linked lists (one per level of refinement) with additional parent/child pointers. Storing the mesh hierarchy reduces derefinement to simply a matter of removing a set of elements without reconstruction of a coarser mesh. This simplicity and speed is offset by the extra memory requirements of storing the hierarchy of nodes, edges, faces and elements. Solver specific data is attached to mesh objects such as nodes or elements by a void pointer, thus allowing the data-structure to store auxiliary information required by the PDE solver at edges, nodes or element centroids. This flexibility makes it possible to use TETRAD with a variety of finite-element or finite-volume methods, e.g. both cell-vertex finite element and volume schemes, Tomlin et al. [19] as well as the cell-centred schemes described here and incompressible flow algorithms under development. Other general purpose codes such as those of Flaherty [6] and Biswas [4] have, to the

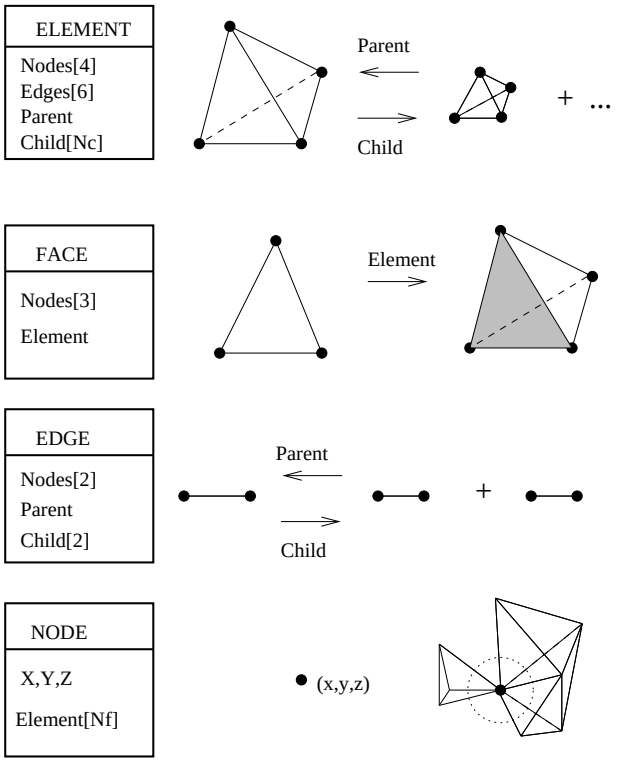

Figure 1: Mesh Data-Structures in TETRAD

best of our knowledge, similar storage requirements.

2.2 Refinement Types

There are two categories of refinement used in TETRAD. Elements with all their edges marked for refinement are refined into 8 'regular' child elements. This is illustrated in Figure 2. The choice of internal diagonal is chosen in the manner of Ong [14] in order to ensure that geometrical mesh "quality" is retained. Elements with five or less edges marked are refined in a so-called 'green' manner. This proceeds by creating a new node at the centroid. All the nodes in the element (including those created by refining the marked edges) are connected to this new node. This creates between 6 and 14 new elements, depending on how many edges were refined. An example of green refinement is shown in Figure 3. Green elements thus perform the task of ensuring the mesh is conforming and are used as a transition between area of differing levels of refinement.

Green elements are typically more geometrically distorted [14] than regular elements and thus are not refined further to prevent further possible distortion. In the situation where error estimates require further refinement of a green element, the element is first derefined and subsequently re-refined regularly. This procedure may have a knock-on effect. Consider the situation illustrated with triangles in Figure 4. If edge a is marked for refinement, the triangle T_1 will have to be refined. As T_1 is green it is must be first derefined and subsequently regularly refined. This means that the edge b, shared between triangles T_1 and T_2, must be refined. Consequently, T_2 must be derefined and re-refined as it is also green. The same knock-on effect also occurs with tetrahedral meshes and is catered for in the serial code by a

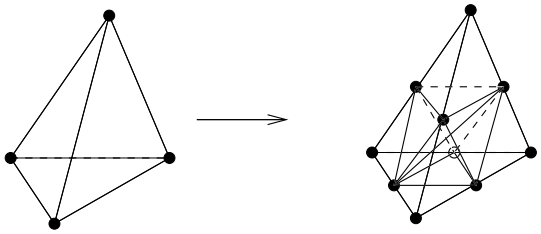

Figure 2: Regular refinement into 8 children

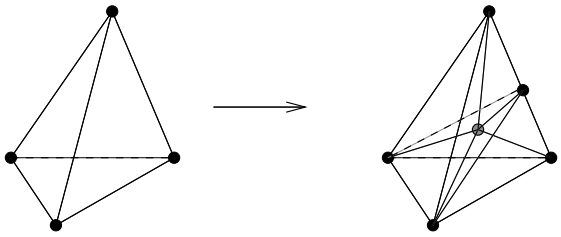

Figure 3: Green refinement into 6 children resulting from refinement of one edge

depth-limited recursive search. This search runs through the list of green elements that are to be derefined and subsequently re-refined. For each of these elements, a check is made of all the edge adjacent elements. If these elements are green and are a level coarser than the original green element they are added to the list of elements to be derefined and the search continues recursively with the new element as the starting point. The search thus progresses spatially through the mesh and terminates upon reaching base elements.

An important issue is that although the green refinement approach adopted here preserves the geometrical quality of the tetrahedra, the general issue of mesh quality is considerably more complex for non-isotropic solutions and involves not only the discretisation error but also the norm, [2, 3], in which the error is to be controlled.

3 Parallel Data Structures

The parallelisation of TETRAD has required the consideration of two main data structure issues. The first being how to partition the hierarchical mesh and the second being the provision of support for communications between the subdomains of this partitioned mesh.

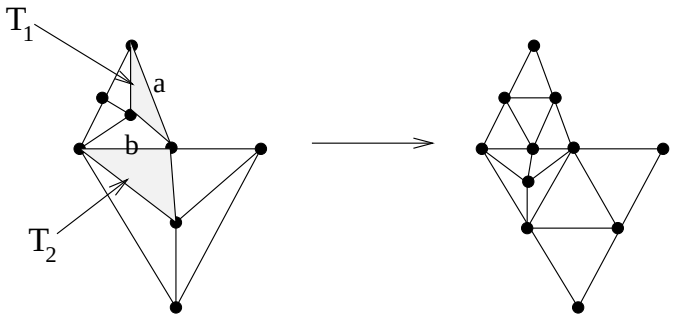

Figure 4: Knock-on effect of refining green elements for a triangular mesh

3.1 Parallel Data Structure Partitioning

There are two main options for partitioning a hierarchical mesh. The first is to partition the grid at the root or coarsest level. When using standard partitioning tools such as Chaco [7] or Metis [11] this requires weighting the dual graph node corresponding to the coarse mesh element by the descendents of that element in the computational mesh. This approach has the advantage that the local hierarchy of elements is maintained on a processor and thus all parent/child interactions (such as refinement/derefinement) are local to a processor. The partitioning cost is fixed, because the base mesh never changes, and low, as the base mesh is generally small in comparison with the adapted computational mesh. The main disadvantage with this approach however is that for small coarse meshes with large amounts of refinement, it is difficult to get good partitioning, both in terms of load balance and in cut weight minimisation, due to the large and varying weights that will be used.

