

# Dynamic Load-Balancing for Adaptive PDE Solvers with Hierarchical Refinement

Nasir Touheed\*

Peter Jimack\*

## Abstract

Many adaptive techniques for the solution of both steady and time-dependent PDEs rely on a hierarchy of meshes: starting with a coarse global mesh and refining this by different amounts in different regions. Since this refinement (or de-refinement) is part of the solution process it is necessary to dynamically alter the way in which the mesh is partitioned if a parallel implementation of such an algorithm is to remain load-balanced. A technique for achieving this is described, based upon consideration of the coarse mesh.

## 1 Introduction

In this work we consider the dynamic load balancing problem which arises in the adaptive solution of time-dependent partial differential equations using hierarchical h-refinement. For the purposes of this paper we restrict our consideration to problems in two space dimensions of the form

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

where  $\Omega \in \mathbb{R}^2$  and  $\mathcal{L}$  is some spatial operator. We further assume that the spatial discretization is based upon a triangulation,  $\mathcal{T}$  say, of  $\Omega$  which is allowed to adapt with time (see the finite element method of [7] or the finite volume scheme of [3] for example). This adaptivity will be regarded as hierarchical in the sense that there is a coarse root mesh,  $\mathcal{T}_0$  say, whose elements are refined to different levels at different times: based upon local spatial error estimates. The refinement of the elements is recursive, with each element able to divide into four children (see figure 1), and neighbouring triangles are not allowed to differ by more than one refinement level (see [2, 7] for examples of this refinement strategy).

If we consider a distributed memory parallel solution algorithm based upon geometric parallelism and this hierarchical element data structure, then it is clear that at each time step we would like each processor to hold an equal number of leaf elements, so as to ensure that load-balance is achieved in the solver. Note that if we only partition elements of the root mesh then these will not generally be split equally amongst the processors, since each root element will contain a different number of leaf elements. An additional feature of a good partition is that the number of triangles along the partition boundary should be as low as possible so as to minimize the communication required in the solution algorithm.

If we assume that at a given time we have a good partition of the root mesh but that the leaf mesh is then subject to alteration through local refinement and/or coarsening, then the modified mesh will not generally be load-balanced. In order to re-balance the mesh

---

\*School of Computer Studies, University of Leeds, Leeds, LS2 9JT, UK.

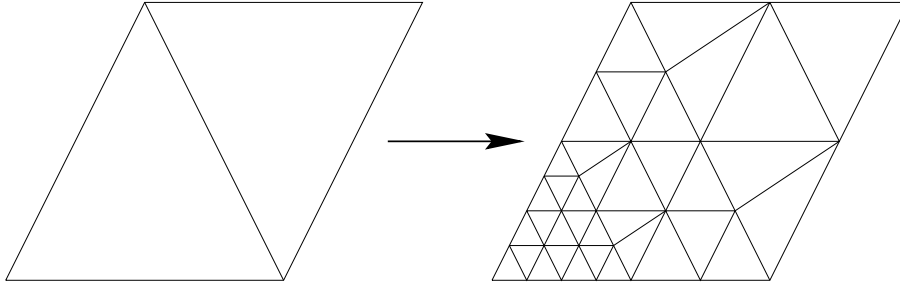


FIG. 1. *Refinement of triangles by recursive subdivision.*

dynamically it is necessary to pass root elements (and their children of course) between processors. This re-balancing should have the following objectives:

1. production of a load-balanced partition,
2. a minimal amount of data migration,
3. a minimal number of elements ending up on the partition boundary,
4. the possibility of an efficient parallel implementation.

In the following sections we outline a new dynamic load-balancing algorithm which is designed to meet these objectives: it is based upon earlier work of Vidwans *et al* [12]. It should be noted that the algorithm introduced may be directly applied to a more general class of dynamic load-balancing problem than that which stems from (1). In particular, the adaptivity may be based upon a 3-d hierarchical mesh of tetrahedra (see [10] for example) and/or an adaptive p-version of the finite element algorithm (as in [1] for example).

## 2 A Dynamic Load-Balancing Algorithm

Suppose we have a hierarchically refined root mesh which is distributed across  $p$  processors. The weight of a root element will be defined as the number of leaf elements inside it and the weight of an edge of the root mesh as the number of leaf element edges along it. We may now define the weighted dual graph of the root mesh: where each node of this graph corresponds to an element of the root mesh,  $\mathcal{T}_0$ , and two nodes are connected if the corresponding root elements are neighbours. The task of repartitioning the triangulation  $\mathcal{T}$  may therefore be represented in terms of this weighted graph. In particular, we require an algorithm for repartitioning such a graph which satisfies the four criteria enumerated in section 1.

