

An Overview of Parallel Dynamic Load-Balancing for Parallel Adaptive Computational Mechanics Codes

P.K. Jimack
Computational PDE Unit
School of Computer Studies
University of Leeds
Leeds LS2 9JT, UK

Abstract: This paper begins with a brief review of how dynamic load-balancing problems arise in the context of parallel adaptive computational mechanics codes. Particular emphasis is given to the h-version of the finite element method applied to time-dependent problems and the distinctive features of dynamic load-balancing, as opposed to static load-balancing, are emphasized. These include the issues of data locality and the need for parallel implementations. Following this some popular classes of dynamic load-balancing algorithm are discussed, beginning with a review of how some well-known static load-balancing algorithms might be applied to dynamic problems and then moving on to look at various diffusion algorithms. In addition a variety of other techniques are discussed, including those designed to minimize the amount of data relocation required, multilevel methods and recursive algorithms.

1 Introduction

This paper is concerned with the issue of dynamic load-balancing which arises during the parallel, adaptive solution of various computational mechanics problems. For example, when time-dependent problems are solved using the h-version of the finite element method (see [24, 36, 40] for example) it is frequently the case that parts of the spatial domain which are heavily refined at one time will be significantly less refined at some later time (and vice versa). Given that an efficient parallel algorithm requires approximately the same amount of work to be performed by each processor, this adaptivity therefore adds significant complexity to the problem of partitioning the computational load.

In the next section of the paper a typical adaptive finite element algorithm is briefly described for the solution of a straightforward 2-d model problem. This is then used to motivate the need for a dynamic load-balancing procedure within a parallel solver, and the requirements for such a procedure are specified. It should be emphasized that both this particular choice of adaptive solution algorithm

and choice of test problem are selected for illustration purposes only. In fact the application of dynamic load-balancing is necessary for far more general adaptive methods (such as those using p- or hp-refinement [6, 7, 8]) and far more complex time-dependent problems, including those in three dimensions.

Section 3 then goes on to review a number of possible heuristics for dealing with the dynamic load-balancing problem that has been defined. Whilst this review is by no means exhaustive, it does aim to introduce and contrast a broad cross-section of the published work in this area. The paper concludes with a short discussion on the practical use of some of the methods discussed within reliable and robust computational mechanics software.

2 Background

In this section we attempt to demonstrate how the dynamic load-balancing problem typically arises in parallel computational mechanics codes by studying a particular adaptive solution algorithm for a linear test problem. This problem is introduced in the first subsection and in the second subsection the adaptive algorithm is described, along with its implications.

2.1 Parallel implementation of the finite element method for a time-dependent test problem

Consider the linear diffusion equation with a time-dependent source term on a spatial domain $\Omega \subset \mathfrak{R}^2$:

$$\frac{\partial}{\partial t}u(\underline{x}, t) = \nabla^2 u(\underline{x}, t) + f(\underline{x}, t) \quad \text{for } (\underline{x}, t) \in \Omega \times (0, T], \quad (1)$$

subject to the initial condition

$$u(\underline{x}, 0) = u^0(\underline{x}) \quad \text{for all } \underline{x} \in \Omega,$$

and the boundary conditions

$$u = u_E \quad \text{on } \Gamma_1 \quad \text{and} \quad \frac{\partial u}{\partial n} = g \quad \text{on } \Gamma_2 \quad \text{for all } t \in (0, T],$$

where $\partial\Omega = \Gamma_1 \cup \Gamma_2$ and $\Gamma_1 \cap \Gamma_2 = \{\}$. We may seek a piecewise linear finite element approximation, u^h say, to u by creating a triangulation, \mathcal{T}^h , of Ω and setting

$$u^h(\underline{x}, t) = \sum_{i=1}^{N+M} u_i(t)N_i(\underline{x}), \quad (2)$$

where $N_i(\underline{x})$ are the usual linear hat basis functions on \mathcal{T}^h and M and N are the number of vertices of \mathcal{T}^h which do and do not lie on Γ_1 respectively. The finite element equations which result from this approximation take the form

$$M \frac{d\underline{u}}{dt} = -K\underline{u} + \underline{F}(t), \quad (3)$$

where $\underline{u} : t \rightarrow \mathfrak{R}^N$ contains the solution estimate at each vertex of \mathcal{T}^h not lying on Γ_1 . Here M is an $N \times N$ mass matrix, K is an $N \times N$ stiffness matrix and the vector $\underline{F} \in \mathfrak{R}^N$ incorporates both the source term and the boundary conditions. (See [26], for example, for a more complete description than is possible here.)