The second approach is to partition the leaf level mesh, i.e. the actual computational grid. The pros and cons of this approach are essentially the opposite of those with coarse mesh partitioning. In particular, the quality of the partition in terms of cut weight and load balancing is likely to be better, albeit at the expense of a longer and non-constant partitioning time. Also, the data-structures have to be more complicated as hierarchical operations (such as multigrid V-cycles) are no longer necessarily local to a processor. This would, for example, lead to slower derefinement as communication may now be required. Fine mesh partitioning in a hierarchical data structure would also add an extra level of complexity to the coding due to the non-local operations in adaptation.

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

3.2 Communication Support

Given a partitioned mesh, new data structures are required in order to support the inter-processor communications and to ensure data consistency. In common with many other parallel PDE solvers, e.g. [5], so-called halo mesh objects (alternatively known as ghost or shadow objects) are utilised in order to reduce communications overheads. Halos act as a form of communication cache, being a local copy of remote data that is needed frequently. The choice of which halo data to have is solver dependent. Typically, for a given element on a partition, all elements in the computational stencil will have a halo copy on that partition. For example, Figure 5 shows a first-order cell-centred scheme that has a halo consisting of triangles (tetrahedra) that share an edge (face) with any triangle (tetrahedron) on the edge of the partition. In order to ensure that local data structures are complete, halos are kept of all edges, nodes and faces that compose an element.

Possible data inconsistencies due to multiple copies are resolved by assigning each object that is a copy the same owner as the original object. Thus halo objects are not owned by the processor on which they reside. In situations where halos may have different data than the original, the original definitive value is used to overwrite the halo copies.

The updating of halo data (e.g. at the end of a time-step) requires communication between a mesh object and its copies. The remote locations of halos are stored in a one-way linked list as it is not known *a-priori* how many halos any given mesh object might have. Each link in the list consists of a processor–pointer pair that locates a halo on a remote processor. This

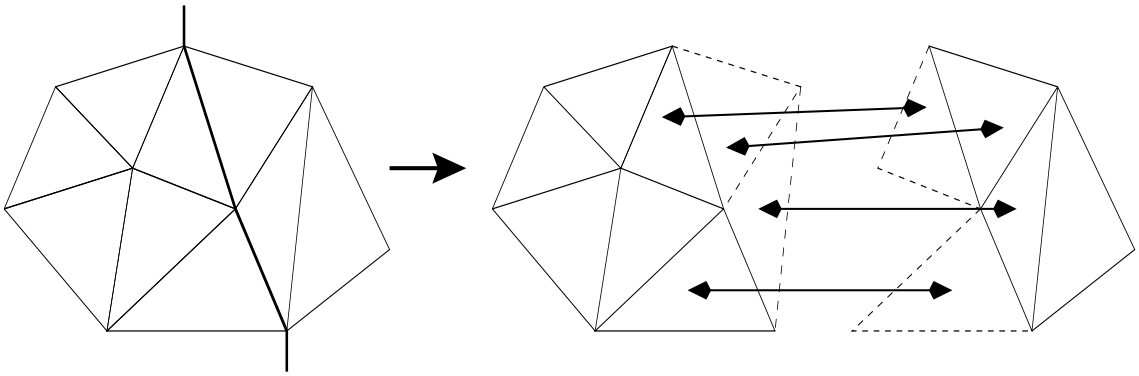

Figure 5: A 2D partitioned mesh and the distributed structure. Halo elements are dashed and arrows indicate communication links

method eliminates searching on the remote processor and ensures efficient halo updates. When adapting the mesh however, new elements may be created in the inter-partition boundary region. These elements will have (or be) halos and thus new communication linked lists will need to be created. To do this, new halo objects have to send their location to their original in order to create the linked list.

The communications links (backward) are set up by utilising the existing hierarchy to locate the copies of new data on remote processors. Each processor loops around the newly created data (owners have been decided by now), and if the owner is remote, packs a buffer with the pointer to the local information as well as parent pointers (for the remote processor) and any further information that is required to traverse the remote hierarchy. The communications cost is approximately the same as setting up the forward communications links in that there are the same number of messages to/from the same processors. This communication is in the opposite direction to the links already established, and thus without adding a new communication link (from halo to original) the original mesh object would have to be searched for on the remote processor. This is inefficient and thus halos also store a pointer to their original mesh object. The arrows in Figure 5 illustrate the communications links stored in the parallel data structure.

4 Parallel Adaptation Algorithms

The general structure of the parallel adaptation code follows that used in the serial case, albeit with added communications and synchronisation phases, and is illustrated in Figure 6.

The initial mesh is read from file, complete with a partition vector, on a single processor and then distributed across the parallel machine together with halo copies of mesh objects required to complete the subdomain on each partition and provide support for communications. Alternatively, a ready partitioned mesh (complete with halos) can be read in on each processor. This second approach uses rather more disk space and more files than the first, but is quicker and overcomes the limitations of a single processor's memory that the first approach has. Once the initial mesh has been read in, the refinement criteria to be used throughout the calculation are applied to the initial mesh and the mesh refined. The initial condition is then re-evaluated on the revised mesh and the process continued until the mesh fits the initial conditions.

TETRAD uses a similar adaptation strategy to that of Lohner [12]. Following a time-step, edges are marked for either refinement or derefinement based on some solution derived criteria. This may be either an error estimate or simply utilise gradients of solution values. The first communication phase occurs here in order to ensure that halo edges are marked in the same manner as their original edges.