### 2.1 Group balancing

Following Vidwans *et al* [12], we define a further weighted graph: the weighted partition communication graph (WPCG). This represents the face adjacency of the  $p$  processors being used (processors that share at least one edge of a root element with a given processor are said to be face adjacent to that processor). A WPCG is obtained by having one vertex for every processor and an edge between two vertices if and only if they are face adjacent to each other. The weight  $w_{N_i}$  of the  $i^{th}$  vertex is equal to the sum of weights of all root 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 root element edges on the interpartition boundary between the two processors.

We now divide the WPCG into two subgroups denoted by Group1 and Group2. Unlike in [12] however we use a weighted version of the spectral bisection method [5] which results in a partition which is based upon having an approximately equal weight in each group, rather than an equal number of processors. Moreover the spectral algorithm is also designed to keep the weight of those edges of the WPCG which are cut by the partition (the “cut-weight”) as low as possible. The cost of implementing this algorithm is not significant since the number of processors,  $p$ , is always small compared with the size of the root mesh.

A full description of the weighted spectral bisection algorithm may be found in [5]. Briefly, a weighted Laplacian matrix,  $L$ , for the WPCG is first formed and then scaled by the diagonal matrix  $D = \text{diag} \left( \frac{1}{\sqrt{w_{N_i}}} \right)$ . The second eigenvector (or Fiedler vector),  $\underline{u}_2$ , of the scaled matrix  $S = D^T L D$  is then found. Finally a partitioning vector,  $\underline{x}$ , is defined by  $x_i = u_{2_i} / w_{N_i}$  for  $i = 1, \dots, p$ . The two sub groups are then defined by sorting the  $p$  vertices of the WPCG according to the size of their entry in  $\underline{x}$  and placing elements represented by  $x'_i: i = 1 \dots n$  in one group (with  $\underline{x}'$  being the sorted vector) and those by  $x'_i: i = n + 1, \dots, p$  in the other, with  $n$  chosen so that

$$(2) \quad \left| \sum_{i=1}^n w'_{N_i} - \sum_{i=n+1}^p w'_{N_i} \right|$$

is as small as possible (where  $w'_{N_i}$  is the weight of the vertex represented by  $x'_i$ ).

If the leaf mesh is quite uniformly distributed across the processors then we would expect each group to contain about the same number of processors and an almost identical total weight. If the existing partition is not well load-balanced however then the number of processors in each group may be very different. In either case the cut-weight resulting from this bisection will generally be very small. In the next stage of the algorithm we use the idea of local migration from the “larger” to the “smaller” group so that after migration each group contains approximately the same average weight per processor without there being a significant increase in this cut-weight.

## 2.2 Local migration

As mentioned above the subgroups formed in the last subsection may not be ideally balanced. To balance them we now migrate nodes of the weighted dual graph from the “larger” to the “smaller” group. There are many ways to do this but, due to the non-linear complexity of the Kernighan and Lin algorithm ([8]), we apply the ideas of Fiduccia and Mattheyses ([4]) who suggest a similar algorithm but whose complexity is linear.

We first decide which of the subgroups is to be the Sender and which the Receiver. We then define the number  $\text{Mig}_{tot}$  to equal the total weight of all the nodes which are to be migrated from the Sender to the Receiver. Let  $N_1$  and  $N_2$  be the number of processors in Group1 and Group2 respectively. Also let  $Ave$  be the average weight per processor in the WPCG and  $Ave_1$  and  $Ave_2$  be the average weights per processor in Group1 and Group2 respectively. Then the calculation of Sender, Receiver and  $\text{Mig}_{tot}$  is shown in figure 2 below (in order to calculate  $\text{Mig}_{tot}$  one simply multiplies the average excess load per processor by the number of processors in the Sender group). Note that if the combined weight of the nodes transferred between the Sender and the Receiver is nearly or exactly equal to  $\text{Mig}_{tot}$  then the two groups will be load-balanced upon completion.