Equations (3) represent a finite element semi-discretization of the original problem (1). Suppose we use the theta method to solve this initial value problem (see [34] for example). Then at each time step it is necessary to solve the linear system

$$(M + k\theta K)\underline{u}^{n+1} = (M - k(1 - \theta)K)\underline{u}^n + (1 - \theta)\underline{F}(t^n) + \theta\underline{F}(t^{n+1}), \quad (4)$$

where $\underline{u}^{n+1} \approx \underline{u}(t^{n+1})$, $\underline{u}^n \approx \underline{u}(t^n)$ and $k = t^{n+1} - t^n$.

There are numerous ways in which (4) may be solved on a parallel distributed memory computer, however we follow the domain decomposition philosophy here (see [25] for an overview), since this also permits the equations to be assembled and stored in a parallel and distributed manner. The first step therefore is to decompose the triangulation \mathcal{T}^h of Ω into non-overlapping subdomains (see figure 1) and then order the unknown components of $\underline{u}(t)$ so that the internal nodes for each subdomain are enumerated first, one subdomain at a time, followed by the remaining unknowns which lie on the partition boundary. With this ordering, each of the matrices M and K have the following block arrowhead structure:

$$A = \begin{bmatrix} A_{II(1)} & & & & A_{IB(1)} \\ & A_{II(2)} & & & A_{IB(2)} \\ & & \ddots & & \vdots \\ & & & A_{II(P)} & A_{IB(P)} \\ A_{IB(1)}^T & A_{IB(2)}^T & \cdots & A_{IB(P)}^T & A_{BB} \end{bmatrix}. \quad (5)$$

Here it is assumed that there are P subdomains and that A represents either the mass or the stiffness matrix. If each subdomain is stored on a single processor, numbered from 1 to P , then it is possible to compute and store each of the blocks $A_{II(i)}$ and $A_{IB(i)}$ independently on processor i . Moreover, each processor can independently compute and store its own contribution to A_{BB} , $A_{BB(i)}$ say, where $A_{BB} = A_{BB(1)} + \dots + A_{BB(P)}$.

If an iterative method, such as the conjugate gradient algorithm (see [13] for example), is used to solve (4) at each time step then, in the absence of a preconditioner, the major computational step within each iteration is to perform a matrix-vector multiply: $\underline{q} = (M + k\theta K)\underline{p}$ say. In the block notation of (5) this is

$$\underline{q}_{I(i)} = (M_{II(i)} + k\theta K_{II(i)})\underline{p}_{I(i)} + (M_{IB(i)} + k\theta K_{IB(i)})\underline{p}_B \quad (6)$$

for $i = 1, \dots, P$ (which may all be computed in parallel), and

$$\underline{q}_B = \sum_{i=1}^P (M_{IB(i)} + k\theta K_{IB(i)})^T \underline{p}_{I(i)} + (M_{BB(i)} + k\theta K_{BB(i)})\underline{p}_B \quad (7)$$

(which may also be computed in parallel; with the additional requirement of a global summation). The important point to observe from (6) and (7) is that, for

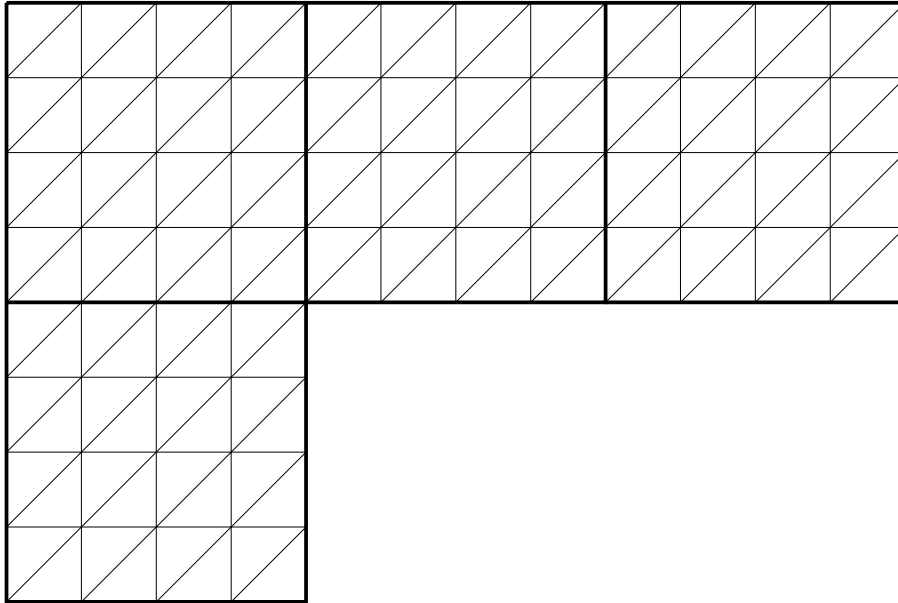


Figure 1: A possible decomposition of a finite element mesh.