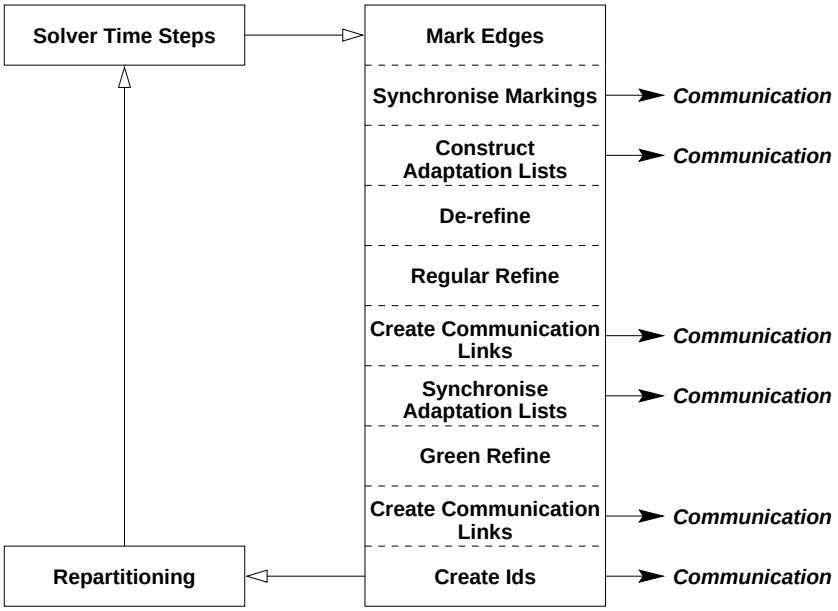

Figure 6: Overview of PTETRAD Adaptation algorithm

4.1 Green Tetrahedra and Recursive Searching

Once all edges are consistently marked, the elements and edges adaptation lists are constructed. Separate lists are generated for derefinement and regular and green refinement. This list construction enforces the adaptation rules that are used, such as not refining further an element that is currently refined in a green manner. The recursive search described in Section 2.2 is then used to ensure that all elements are refined in the correct manner.

The spatial nature of the parallel search is problematic as merely running the serial search in each subdomain is not sufficient, given that the search may well leave the subdomain. To allow for this, any edges encountered in the search that have halos are stored on a list (together with relevant information for continuing the search remotely) for later processing. Once the searches have terminated, the stored list of edges is exchanged with the relevant processors, and the search restarts on these halo edges. This process repeats until *all* processors have empty lists for communication. This is necessary as the search may later return to processors that previously have finished searching. This results in some processors possibly being idle during this stage of the adaptation process. As in the serial case, the search (and hence number of data exchanges) is limited by the depth of refinement in the mesh. Getting this search to work correctly is very important as this ensures that all elements (and their copies) are marked consistently for adaptation and thus avoids the need for much explicit consistency checking.

4.2 Adaptation algorithm

Once the final, consistent adaptation lists have been computed, the adaptation proceeds in a similar manner to the serial code. Derefinement is followed by regular refinement on elements and on any copies independently. The assignment of ownership to newly created data, unlike the serial case, cannot always be done consistently without communication as it likely that a given processor will not have enough information at the boundaries of the mesh it holds. Thus it is necessary to have an explicit update of ownership for edges on inter-processor boundaries to ensure that these edges and their copies have the same owner.

All the newly-created data that is on the inter-processor boundaries then has its communication links created. This is termed the *links phase* and is problematic as the communications links necessary for doing this are precisely the ones that are in the process of being created. By utilising the mesh hierarchy however it is possible to solve this problem. For example, communications links for new elements are created by using the existing links in place for the parent elements. Similarly links can be created for new edges and nodes by again using parent links to a higher level edge or element. These new links are then utilised to create the links from halos to the original mesh objects.

The next stage of the algorithm is to construct a list of edges that have been refined but whose halos have not. These halo edges are then refined in order to complete the mesh consistency. Such a situation exists as some edges are removed from the original refinement lists during derefinement of elements that on the inter-processor boundaries. The final refinement stage is that of green elements where hanging nodes are removed and the final adapted mesh is completed. As for the regular elements, this refinement is done independently until communication links for the newly created halo data are set up. The final stage of the algorithm is to assign globally consistent numeric IDs to new data. Again this process requires a communication to ensure that all halos of a mesh object have the same ID as the original.

5 Parallel Implementation and Portability

5.1 Coding Issues

Parallel TETRAD was implemented using ANSI C with MPI [13] due to the need for portability. The low-level of a message passing approach is not ideal for the complex data-structures and large amounts of communication involved in parallel adaptation. This type of application (with irregular, unstructured data) is currently poorly supported by libraries and compilers however and message passing is the only real option. As far as possible, communications are performed by using nonblocking MPI functions to avoid deadlock and allow a degree of overlap between computations and communications. Following the model in [15], communications are coalesced wherever possible to minimise total latencies and maximise bandwidth usage.

The style of programming used is similar to that of the BSP [23]. Each processor works in 'supersteps' which are separated by communications phases when data is updated across processors. Unlike the BSP paradigm, however, not all the data is held consistently at each superstep boundary. In derefinement, for example, halo edges may be removed from adaptivity lists as the processor does not have enough information to make a correct decision. This is corrected at a later stage, but only after all the regular refinement and subsequent construction of communications links has taken place. While the derefinement and regular refinement proceed without communication (they are essentially a superstep) the construction

of communication links involves much message-passing and would be therefore a superstep boundary. The synchronisation of the edge refinement lists takes place after this and thus does not fully conform to BSP style.

The major difficulty encountered in developing parallel adaptation routines using message passing is that it is very difficult to maintain consistency between mesh objects and their copies. Debugging situations where inconsistency occurs are particularly awkward as the inconsistency may not cause problems until the end of the superstep where communication occurs. Moreover, this problem tends to manifest itself in that send and receive buffers will not match up in size (as communication is coalesced) and discovering exactly which mesh object causes the problem can be time consuming. While debugging tools such as SGI's Workshop Debugger are helpful, they do not yet have the same ease of use as their serial counterparts. The lack of high level language support for irregular parallelism also adds to the programming difficulties and is one of the major obstacles for the more widespread use of irregular parallelism.

5.2 Order Independence

In implementing mesh adaptation it is necessary to ensure that all the adaptation functions are order independent. That is, they give the same final mesh regardless of the order in which elements are processed. This is vital as elements and their halo copies may well be processed in different orders on different processors. Serial TETRAD has a number of cases where processing order can affect the final mesh. One such example is illustrated in Figure 7. Here, a face is illustrated which is to be refined in a green manner (as only two edges are refined). There are two ways of doing the refinement, both of which result in refined meshes of equal quality. The serial code picks the first of these refinements that it encounters. Doing this in parallel however can result in the faces being refined in different ways on different processors. In particular, this can result in differing connectivity in the mesh across different processors. This inconsistency is clearly unacceptable and is eliminated by utilising node co-ordinates to impose an order in such situations. There are a number of such order dependent code segments in the serial code which have required careful analysis and replacement so that the parallel code produces a consistent mesh across the processors.