Having established the required load to be transferred, the next issue to address is that of how many nodes each processor in the Sender group should actually send and

```

if( $Ave_1 \leq Ave_2$ ){
    Sender = Group2;
    Receiver = Group1;
    Migtot =  $N_2 * (Ave_2 - Ave)$ ;
}
else{
    Sender = Group1;
    Receiver = Group2;
    Migtot =  $N_1 * (Ave_1 - Ave)$ ;
}

```

FIG. 2. Calculation of Sender, Receiver and Mig<sub>tot</sub>.

which processors in the receiver group they should be sent to. Again we build upon the algorithm of Vidwans *et al* [12], by defining the concept of candidate processors. Processors in each group that are face-adjacent to at least one processor in the other group are called candidate processors. We only allow the candidate processors to be involved in the actual migration of nodes from Sender to Receiver. Let  $N_{tot}$  be the total weight on all candidate processors of the Sender group. Then if the  $i^{th}$  candidate processor in the Sender group is face adjacent to more than one candidate processor in the Receiver group we migrate nodes to that candidate processor which has the “longest” boundary (by this we mean that the cut-weight between the two processors involved is maximum as compared to other possible pairs). The amount of load shifted from the  $i^{th}$  candidate processor in Sender group is denoted by  $Mig_i$  and is given by,

$$(3) \quad Mig_i = \left( \frac{N_i}{N_{tot}} \right) * Mig_{tot},$$

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

Finally, it is necessary to decide precisely which nodes in the weighted dual graph of the root mesh should be transferred. Our aim is to transfer those nodes which result in as low a cut-weight as possible. The fundamental ideas behind this are the concepts of the “gain” and “gain density” associated with moving a node onto a different processor. For a node,  $k$  say, which is situated on the  $i^{th}$  candidate processor in the Sender group, we define the gain( $k$ ) associated with this node to be the net reduction in the cut-weight that would result if this node were to migrate to the Receiver group (the  $j^{th}$  candidate processor in the Receiver group say). The calculation of gain( $k$ ) is shown in figure 3.

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

FIG. 3. The calculation of gain.

The gain density of a node is defined as the gain of the node divided by the weight of the node. The bulk of the work needed to make a move consists of selecting the base node (a node which is about to be shifted from one processor to another processor is called a base node), moving it, and then updating the gains of its neighbouring nodes. We solve the first

problem, that of selecting a base node, by choosing the node with the largest gain density on the  $i^{th}$  processor and whose weight is less than or equal to  $Mig_i$ . We shift the node to the receiving processor and update the gains of its neighbouring nodes (observe that in general the node  $k$  will have three neighbours when we are working with triangulations of two-dimensional domains and four neighbours when we are working with triangulations of three-dimensional domains) using the algorithm outlined in figure 4. Observe that, if the gain associated with the base node is positive, then transferring it will not only improve the load-balance but will also reduce the total cut-weight between the two groups.

```

For each  $n_k$  which is a neighbour of the node  $k$  {
  Let  $p_k$  be the processor to which  $n_k$  belongs;
  if ( $p_k == j$ ) then
    decrement  $gain(n_k)$  by  $2 * w_{E_{n_k k}}$ ;
  else if ( $p_k == i$ ) then
    increment  $gain(n_k)$  by  $2 * w_{E_{n_k k}}$ ;
}

```

FIG. 4. *Updating the gains.*

### 2.3 Divide and conquer and parallel implementation

Once we have obtained Sender and Receiver groups with the same average weights, it is possible to recursively apply the above splitting algorithm to each of these two processor groups in parallel: bisecting them and load-balancing them. The recursion terminates when every group consists of a single processor: each with approximately the same load.

This divide and conquer approach naturally permits a certain degree of parallelism in its implementation. Further parallelism is also facilitated by the fact that it is possible for more than one sending processor in a Sender group to migrate data onto its corresponding receiving processor at any given time. To ensure that no data conflicts arise as a result of this parallel communication a simple colouring algorithm is used to prevent two neighbouring coarse elements from being transferred simultaneously. A more detailed description of this algorithm may be found in [11].

The implementation of this load-balancing algorithm that is used for the numerical experiments described in the next section was completed using the MPI library ([9]). This is ideally suited to the divide and conquer philosophy since it provides explicit mechanisms for the definition and splitting of processor groups.

## 3 Some Examples

In this section we describe some computations in which a parallel implementation of our dynamic load-balancing algorithm is tested and contrasted with implementations of two alternative methods: those of Vidwans *et al* [12] and Hu and Blake [6]. We begin by briefly describing the two methods used for the comparison and then present some computational results for which some non-uniformly refined meshes are dynamically load-balanced on between 16 and 64 processors of the Cray T3d.

### 3.1 Alternative algorithms

In [12] Vidwans *et al* describe a dynamic load-balancing algorithm which is quite similar to our algorithm, described in section 2. The main differences are that when they bisect a

processor group into Sender and Receiver subgroups this is based purely upon the processor numbers. In addition, where possible, the processors are equally split amongst the two subgroups regardless of the quality of the existing partition. The final substantial difference is that the notion of gains is not utilized in the local migration phase of their algorithm.