the work to be split equally amongst the P processors (thus achieving maximum parallel efficiency) it is necessary that the number of unknowns in each subdomain be as equal as possible. It is also desirable that the number of nodes on the boundary of each subdomain is as small as possible so as to minimize the work in calculating \underline{q}_B . Hence, the algorithm used to partition the mesh \mathcal{T}^h into P subdomains should have two objectives:

1. the number of unknowns in each subdomain should be about the same,
2. the number of unknowns on the partition boundary should be as small as possible.

Note that even if we use a more sophisticated parallel solution algorithm, such as applying an element-by-element or a domain decomposition preconditioner (see [23] or [25] respectively), these two requirements will still hold so as to ensure that the computational load is well-balanced and the additional communication and computation overheads are as low as possible.

When solving (1) on a single mesh, \mathcal{T}^h , for all $t \in (0, T]$ it is only necessary to decompose \mathcal{T}^h into subregions once, at the start of the computation. This decomposition may be expressed as a well-known graph partitioning problem in which it is desired to divide a graph into P equally sized subgraphs in such a way as to minimize the number of edges of the graph which pass between subgraphs. To see this, define a unique vertex of the graph for each element of \mathcal{T}^h and a unique edge of the graph to join those vertices which represent elements of \mathcal{T}^h which share a face. A solution of the graph partitioning problem will then satisfy both requirements 1 and 2 stated above. Unfortunately, finding an exact solution for this problem is NP-hard. Nevertheless, for our purposes, any “good” approximate solution will be adequate and a large number of heuristics have been developed and analyzed over the years. These heuristics include

greedy algorithms ([10]), geometric algorithms based upon the coordinates of the centroids of the elements in \mathcal{T}^h (e.g. recursive coordinate bisection [43, 52] or recursive inertial bisection [9, 39]), recursive graph bisection algorithms (such as described in [17, 43] or the spectral bisection algorithm used in [14, 43, 57]), multilevel and graph-coarsening algorithms (see [1, 2, 15, 29]), and global minimization methods (such as using neural networks [3, 47], genetic algorithms [47] or simulated annealing [57]).

Numerous public domain software packages have been developed which make use of many of the techniques referenced above (see, for example, [16, 29, 41, 56]) as well as various hybrid strategies. These hybrids tend to consist of applying a standard heuristic in order to get an initial partition of the mesh and then applying a different, local, heuristic (such as [11, 31, 38]) in an attempt to improve the quality of this partition. Finally in this subsection, we note that an alternative to partitioning an existing mesh \mathcal{T}^h is to generate a partitioned mesh in parallel (which should still satisfy the two quality criteria of load balance and short interpartition boundary). A number of algorithms have been proposed for this, including [12, 18, 32, 37, 50] for example.

2.2 The use of adaptivity and the need for dynamic load-balancing

In practice, for many time-dependent problems (including (1) for certain choices of $f(\underline{x}, t)$ or $u^0(\underline{x})$), it is both inefficient and inaccurate to use a single finite element mesh, \mathcal{T}^h , for all $t \in (0, T]$. This is because to represent the solution $u(\underline{x}, t)$ accurately a very large number of degrees of freedom may be required in certain regions of Ω at some times, and many fewer may be required in the same region at other times. For example, the solution to (1) may start out by being extremely smooth and slowly varying throughout Ω but, as it evolves with time, a steep boundary layer might develop near some parts of $\partial\Omega$. To represent such a boundary layer accurately a very fine mesh is required locally but it would be extremely inefficient, and impractical, to have such a fine mesh throughout the whole of Ω . Neither is it possible to produce *a priori* a mesh \mathcal{T}^h which is fine only in the vicinity of the boundary layer since, in general, the location of any such regions will not be known until the problem is being solved.