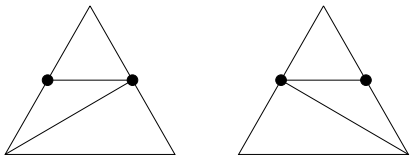

Figure 7: Examples of Green Face Refinement

5.3 Portability

The portability of PTETRAD as coded in ANSI C with MPI, has been demonstrated by moving the code to a variety of platforms including a Cray T3D, an SGI PowerChallenge, and SGI Origin 2000 and on workstation networks (SGI Indys and SGI O2s). Only one line is changed for the differing machines and this is purely for memory efficiency. The Cray T3D is a 64 bit machine and to obtain the same accuracy for `double` and `int` variables as for a 32 bit machine, one need use only `float` and `short`. This can result in halving the memory needed

for all the physical solution and many of the data structures variables. This is achieved simply by using `itype` and `dtype` rather than `int` or `double` in variable definitions. Depending on a `#define` statement in a globally included file, `itype` and `dtype` can be set to the required accuracy by a `typedef` statement. At the same time, the constants `MPI_DTYPE` and `MPI_ITYPE` are set to the relevant MPI datatypes. For example, `MPI_DTYPE` is set to either `MPI_DOUBLE` or `MPI_FLOAT`. This technique is similar to one used in ParMetis [10] and allows us to use larger partitions on the Cray T3D than would otherwise be possible.

Despite this technique, the overhead from 64 bit pointers on the Cray is quite substantial. For example, every element stores 12 pointers (4 nodes, 6 edges, children and parent) and leaf elements are likely to store more for additional connectivity as required by the solver. Edges, nodes and faces also use a large number of pointers. The cumulative affect of the extra memory used, together with the fact that in order to be general the data-structures used are large, is to limit the number of elements that can be stored on a single T3D processing element with 64MB of memory to around 45,000, while more application-specific codes with lightweight data structures may be able to store at least five times as many elements per processor.

6 Parallel Load Rebalancing

In any parallel unstructured mesh calculation, load balancing is a very important issue; for adaptive meshes it is doubly so. Not only is there the requirement that the initial mesh is partitioned in a suitable manner but it is also necessary for the mesh to be repartitioned periodically due to the imbalance created by the mesh evolution. This is particularly true when solving PDE problems whose solutions involve fast moving shocks; the imbalance created by the constantly evolving mesh will require frequent repartitioning to bring the imbalance down to acceptable levels.

6.1 Initial Partition

The initial mesh partitioning is based on ensuring that the number of base mesh elements in each subdomain should be equal (or as near to it as possible) and that the number of halo elements should be minimised. This latter condition is to ensure that communications are minimised. Given sufficient information about the problem (such as initial conditions and spatial and temporal gradients) it is possible to weight the mesh so as to anticipate where future refinement will be needed for the partitioning problem. This will ensure that after the mesh has adapted to the initial condition, it is not too ill-balanced. This is also desirable in that many dynamic partitioning tools make an assumption that the initial mesh partition is reasonable. The partitioning problem is NP complete and thus a variety of differing heuristics have been invented that give a reasonable partition in good time. Many of these have been released as software tools, such as Jostle [24], Metis [11] and Chaco [7] and all have been used successfully with PTETRAD.

6.2 Dynamic Repartitioning

As the mesh adapts it is of critical importance to rebalance the calculation periodically. The requirements for this partitioning problem are rather different from the static case. As well as requiring the repartitioned mesh to be load balanced with minimal communication, it is

also necessary to minimise the amount of data movement so that the total time taken in redistributing the mesh is minimised. Furthermore, the calculation of the new partition must be fast and run in parallel on the already distributed data. Examples of partitioners that fit the task are ParMetis [10] (PARDAMETIS and PARUAMETIS), Parallel Jostle [25] and Touheed's algorithm [21], all of which have been utilised in the parallel code, PTETRAD. The new partition is calculated on the base level mesh, with nodal weights corresponding to the number of leaf elements in the tree under each base element. Edge weights are calculated as either the number of leaf level faces between base mesh elements, or as an average of the nodal weights. Either of these gives a reasonable (and proportionate) approximation to the true weight of the cut.

As the base mesh is fixed and is only a proportion of the size of the whole mesh, the calculation of the new partition is not costly. The weights that tend to arise with adaptation however are often large and varying. This makes the repartitioning problem difficult and means that the resultant partition may not have perfect load balance. A comparison is given in [22] for the performance of ParMetis, Parallel Jostle and Touheed's algorithm for the same test problem used in this paper. Touheed's algorithm tends to move the least data and give the best balance, but does rather poorly for cut-weight. Parallel Jostle and ParMetis produce comparable results, with Parallel Jostle achieving slightly better cut-weights and load balance than ParMetis at the cost of moving more data. This is achieved largely due to the better configurability of Parallel Jostle. Both algorithms utilise graph coarsening algorithms to obtain speed. This may not be ideal, however, when dealing with a relatively small but heavily weighted graph such as those produced by adaptation. Parallel Jostle allows relaxation of the amount of coarsening done and this seems to give it the edge over ParMetis in terms final partition quality.

6.3 Data Redistribution

Redistributing the mesh to match the new partition and setting up the new halo elements is not a simple task. Each processor sends out copies of mesh objects that are either being moved or will be required as halos. This involves communication with as many neighbours processors as are specified by the partitioning software.

The actual moving of mesh data is done by first copying the data to the new processors on which it will reside and later removing repeated data that is no longer required. While this uses more memory, it is a more reliable and faster approach. In order to reassemble the mesh on remote processors, it is essential that mesh objects are copied in the correct order. Nodes have to be moved before edges (as edges point to nodes), edges before elements and elements before faces. As the mesh is unstructured, finding the relevant (for example) nodes for edges to point to requires some form of searching. This is achieved efficiently by using a randomised hash table of node addresses, indexed by the node IDs. This provides a good compromise between memory usage and speed (essential as such searches are frequent). A similar technique is used to avoid replication of mesh objects which would occur when copies are sent from multiple processors. As with adaptation, communications are coalesced wherever possible in order to minimise their overhead. Once all the mesh objects have been copied, any that are no longer required are deleted.

The final task in redistribution is a two-phase process to ensure that all mesh objects and their halos have correct communication links. First, the communication links that are no longer valid are removed, and then the new links are created. There are two options

for removal of invalid communication links. One is to remove only those links which are invalid. Due to the possible inaccuracy of existing links however, this requires global communications for each processor. The alternative method removes *all* communication links before reconstruction. Although this uses larger point-to-point messages in reconstructing communications links, it eliminates global communications and is both quicker and more scalable.

7 Scalability Discussion

The scalability of an adaptive mesh solver is dependent on all three of the main components: mesh adaptation, repartitioning and the solver itself. The solver used here has good scalability characteristics e.g. see [22], and so the focus here will be on mesh adaptation and redistribution. The overall scalability is affected by the scalability of each of these two components and how often they are each invoked. Adaptation is used either when the estimated error exceeds some tolerance, or at regular intervals if (for example) one knows *a-priori* the speed of the main shock in the calculation. Thus fast flows, e.g. [18], will require more frequent adaptation than slow flows e.g. [9]). Moreover, with more adaptation stages, load balance will be destroyed more rapidly and thus repartitioning will be also be required more frequently. It is therefore clear that for fast explicit solvers for high-speed flows, the scalability of the adaptivity and repartitioning will be of greater significance than for solvers with a lot of intensive calculation (such as reactive flow where chemical reactions are modelled along with the fluid flow) and slow flows, for which frequent mesh adaptation and redistribution is not needed.