In our implementation of this algorithm we have attempted to be as fair as possible. In particular we use a local migration strategy which is the same as that described in 2.2, which numerical experiment suggests to be superior to alternatives that we attempted. In addition we number the processors in a way that we consider to be as good as possible, based upon an initial calculation of the Fiedler vector of the original WPCG. (Clearly the numbering of the processors has a very significant effect on the performance of the algorithm described in [12] since the group bisection depends solely upon this.)

The other algorithm that we use for the purposes of comparison is that described by Hu and Blake in [6]. This is not based upon recursive splitting but instead seeks to calculate the optimal route from an existing partition to one in which the weighted dual graph of the root mesh is perfectly load-balanced. For the purposes of this algorithm the term “optimal” is used to mean that the Euclidean norm of the migrated load is minimized. It should be noted that the algorithm doesn’t take into account that the load on each processor comes in discrete units (i.e. the weights of the root elements) and so it may require that the weight to be transferred between two processors is a value that is not actually achievable in practice. Clearly transferring an approximation to this load is the best that can be practically achieved (and on occasions where the root mesh is very heavily locally refined this may be far from ideal).

As with the other algorithm used for comparison we have again tried to be as fair as possible in our parallel implementation of Hu and Blake’s method. In particular we take great care to ensure that the particular root elements that are transferred between processors are chosen with the overall cut-weight in mind: again making use of the notion of the “gain” and “gain-density” of each node on a sending processor.

### 3.2 Comparative results

We are now in a position to compare our new dynamic load balancing algorithm with the two algorithms described above. In this subsection we consider three test problems, each relating to a different geometry  $\Omega$  and a different root mesh. The common feature of each example is that these root meshes are all subject to extremely non-uniform local refinement which leads to a large proportion of the leaf elements being contained on a relatively small number of root elements (this is typical in an adaptive finite element or finite volume solver). In addition, the initial partition of the leaf mesh is not very well load-balanced but it does have a short interprocessor partition boundary (i.e. a small cut-weight).

- In problem I the root mesh contains 3008 elements and the leaf mesh 983509 elements which are split across 16 processors. The geometry used is the “Texas” domain taken from PLTMG [2] and the initial partition has between 8 and 1382 root elements in each subdomain.
- For problem II the root mesh contains 3126 elements and the leaf mesh 1314666 elements which are split across 32 processors. The geometry is taken from [5] (geometry 2) and features a complex hole in the interior of the region. The initial partition has between 3 and 1013 root elements in each subdomain.

- In case III the root mesh contains 6313 elements but the leaf mesh contains just 118077 elements which are now split across 64 processors. The geometry is the region around a NACA0012 aerofoil and the initial partition has between 14 and 1116 root elements in each subdomain.

The results of applying the three algorithms to each of these test problems are summarized in table 1. This clearly shows that the new algorithm consistently achieves a better load-balance than the original algorithm of Vidwans *et al* [12]. In addition, when compared to the optimal approach of Hu and Blake [6], a smaller cut-weight is generally achieved. (Note that the maximum imbalance, Max-imb, is the largest percentage by which the total weight on any single processor exceeds the average weight per processor. Also, the cut-weight is calculated as the total weight of all of those edges of the weighted dual graph of the root mesh which are cut by the partition boundary.)

Example	Feature	Initial	Hu and Blake	Vidwans <i>et al</i>	New
I	Max-imb	8.0%	5.9%	4.5%	4.5%
	Cut-wt	8031	8846	8936	8724
II	Max-imb	14.2%	14.2%	14.2%	10.0%
	Cut-wt	10913	12778	12453	12485
III	Max-imb	12.6%	3.0%	10.1%	3.5%
	Cut-wt	5863	6680	6108	6324

TABLE 1

*Comparison of dynamic load-balancing results using three algorithms.*

There are a number of comments which need to be made concerning these results. Firstly, the results presented for the algorithm of Vidwans *et al* are in fact rather better in terms of the cut-weight produced than if we had followed the original paper ([12]) exactly. This is because we take great care to transfer those root elements with the highest gain densities at each stage. The relatively poor load-balance achieved by this method is not affected however. Also, it may appear on first inspection that all three of these techniques perform quite poorly in terms of the size of the maximum imbalance. This is not really the case however since the mesh refinement in each example is highly localized (as is typical in the adaptive solution of partial differential equations) and so some root elements have extremely large weights compared with others. This makes it impossible to achieve an exact load-balance in these cases without increasing the cut-weight massively. In [11] some further examples are given in which more uniformly refined root meshes are repartitioned. In these examples each algorithm consistently achieves a maximum imbalance of well under 1%. It is also worth noting that in such cases the algorithm of Hu and Blake [6] is generally superior to our algorithm in that the load balance is slightly better and the cut-weight is usually very similar.