For these reasons it is common to use adaptive finite element schemes in which either the size of the elements [24, 36, 40] and/or the degree of the approximation [6, 7, 8] are allowed to vary throughout the spatial domain with time. For the purposes of illustration, in this paper we only consider variations in the local element size: known as h-refinement. We follow [24] and suppose that after each time step (i.e. solution of (4)) an *a posteriori* error estimate is used to indicate in which regions of the domain the error is too large, and in which regions it is unnecessarily small (see [20, 22, 27, 46] for example). Every triangular element which has an unacceptably large error on it is then subdivided into four child elements, and any groups of four sibling elements with an unnecessarily small error are considered for derefinement (see [36]). Figure 2 illustrates this in the case of local mesh refinement only. Note that the initial mesh can never be

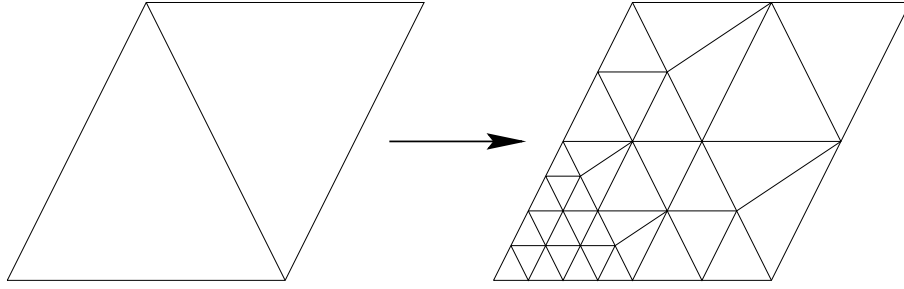


Figure 2: An example of local hierarchical refinement in 2-d.

coarsened, but once an element has been refined its children may be derefined at some later time. If the initial condition, $u^0(\underline{x})$, is not smooth then it is possible to refine the initial mesh at time $t = 0$ so as to allow a good representation of $u^0(\underline{x})$: this initial refinement is not necessarily permanent however. (Note that there are also a number of 3-d generalizations of this algorithm which work in much the same hierarchical fashion [28, 42, 45].)

Now suppose that the above adaptive algorithm is implemented in parallel. After the initial mesh has been refined so as to allow an accurate representation of $u^0(\underline{x})$, it must be partitioned across the available processors. This will be done, as outlined in subsection 2.1, to ensure that each processor has about the same number of elements (or nodes), and that the partition boundary is short. As the solution evolves however, the *a posteriori* error estimates may cause the mesh to be locally refined on some processors and/or locally derefined on others. After some time this is likely to cause the computational load on each processor to become unbalanced, and hence the parallel efficiency of the solver will be substantially reduced.

Once a significant imbalance in the load on each processor has developed it becomes necessary to modify the partition of the mesh so as to rectify the situation. In theory it is possible to apply any of the graph partitioning strategies cited in the previous subsection and then to redistribute the mesh accordingly. Unfortunately, this has at least two undesirable consequences. Since none of these algorithms make any use of the existing partition it is almost certain that the vast majority of elements in the latest mesh will need to be relocated onto a different processor: a very significant communication overhead (in [55] for example, typical calculations lead to about 95% of elements being relocated on 32 processors and an even higher percentage on 64). In addition, nearly all of the algorithms referenced above are inherently sequential, and would therefore cause a parallel bottleneck (i.e. $P-1$ idle processors) whenever repartitioning takes place.

A far more versatile mesh partitioning algorithm is therefore required when a parallel adaptive solver is being used. Such an algorithm needs to be both dynamic, in that it modifies an existing partition incrementally, and parallel. To summarize therefore, we wish to develop a partitioning algorithm with the following four properties:

1. it should produce a load-balanced partition,

2. there should be a minimal amount of data migration,
3. a minimal number of elements should end up on the partition boundary,
4. there should be an efficient parallel implementation.

3 Dynamic Load-Balancing Algorithms

There are two distinct possibilities to choose between when partitioning an adapted finite element mesh. One can either partition the actual elements in the latest mesh or one can just partition the initial mesh (with any child elements being located accordingly). The first option is clearly more versatile (see [8] for example) however it does mean that some elements may be stored on a different processor to their parent, which causes significant additional complexity for the parallel adaptivity algorithm. We therefore concentrate on the second case here, in which the initial mesh is dynamically partitioned. This does not mean that the algorithms discussed below do not apply when the actual mesh is used: we simply choose to work with the initial mesh for clarity.