7.1 Mesh Adaptation

The scalability of the PTETRAD mesh adaptation algorithm is a complex matter. with the choice of partitioning affecting the efficiency of the adaptation calculations. An ideal partition has low cut-weight to minimise the communication involved. Indeed, this is a more important issue in adaptation, as there is far more communication involved in setting up the communication links for new data and ensuring consistency of the distributed mesh than there usually is in a well designed parallel solver.

It is also desirable to balance the load of adaptation equally across the parallel machine. The cost of repartitioning for adaptation is prohibitive however and thus the existing partition for the solver is utilised. The disadvantage of this is that adaptation may only occur on a few processors. When the meshes on these processors are heavily refined there is a lot of communication in establishing the new communication links and ensuring consistency of the various mesh copies. Thus communication is likely to be a limiting factor in the scalability of the adaptation algorithm.

A further scalability issue is that if the initial mesh is fixed, but the number of processors is increased, then each processor will have a smaller number of elements, and more importantly a higher proportion of halo elements and hence communication. This however can be avoided by scaling the *base* mesh with the number of processors.

7.2 Repartitioning and Redistribution

Repartitioning and redistributing data are an integral part of a good dynamic parallel code, but are a significant parallel overhead. As repartitioning consists almost entirely of commu-

nication, it tends not to scale well. Indeed, the more processors are used, the more quickly imbalance is generated by the adaptation and, if repartitioning is triggered only on exceeding an imbalance threshold, the more repartitioning is done. It is not at all clear that this is the right approach for triggering repartitioning however as for problems with large numbers of processors with modest sized grids and fast flows, the gain in run-time speed from using a load-balanced solver may not be better than the time taken by the repartitioning process. It may be better however when also considering the improvement gained by using a new partition for adaptivity, but this may be difficult to predict. Where there is more work for a processor to do before an adaptive step (such as with large grids, small numbers of processors or slow flows), the load balance is more critical to overall performance and repartitioning will pay off. In these cases it may even be appropriate to repartition after every adaptation.

8 Scalability Experiments and Analysis

8.1 Test Problem

The problem we have used to test the parallel adaptivity is that described in Speares and Berzins [18]. This is an inviscid Euler equations gas jet flow problem modelling the shock wave diffraction around the 3D corner formed between two cuboid regions. The test problem arose as part of an industrially-funded study into gas jets. This is an ideal problem for mesh adaptation as the shock moves through the domain and a static mesh would require high resolution throughout. It is also a good test for parallel robustness as the shock moves through the spatially partitioned mesh and repartitioning of the mesh will be essential and frequent. The initial condition for the problem is of Rankine-Hugoniot shock data at the interface of the two cuboid regions in the domain with an initial shock speed of Mach 1.7. The solution is computed with a cell-centred, Riemann-problem-based, finite-volume scheme of the MUSCL type, employing an HLLC style Riemann solver. Full details of both the problem and the (serial) solver can be found in [18].

The parallelisation of an explicit cell-centred finite-volume solver such as the one used here is relatively straightforward given the data-structures designed into the parallel adaptation code. Each processor holds solution values for elements in its domain, with copies of solution values on the halos as required by neighbouring sub-domains. The calculation proceeds using an 'owner computes' rule. Since the halos were chosen appropriately, this allows the the computation to proceed with minimal communication. There are two updates of solution values per time-step, after the Hancock half-step and at the completion of the full time-step. At both times, only the changed data are updated, with boundary gradient information calculated with the new solution values, rather than included in the communication. The only other communication in the time-step is to ensure that all processors are using the same sized time-step. This is necessary as CFL constraints are different on differing subdomains due to the varying mesh sizes and solution characteristics.

The form of the developing shock solution and an indication of how the mesh adapts to is given in Figures 8 and 9. Figure 8 shows the base mesh adapted to the initial shock condition, and Figure 9 shows the solution after the solution has started developing. The Figures show the mesh on the boundary (with three boundary walls cut away for ease of viewing).

The scalability experiments use a Cray T3D with 64MB of memory per node. Two different base meshes are used; one with 5,184 elements and the other with 34,560 elements. The code has been used with much larger base meshes and higher resolutions, but due to the

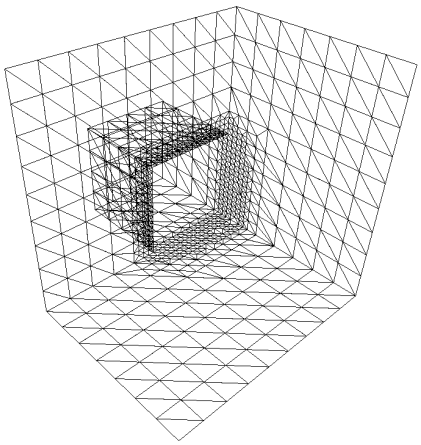

Figure 8: Coarse mesh of $5,184$ elements adapted to initial shock condition.

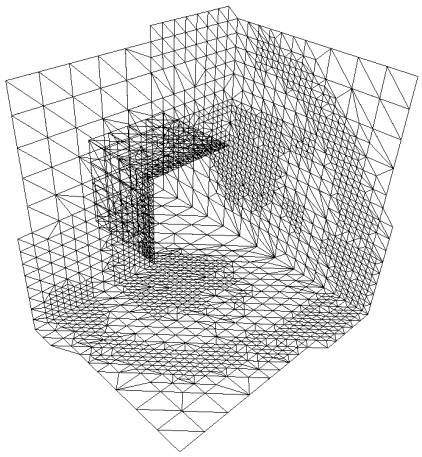

Figure 9: Adapted mesh after 540 time-steps

per node memory limitations of the T3D, they are not very useful in illustrating scalability. This storage limitation is not such an issue on machines such as the Origin 2000 with much larger per-processor memories.

Scalability experiments have used used up to three levels of refinement from the base mesh. For the smaller base mesh, the computational mesh resolution is equivalent to a regular mesh of 41,472 elements for one level of refinement, 331,776 elements for levels and 2,654,208 elements for three levels. This is achieved by using meshes of 12,391 elements, 3,860 elements and 87,082 elements respectively. Similarly, the larger base mesh has fine mesh resolution of 276,480 elements with one level of refinement, 2,211,840 elements with two levels and 17,694,720 elements with three levels achieved by using meshes with 49,716 elements, 97,481 elements and 295,275 elements.