A final note for this section is to observe that the parallel execution times for all of these algorithms are generally quite similar. Not surprisingly the algorithm of Hu and Blake is usually a little faster to execute due to fact that it transfers the least amount of data. Nevertheless, the other algorithms appear to scale in a very similar manner and so never finish that far behind (for example, in case III the optimal algorithm takes 2.8 seconds to execute on the T3d whilst the group algorithms both take about 3.9 seconds).

## 4 Conclusions

In this paper we have derived a new algorithm for solving the dynamic load-balancing problem that arises during the parallel adaptive solution of time-dependent PDEs using hierarchical meshes. We base our decomposition of the problem upon a partition of the root mesh used in the adaptivity and seek to ensure that each processor holds an equal number of leaf elements. In addition to this constraint we also seek to minimize the number of elements with edges on the boundary between processors and to minimize the amount of data migration that is required. Finally we have implemented our algorithm in parallel and contrasted its performance with that obtained from parallel implementations of two alternative algorithms.

The results presented suggest that our algorithm strikes a good balance between the often conflicting demands of achieving the best load-balance whilst maintaining a low cut-weight. This is particularly true in the highly demanding (although very realistic) situation where the local mesh refinement is extremely non-uniform. In these cases some root elements have a weight which is very much greater than that of others and this causes the performance of the dynamic load-balancing algorithms to be severely tested. The new algorithm appears to achieve the best results in this situation.

## Acknowledgements

Our parallel computations were carried out on the Cray T3d computer at the Edinburgh Parallel Computing Centre. NT would like to acknowledge the financial support of the UK and Pakistan governments in the form of ORS and COTS scholarships respectively.

## References

- [1] I. Babuska and I.N. Katz and B. Szabo (1981). *The P Version of the Finite Element Method*, SIAM J. Numer. Anal., 18, pp. 515-545.
- [2] R.E. Bank (1994). *PLTMG Users' Guide 7.0*, SIAM, Philadelphia.
- [3] M. Berzins and J. Ware (1995). *Positive Cell Centred Finite Volume Discretization Methods for Hyperbolic Equations on Irregular Meshes*, Appl. Num. Math., 16, pp. 417-438.
- [4] C.M. Fiduccia and R.M. Mattheyses (1982). *A Linear Time Heuristic for Improving Network Partitions*, Proc. of the 19th IEEE Design Automation Conference, IEEE, pp 175-181.
- [5] D.C. Hodgson and P.K. Jimack (1996). *Efficient Parallel Generation of Partitioned, Unstructured Meshes*, Advances in Engineering Software, 27, pp. 59-70.
- [6] Y.F. Hu and R.J. Blake (1995). *An Optimal Dynamic Load Balancing Algorithm*, Preprint DL-P-95-011 of The Central Laboratory for the Research Councils, Daresbury Laboratory, Daresbury, Warrington, Cheshire WA4 4AD, UK.
- [7] P.K. Jimack (1993). *A New Approach to Finite Element Error Control for Time-Dependent Problems*, In Numerical Methods for Fluid Dynamics 4, ed. M.J. Baines and K.W. Morton (Clarendon Press, Oxford), pp. 567-573.
- [8] B. Kernighan and S. Lin (1970). *An efficient heuristic procedure for partitioning graphs*, Bell System Technical Journal, 29, pp 209-307.
- [9] Message passing Interface Forum (1994). *MPI: A Message Passing Interface Standard*, Int. J. of Supercomputer Applications, 8, no. 3/4.
- [10] W.E. Speares and M. Berzins (1995). *A Fast 3-D Unstructured Mesh Adaption Algorithm ith Time-Dependent Upwind Euler Shock Diffraction Calculations*, In Proc. of 6th Int. Symp. on Computational Fluid Dynamics, ed. M. Hafez and K. Oshima, vol. III, pp. 1181-1188.
- [11] N. Touheed and P.K. Jimack (1996). *Dynamic Load-Balancing for Adaptive PDE Solvers*, School of Computer Studies Research Report 96.34, University of Leeds, Leeds LS2 9JT, UK.
- [12] A. Vidwans, Y. Kallinderis and V. Venkatakrishnan (1994). *Parallel Dynamic Load-Balancing Algorithm for 3-Dimensional Adaptive Unstructured Grids*, AIAA Journal, 32, pp. 497-505.