As with the static problem it is convenient to express the task of producing a new decomposition of an adapted mesh in terms of a graph partitioning problem. For each element, i , of the initial mesh define a vertex of a dual graph and let this vertex have weight v_i , where v_i is the number of elements of the actual mesh currently contained inside i . For each pair of face adjacent elements in the initial 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 elements in the actual mesh which currently meet along edge j . For example, in figure 2 we see an initial mesh with just two elements and one common face. Hence the dual graph has two vertices (with weights 30 and 6 respectively) and one edge (with weight 4). At the end of each call to the adaptivity procedure the weights of the dual graph must be updated and it is necessary to assess whether the current partition has become unbalanced. If it has then a dynamic load-balancing technique should be applied so as to improve the quality of the partition. We now consider a number of possibilities for doing this.

3.1 Generalizations of static algorithms

In this subsection we show how some of the static partitioning algorithms referenced in subsection 2.1 can be applied to the dynamic problem: sometimes after they have been modified somewhat.

3.1.1 geometric algorithms

Recursive coordinate bisection

This approach is described in [43], where it is presented as a slight generalization of an earlier method used by Williams in [57]. For a 2-d domain the method is based upon recursively bisecting the mesh by cutting it either in a horizontal or a vertical direction. This cut is made across whichever is the shorter out of the

height and breadth of the domain/subdomain, and is positioned so as to split the weight as equally as possible.

Although this technique is very straightforward and computationally inexpensive it is not particularly popular in practice. This is because the number of elements on the partition boundary tends to be quite high for meshes of irregular density. Nevertheless it is a relatively good method from a dynamic load-balancing point of view since it naturally tends to preserve the locality of a very high proportion of the elements. In addition it may be implemented in parallel since, at each level of the recursion, more and more subdomains may be bisected simultaneously.

Recursive inertial bisection

This is a more general technique than recursive coordinate bisection that is described in [9, 39] for example. Here, the vertices of the dual graph are considered as point masses (given by their weights v_i) located at the centroid of their corresponding initial element. The principal axis of inertia for these point masses is then calculated and the domain is bisected by making a cut which is orthogonal to this axis (with approximately equal weights on either side of it). This procedure is then repeated recursively for each subdomain.

In [9] it is reported that this method is quite fast but that the number of elements next to the partition boundary is often still quite high (an issue which is addressed by applying the local improvement algorithm of [11] after each bisection). As with other geometric approaches data locality is generally preserved quite well when using recursive inertial bisection since the principal axes will only change gradually as the mesh steadily refines and derefines. The method is also amenable to parallel implementation both through its recursive nature and through an ability to calculate the principal axis of inertia for a given subdomain in parallel.

3.1.2 Spectral algorithms

Another algorithm frequently used in static graph partitioning is recursive spectral bisection (see [43, 57], with a version for weighted graphs described in [14]). If we wish to partition our weighted dual graph into two subdomains of equal size then, for each vertex i , we wish to assign a value of ± 1 to x_i such that

$$\sum_i x_i v_i = 0 . \tag{8}$$

All vertices for which $x_i = +1$ will then be in one subgraph whilst the remainder, for which $x_i = -1$, will be in the other. If we also wish to ensure that the total weight of edges of the dual graph that cross the partition boundary is as low as possible then it is also necessary to choose the values of x_i such that we

$$\text{minimize } \frac{1}{4} \sum_j e_j (x_{j(1)} - x_{j(2)})^2 , \tag{9}$$

where $j(1)$ and $j(2)$ are the numbers of the two vertices at the ends of edge j . When using spectral bisection the key is to treat this constrained minimization problem (i.e. (9) subject to the constraint (8)) as a continuous rather than a discrete problem. This can be solved by evaluating a single eigenvector of the Laplacian matrix of the weighted dual graph: one then assigns discrete values of ± 1 to the x_i according to the corresponding entry in this eigenvector – hence the name “spectral bisection”.

It turns out that when this spectral bisection algorithm is applied recursively it tends to give extremely good static partitions of most graphs (i.e. the number of edges cut by the subdomain boundary is very low). Unfortunately, in the form described here, it is of little use in dynamic load balancing since there is no mechanism for ensuring that an original partition of the dual graph is in any way similar to the recursive spectral partition. Hence it will usually be the case that a large amount of data migration is required whenever this algorithm is used to repartition the initial mesh.

In [51] Van Driessche and Roose seek to overcome this difficulty by adding some additional vertices and edges to the weighted dual graph. A new vertex is added for each of the existing subgraphs and an edge is added to each of the original vertices: joining it with the new vertex which corresponds to the subdomain to which it currently belongs. Note that none of these new edges are cut by the current partition of this extended graph. If we can repartition this graph so that there is still precisely one new vertex per subdomain after repartitioning then it follows that any new edges which are now cut by the partition must correspond to the migration of a vertex of the dual graph from one subdomain to another. Given that the spectral algorithm performs well in terms of keeping the number of cut edges quite low it is to be expected that, when applied to this extended graph, it will lead to a small amount of data migration. It turns out that the success of this approach depends quite heavily on the choice of weight that is used for the new edges in the extended dual graph and also on the ability to ensure that the new partition does indeed have just one new vertex per subdomain. Nevertheless, the results obtained in [51] are extremely encouraging.