The experiments all use Parallel Jostle for repartitioning with the coarsening threshold set to 300, [24], as experience indicates this is the most suitable value for this example.

An important feature of this supersonic flow calculation is that the solution changes rapidly, mesh adaptation is required frequently. In this case the mesh is adapted every fifteen time-steps. Moreover, the solver uses explicit time integration, is computationally light and so scales well. This means that the scalability of the adaptation is very important to the scalability of the whole process. The decision when to repartition is also very important as frequent remeshing with an inexpensive solver is precisely the most difficult case encountered. There are clearly better choices for when to repartition, but here repartitioning is performed once the imbalance has exceeded a certain percentage threshold.

It is also worth considering how the scalability is affected by using solvers with differing amounts of work per remesh. For example, by using a solver for reacting flows with multiple chemistry species (such as that considered in [9]). In order to simulate the differing amounts of work required per remesh, total timings can be recalculated to give more weight to the solver contribution to the total timings. This is equivalent to running the existing solver for larger numbers of timesteps and thus is a fair comparison as the solver communication times are increased with the computation times. The present solver is only first order in both space and time; the use of second order methods would increase the cost by a factor of at least four per timestep. Furthermore, in the present problem, remeshing takes place after very few

timesteps, whereas in some calculations such as that of Tomlin et al. [19] and Johnson et al. [9] it has proved possible to use the same mesh for many hudreds of timesteps.

8.2 Computational Results

Results are shown in Figures 10 and 11 for the scalability of the entire solution process on a mesh of 34,560 base elements with 1 and 2 levels of refinement respectively. The scalability is shown for calculations with 15, 150 and 1500 time steps per remesh. In all these cases, the solution has been calculated with a number of different thresholds for repartitioning. The best result is used in each case. The timings are scaled (to 100 for the smallest number of processors used) in order that the graphs for differing amounts of work may be compared.

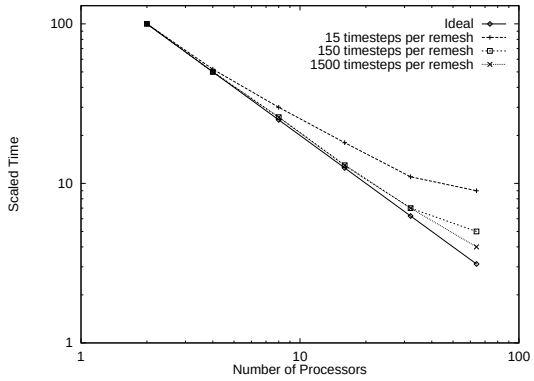

Figure 10: Scalability for 34,560 element mesh with 1 level of refinement

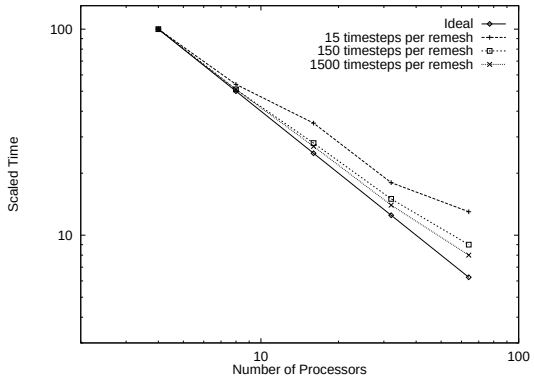

Figure 11: Scalability for 34,560 element mesh with 2 levels of refinement

It is clear that in both cases, scalability improves as the number of time-steps per remesh increases. This is not surprising as the amount of parallel overhead in a remesh is large both from the communication inherent in adaptation, but also as frequent remeshing tends to lead to frequent redistribution (which is an entirely parallel overhead). Similar behaviour is found in Figures 12 and 13 which illustrate scalability for a mesh of 5,184 base elements with 1 and 3 levels of refinement. The scalability for 64 processors is rather poor in this case however, with a slowdown for 1 level of refinement and small amounts of work per remesh. For Figure 12 this poor performance is largely due to a lack of work on larger numbers of processors and the comparatively large repartitioning times. The performance improves considerably with more work inbetween each remesh, but is still limited by the overall compute/communications ratio in the solver. A further factor (which also affects the scalability in Figure 13) is that a base mesh of only 5,184 elements is rather small for 64 processors, giving a large halo/computational element ratio. As was mentioned in Section 3 this is to be expected with a base mesh partitioning if the base mesh is kept fixed.

An issue that requires more consideration is that of which repartitioning threshold is most appropriate for a given problem size and number of processors. Table 1 gives timings for a number of thresholds for 2 levels of refinement of the smaller base mesh. For 15 and 150 timesteps per remesh, a 25 % threshold is most efficient as the time required by the extra repartitioning steps in the 10 % case is prohibitive. For the 1500 timesteps case, however, the gain in solver efficiency of the 10 % case is enough to overcome the extra partitioning

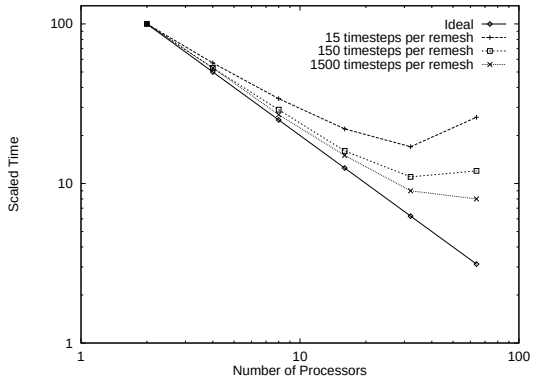

Figure 12: Scalability for 5,184 element mesh with 1 level of refinement

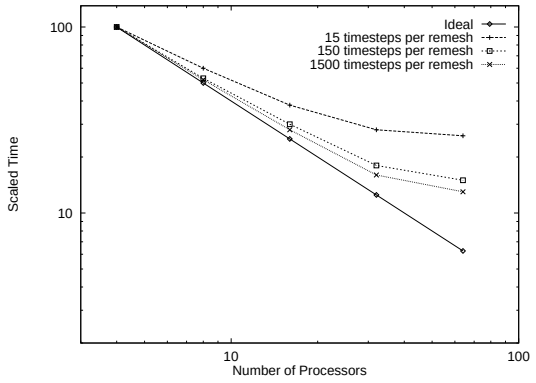

Figure 13: Scalability for 5,184 element mesh with 3 levels of refinement

Threshold (%)	Number of Repartitions	Time (sec) (15 ts)	Time (sec) (150 ts)	Time (sec) (1500 ts)
10	16	424.5	2011.8	17968.7
25	5	319.5	1973.5	18595.4
40	5	349.1	2108.8	19784.9

Table 1: Timings for 5,184 element base mesh, 2 levels of refinement, 16 processors, varying repartitioning thresholds, and for 15, 150 and 1500 timesteps per remesh

costs incurred. Other cases exhibit similar behaviour in that the more work there is between remeshes, the more vital load balance becomes.