A rather different modification to the spectral algorithm, also designed to allow it to be applied dynamically, is proposed by Walshaw and Berzins in [54]. In this algorithm the dual graph is coarsened by merging into a single “super vertex” all of the vertices in each subdomain that are more than d edges away from the subdomain boundary. The weight of each of these super vertices is equal to the sum of the weights of its components so they are usually very heavy. Hence, when the weighted spectral bisection is applied to this coarsened graph it is generally the case that there is exactly one of these super vertices in each of the new subdomains. This in turn ensures that very little data migration is required to move from the old partition to the new one. In addition the low edge cut property of the spectral approach ensures that the new partition boundary is still reasonably short.

From the viewpoint of parallel implementation, both of these variants have the advantage of being recursive, which automatically permits a certain degree

of concurrency. There is also the possibility of using a parallel implementation of the spectral algorithm at each stage in the recursion (see [2, 35] for example), however this does not appear to have been attempted as yet.

3.1.3 A multilevel algorithm

The use of multilevel algorithms for the static graph partitioning problem has been shown to be highly effective in a number of recent publications (see [1, 2, 15, 29] for example). The general idea behind these techniques is to produce a hierarchy of coarsenings of the original weighted graph (where each level in the hierarchy is produced by merging together groups of neighbouring vertices of the graph at the previous level), and then to perform a *global* partition only for the coarsest graph. This partition is then projected onto the graph at the previous level and then modified using a *local* algorithm (such as [11, 31, 38]) 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. It is possible to make quite an expensive choice for the global partitioner (a spectral algorithm for example [14]) since it is only applied once and to the coarsest graph. Moreover, the technique may either be applied to find a bisection of the original graph and then be repeated recursively on each subgraph, or it may be applied to find a k -way partition directly.

Recently (in [30]) Karypis and Kumar have proposed a parallel version of their sequential multilevel technique (see [29]) which is suitable for the dynamic load-balancing problem. In this parallel version the original graph is already partitioned into k subgraphs and the coarsening algorithm only permits vertices in the same subgraph to be merged together at each level. The coarsest graph in the hierarchy can then be repartitioned quite cheaply (since it is small) before the refinement stages begin. At each of these refinement stages further transfer of vertices is allowed to take place so as to permit local improvements to the partition through the use of a greedy heuristic. Figure 3 illustrates this process by showing one level of coarsening, followed by a repartition and then one level of refinement. Notice that after the refinement stage the partition has been modified locally.

Initial results using parallel multilevel algorithms such as that described here (see also [55]) are quite encouraging. Nevertheless there are numerous difficulties associated with keeping the amount of interprocessor communication under control: especially at the local improvement stages (with the possibility of thrashing not entirely eliminated). Also, it is not clear how important the choice of partitioning algorithm for the coarsest graph will be in general, nor what the optimal degree of coarsening should be. Hence quite a lot of further research is still required in this area.

3.2 Diffusion algorithms

In section 2 of this paper the dynamic load-balancing problem is introduced in the context of parallel adaptive computational mechanics software. In this

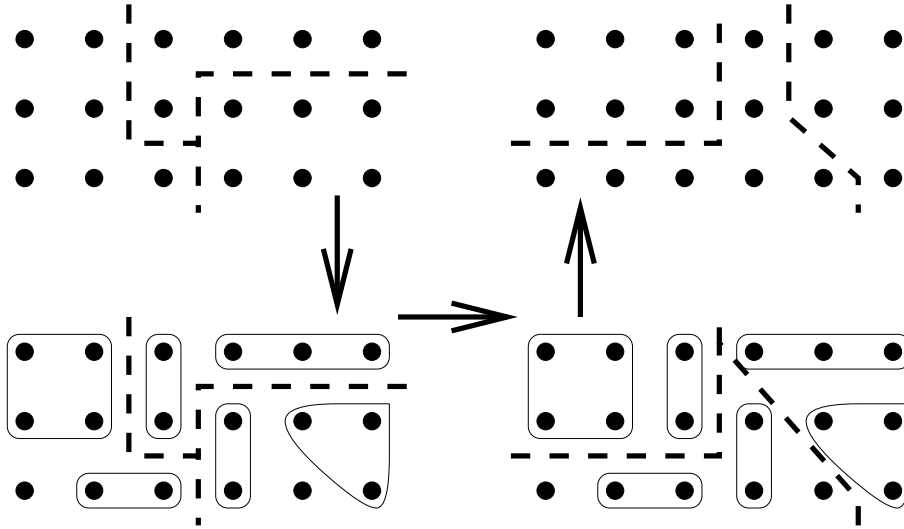


Figure 3: A schema showing a single level of the coarsening and refinement process for Karypis and Kumar’s k -way algorithm (here $k = 3$).

subsection we briefly consider the application of a very general class of dynamic load-balancing techniques to such software: algorithms based upon diffusion [4, 5, 19, 33, 44, 58, 59]

To illustrate the simple idea behind diffusion algorithms it is convenient to introduce a new graph which, following [53], we call a weighted processor communication graph (WPCG). Given a partition of the weighted dual graph of a finite element mesh onto P processors, the WPCG has P vertices: one for each processor. The weight of each vertex is equal to the total number of actual elements in the corresponding subdomain, and two vertices are connected by an edge if the corresponding subdomains are face adjacent (the weight of this edge being equal to the number of element faces shared by the subdomains). If one imagines the WPCG as a physical network where each vertex is a point at a certain temperature (given by its weight) and each edge is a wire which connects a pair of points, then a diffusion algorithm simply attempts to simulate the flow of thermal energy through this network until equilibrium is reached. At equilibrium each point will be the same temperature which, in this analogy, corresponds to an equal load on each processor.

It is apparent therefore that diffusion algorithms are iterative processes in which neighbouring processors compare loads and, at each iteration, a proportion of the difference is passed from the heavier to the lighter loaded processor. Such algorithms are inherently parallel and only require local communication between individual processors. In the well-known paper of Cybenko [5], the convergence of such schemes is proved and the rates of convergence are studied for a number of processor topologies. (Note that there are two different possible definitions of “neighbouring processors” that could be used here. Either their physical connectivity may be considered (as in [5]) or their connectivity within the WPCG may be used (which is more likely to keep a short partition boundary). These two definitions clearly lead to very different algorithms: with the latter being the

more appropriate for the application being considered here.)

One of the conclusions of Cybenko’s analysis (and others such as Boillat [4]), which is readily observed in practice, is that the classical diffusion algorithm is very slow to converge (especially when near to equilibrium). He therefore proposes a more robust variant of the algorithm, known as the dimension exchange method [5], which is designed specifically with a hypercube architecture in mind. This idea has been generalized and analyzed further by Xu and Lau [58, 59]. A rather different approach to achieving faster convergence has been proposed by Horton in [19]. This makes use of the multilevel idea that we have already encountered so as to allow the load to be shifted around the network more quickly and coarsely in the early stages.

Despite these, and numerous other, improvements to the basic diffusion algorithm (e.g. an asynchronous version due to Song [44]), there are still a number of major drawbacks associated with using these methods for balancing adaptive finite element computations. In particular, no information is given about *which* elements should be transferred from one processor to another: one only calculates the total weight to be transferred. Moreover, this weight is a real number whereas the vertices of the dual graph have weights which are integers (possibly quite large in areas of heavy local refinement), which means that achieving the desired transfer weight may not be possible in practice. Finally, it should be noted that, in general, diffusion algorithms take no explicit account of the requirement that we have to keep the total amount of data migration to a minimum.

3.3 Minimizing data migration

At the end of section 2 four properties are stated which we would like our dynamic load-balancing algorithm to possess. In fact, it is not difficult to see that these requirements are not generally self-consistent. For example, given an initial partition, it may be impossible to obtain close to the minimum possible number of elements on the partition boundary (subject to the load-balance constraint) without a substantial amount of data migration. It follows therefore that there has to be some trade-off between all of these requirements: we would like our subdomains to be approximately load-balanced and the interpartition boundary to be reasonably short but we would also like to avoid too much data migration. The diffusion algorithms introduced in the previous subsection place their emphasis on the quality of the final partition more than on the need to minimize data migration. In contrast, in this subsection we consider a technique which shifts this emphasis onto maintaining as much data locality as possible.