An interesting comparison can be made of the scalability of the individual parts of the entire algorithm. One would not expect the adaptation to scale in the same way as the solver given the different nature of the algorithms and amounts of communications involved. In a similar manner, repartitioning (a similarly communications intensive process) will scale slower than the solver. Figure 14 gives a comparison of scaling for the final adaptation, repartitioning and computations phases in a calculation with 10 % repartitioning threshold on a mesh with 34,560 base elements and 2 levels of refinement.

The solver clearly scales well, as would be expected for an explicit finite-volume scheme. The adaptation and repartitioning scale at a similar rate to each other, but at a much lower rate than the solver. This is not surprising as they are far more communication intensive and do not have the work involved evenly distributed. It is interesting to note that in this case the repartitioning scales smoothly (this is not universally the case) while the adaptation is is rather oscillatory. This is likely to be a consequence of how well the partition happens to suit adaptation. In some cases (those with higher levels of refinement) the adaptation and repartitioning scale less well again due to the further increased communications and repetition of work involved in adapting the halos.

Since we have obtained these results, similar results have been obtained by Touheed [20] using the same problem on a 32 processor Origin 2000. These results using the larger base mesh and Jostle are shown in Table **??**.

All times are in seconds. From this table we see that the migration frequency grows with

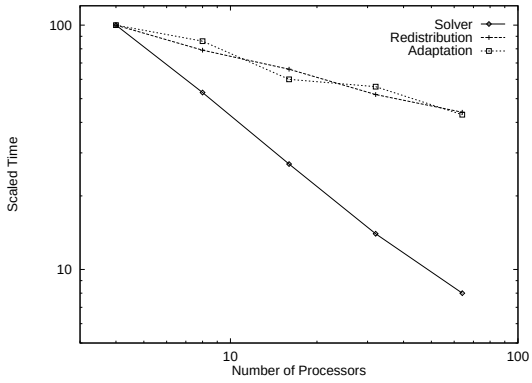

Figure 14: Scalability comparison of the individual parts of the solution process

P	2	4	8	16	32
Solver Time	2873	1475	833	430	211
Redistribution Time	37	19	42	63	78
Migration Frequency	1	1	2	3	7

Table 2: Timing Results on 32 Processor SGI O2000

the number of processors if migration is invoked after a fixed percentage imbalance (10 %) in this case.

8.3 Analysis of Scalability of Adaptation and Redistribution

An analysis of the scalability of the adaptivity is not straightforward but may be summarised by stating that relatively few processors are involved in refining elements and subsequently distributing relatively large amounts of data.

In order to explore this issue in detail the example used above will be considered in detail in the case of the first significant mesh adaptation after the start of integration. The base mesh is 34,560 tetrahedra and two levels of refinement are used giving a mesh of about 97,000 tetrahedra. For this case, the proportion of processors involved significantly in adaptation P_{adapt} is $\frac{3}{4}P$. This is approximately constant as P increases. Table 2 shows the average halo size $AvgNh$ and the maximum halo size $MaxNh$ for the number of processors P . The average size changes proportionately to 1/ log(P). The individual timings for the parts of the mesh adaptation algorithm are given by Table 3 in which the following abbreviations are used:

- Flags - setting and updating adaptation flags and creating of adaptation lists as in

P	4	8	16	32	64
Max Nh	11425	12951	9296	6343	3926
Avg Nh	6703	6695	4649	3188	2174

Table 3: Comparison of numbers of halo elements

P	4	8	16	32	64
Flags	2.7	1.8	1.2	0.8	0.6
Deref	2.0	0.9	0.6	0.4	0.2
Regular	4.3	2.4	0.9	0.4	0.4
Links	3.2	4.0	3.4	2.1	1.6
Green	3.8	2.3	2.2	1.8	0.7
Links	4.1	5.0	3.8	2.9	2.8
IDs	1.7	2.4	2.2	1.4	1.4

Table 4: Times for Mesh Adaptation Steps

P	4	8	16	32	64
Owner + halo	10.3	8.1	6.4	4.5	4.0
Data Movement	11.6	12.7	8.2	7.2	6.0
Connect	0.1	0.1	0.1	0.1	0.1
Communications	2.2	4.2	2.1	1.8	1.3

Table 5: Times for Mesh Redistribution Sub steps

Section 4.1.

- Deref - Derefinement of tetrahedra.
- Regular - Regular refinement of tetrahedra.
- Links - Creation of adaptation links forward and back. Used after both regular and green refinement, see Section 4.1.
- Green - Green refinement including synchronisation, see Section 4.1.
- IDs - creation of consistent IDs and final cleanup operations, see Section 4.2.

The individual timings for the parts of the mesh redistribution algorithm described in Section 6.3 are given by Table 4 in which the following abbreviations are used:

- Owner and halo - calculation of new partition (Jostle) and the new halo data.
- Data movement - copying of mesh objects across machine
- Connect - calculation of connectivity information and deletion of surplus data.
- Communications - deletion of old and creation of new communication links.

The two tables above show that the two key components in the relatively poor scalability of mesh adaptation and redistribution are the *Links* phase as shown in Table 3 and the *Data Movement* phase as shown in Table 4. Analysis of the links phase is problematic. This is the largest parallel overhead in the adaptation algorithm - most of the rest of the algorithm requires only relatively modest communication. The Links phases require a lot of communication as all new edges, elements, nodes and faces require links to be created -

P	4	8	16	32	64
N_{moved}	30249	51775	63611	85974	93075
P_{neigh}	3	7	10	12	13
N_{maxP}	8710	12098	7977	5329	4058

Table 6: Number of Elements and Number of Processors in Redistribution

both forward and backward. Moreover in this case (unlike in redistribution) consistent IDs are not known and cannot be used in a hash table to locate remote data, so rather more information about the hierarchy has to be sent in order to locate the data. This data requires more packing/unpacking as well as bandwidth. Suppose that a fraction β of the elements in a halo proportional to 1 / log(P) is being refined in the partition of any one processor. The time taken for this is thus proportional to β /log(P).

In the data movement phase for this example the number of elements moved (N_{moved}) and the number of processors (P_{neigh}) connected to any other is shown in Table 5. The maximum amount of data moved by any processor is shown by N_{maxP}. Many edges and nodes are also moved (in proportion to the number of elements moved) and the amount of data moved scales as $N_{moved} \approx \gamma\ log(P)$, where P is the total number of processors. For 64 processors, a large proportion of the whole mesh is moved (although not as many as initial inspection may suggest as many of the elements moved are halos). A comparison between Tables 4 and 5 shows that there is a reasonably good correspondence between the maximum number of elements that any processor has to move and the time taken for data redistribution. It is interesting to note that dynamic graph repartitioning algorithms minimise the total data moved rather than the maximum for a single processor, which this analysis suggests would be the more relevant metric.

9 Conclusions and Future Work

The parallelisation of an unstructured adaptive mesh code in a portable fashion involves careful consideration of complex irregular data-structures. Maintaining consistency of this data is particularly difficult. Moreover, the necessity to code at the message passing level for efficiency purposes does not lead to transparent, easily maintained software. The parallelisation of TETRAD has been achieved however and the strengths and weaknesses of the approach have been demonstrated for a testing shock problem. The adaptation itself scales in a modest manner due to the amount of communication involved in maintaining consistency of halo mesh objects and the cost of redistribution. It is thus not clear that present (re)partitioning algorithms address this issue as well as they might.

As part of the larger solution process however we have demonstrated that by choosing repartitioning thresholds carefully, good scalability may be achieved, providing that there is sufficient computation in between remeshing. It is thus clear that the problems most suited to parallel adaptive solution involve flows with larger amounts of computation per remesh (such as chemical reaction-advection problems) and fairly low levels of refinement (up to 3 levels appears to give the best results).

A key issue to emerge is that the total cost of repartitioning and solution must be considered when deciding to repartition and in particular that the amount of data moved when

repartitioning is an important part of this.

Further work has involved integration of the PTETRAD code with the SCIRUN steering system and the SPRINT integration code, [9] in order to solve problems of atmospheric dispersion and orographic flows. Work is also ongoing to develop a new programming abstraction based on Shared Abstract Data Types in order to simplify the process of developing and maintaining codes for irregular mesh problems [17].

Acknowledgements

The authors would like to thank Peter Dew, Jonathan Nash, Nigel Weatherill, Ken Morgan, Nick Verhoeven, Bill Speares, Peter Jimack and Nasir Touheed for discussions and collaboration with this and related work. The authors are also grateful to George Karypis, Bruce Hendrickson and Chris Walshaw for providing ever improving partitioning codes. The first author also thanks the UK EPSRC for financial support under grants GR/84915 and GR/L73104.

References

[1] R.E. Bank, *PLTMG: A Software Package for Solving Elliptic Partial Differential Equations. Users' Guide 7.0*, SIAM, (1994)

[2] M.Berzins, *"A Solution- Based Triangular and Tetrahedral Mesh Quality Indicator"*, SIAM Journal on Scientific Computing 19,6 2051-2060, 1998.

[3] M.Berzins, *"Solution- Based Mesh Quality for Triangular and Tetrahedral Meshes."*, pp 427-436 in Proceedings of 6th International Meshing Roundtable, Sandia Report SAND 97-2399, Sandia National Labs, PO Box 5800 MS 0441, Albuquerque, NM 87185-0441

[4] R.Biswas, L.Oliker and H.N. Gabow. *Performance Analysis and Portability of the PLUM Load Balancing System*, Euro-Par'98 Parallel Processing, Lecture Notes in Computer Science, Vol. 1470, Springer-Verlag, 1998, pp. 307-317.

[5] J. Cabello. *Parallel Explicit Unstructured Grid Solvers on Distributed Memory Computers.* Advances in Eng. Software, 23, 189 (1996).

[6] J.E. Flaherty, M. Dindar, R.M. Loy, C. Ozturan, M.S. Shephard, B.K. Szymanski, J.D. Teresco, L.H. Ziantz, *An Adaptive and Parallel Framework for Partial Differential Equations*, In D.F. Griffiths, D.J. Higham, and G.A. Watson, Eds., Numerical Analysis 1997 (Proc. 17th Dundee Biennial Conf.), pages 74-90, 1998.

[7] B. Hendrickson and R. Leland, *The Chaco User's Guide: Version 2.0*, Tech. Report SAND94–2692 Sandia National Labs, (1994).

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

[9] C.R. Johnson, M. Berzins, L. Zhukov, and R. Coffey. *"SCIRun: Application to Atmospheric Dispersion Problems Using Unstructured Meshes."* in Numerical Methods for Fluid Dynamics VI (ed. M.J.Baines), OUP, 1998.

[10] 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, (1997).

[11] G. Karypis and V. Kumar, *A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs*, Tech. Report TR 95-035, Dept. of Computer Science, University of Minnesota, (1995).

[12] R. Lohner and J. D. Baum, *Adaptive H-Refinement on 3D Unstructured Grids for Transient Problems*, J. Num. Meth. Fluids, Vol. 14, (1992), pp 1407–1419

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

[14] M. E. G. Ong, *Uniform Refinement of Tetrahedron*, SIAM J. Sci. Comp., Vol 15, No. 4, (1994).

[15] P. M. 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*, Proc. Parallel CFD '96, (1997).

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

[17] P.M.Selwood, M. Berzins, J.M. Nash and P.M. Dew, *"Portable Parallel Adaptation of Unstructured 3D Meshes"*, pp 56-67 in "Solving Irregularly Structured Problems in Parallel" Proc. of Irregular 98 Conference (Ed. A.Ferreira et al.), Springer Lecture Notes in Computer Science, 1457, 1998.

[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 (1997).

[19] A.S.Tomlin S.Ghorai G.Hart and M.Berzins *The Use of 3-D Adaptive Unstructured Meshes in Air Pollution Modelling.* in Z.Zlatev et al. (eds) Large Scale Computations in Pollution Modelling, 339-348. Kluwer Academic Publishers. (Proceedings of Nato Workshop on Air Pollution Modelling,held in Sofia, Bulgaria, 1998).

[20] N.Touheed. *Parallel Dynamic Load Balancing for Adaptive Distributed Memory PDE Solvers.* Ph.D. Thesis, School of Computer Studies, The University of Leeds, Leeds LS2 9JT, UK. March 1999.

[21] 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).

[22] N. Touheed, P.M. Selwood, M. Berzins and P.K. Jimack, *"A Comparison of Some Dynamics Load Balancing Algorithms for a Parallel Adaptive Solver "*, Parallel and Distributed Processing for Computational Mechanics II (ed. B.H.V. Topping), Saxe-Coburg, 1998.

[23] L. Valiant, *A Bridging Model for Parallel Computation*, Communications ACM, Vol. 33, No. 8, (1990).

[24] C. Walshaw, M. Cross and M. Everett, *A Localised Algorithm for Optimising Unstructured Mesh Partitions*, Int. J. Supercomputing Appl, Vol. 9, No. 4, (1995).

[25] C. Walshaw, M. Cross and M. Everett, *Mesh Partitioning and Load-Balancing for Distributed Memory Parallel Systems*, in Proc. Parallel & Distributed Computing for Computational Mechanics 1997, (ed. B. Topping), (1997). for Unstructured Meshes