We will again make use of the weighted processor communication graph (WPCG) defined in the previous subsection. Following the notation of Hu and Blake [21] for this graph, let us denote by ℓ_i the total load on each processor (i.e. the weight of vertex i for $i = 1, \dots, P$) and by $\bar{\ell}$ the average load per processor. If δ_{ij} is the total weight that we are to transfer from processor i to processor j (where the corresponding vertices of the WPCG are connected by an edge,

(i, j)), then to balance the load across the processors we require

$$\sum_{(i,j) \in E} \delta_{ij} = \ell_i - \bar{\ell} \quad \text{for } i = 1, \dots, P-1, \quad (10)$$

where E is the set of all edges in the WPCG. (Note that if equations (10) are satisfied then it must also follow that the same equation also holds for $i = P$.) Given that $\delta_{ij} = -\delta_{ji}$, Hu and Blake observe in [21] that (10) corresponds to $P-1$ equations in $|E|$ unknowns. In general, the number of edges will be strictly greater than $P-1$ (often much greater) and this problem will have an infinite number of solutions (at worst $|E| = P-1$ for a connected graph and there will be a unique solution). Hence Hu and Blake propose an algorithm for finding the solution of (10) which minimizes the Euclidean norm of the vector whose components are δ_{ij} for each $(i, j) \in E$. They therefore produce a repartitioning schedule which minimizes the total amount of data migration in the L^2 norm.

As with the diffusion algorithms described in 3.2 Hu and Blake’s method does not tell us anything about which particular elements should be transferred between processors when applied to the adaptive finite element problem. This choice should be made so as to try to keep the length of the new partition boundary as short as possible: possibly based upon one of the local heuristics in [11, 31]. Even if the best possible choice is made for which elements to migrate however, this approach is unlikely to yield a subdomain boundary which is as short as that obtained from a spectral method, for example, since there is nothing in the algorithm which explicitly attempts to keep this boundary short.

Despite these problems, there does appear to be a lot of potential to the minimization approach of [21], and this is possibly best demonstrated by the work described in [55]. Here Walshaw *et al* make use of Hu and Blake’s method within their own, more robust, code. In addition, this work makes use of graph coarsening to reduce the size of the global partitioning problem and local improvements to the partition when refining the coarsened graphs. Interestingly, in [55] it is observed that if one solves the linear algebra problem associated with (10) using an iterative method, then Hu and Blake’s algorithm may be interpreted as a diffusion method for which the diffusive coefficients change at each iteration. Preliminary results from Walshaw *et al*, reported in [55], suggest that their quite sophisticated algorithm often strikes a good balance between the conflicting requirements of partition quality and data locality.

3.4 Other recursive methods

We finish the section by briefly outlining two more dynamic load balancing algorithms that may be used in parallel adaptive computational mechanics codes. As with the geometric and spectral approaches outlined in 3.1.1 and 3.1.2 respectively, these algorithms are also recursive in nature: often referred to as divide and conquer. Unlike the previous approaches however the algorithms introduced in this section rely entirely on local migration heuristics, such as [11, 31, 38], and contain no global repartitioning step.

In [28, 53] a parallel adaptive three-dimensional unstructured finite element code is described for the solution of Navier-Stokes problems with complex ge-

ometries. The load-balancing strategy used here assumes that the number of processors is a power of two and begins by splitting the processors into two equal groups based upon their IDs. Following this the total load in each group is determined so that half of the difference may be transferred from one to the other. The group with the higher load is labeled as the *sender group* whilst the other is labeled as the *receiver group*. Only those processors in the sender group which are face adjacent to a processor in the receiver group are allowed to pass elements between the groups and each such processor concurrently sends a proportion of the total load to be transferred. Moreover, the actual elements which are transferred are chosen only from those elements which lie on the partition boundary or which are contiguous to other elements being transferred. These restrictions are designed to try to prevent the length of the new partition boundary from becoming too large.

Once the bisection into two groups is complete the above algorithm may be repeated recursively in each group until there is just one processor per group. At this point the load will be equally balanced across the processors because such a balance is always maintained between the groups. As is pointed out in [53], there are two ways in which parallelism may be exploited in the implementation of this algorithm: as well as the divide and conquer approach allowing greater parallelism after each level of the recursion, it is also possible to implement in parallel the sends and receives between groups at the same level.

In [48, 49] a generalization of this approach is described in which a more robust version of the divide and conquer algorithm is sought. The method still divides the processors into two groups recursively but unlike in [53] these groups are not selected according to their processor IDs but instead by bisecting the weighted processor communication graph (WPCG). This bisection is obtained using a weighted spectral algorithm ([14]) and does not necessarily create groups with equal numbers of processors but instead attempts to create groups with approximately equal weights. Now the sender group is the group with the larger average weight per processor and the total load to be migrated is such that the final average weight per processor should be the same in both groups. As with [53], the algorithm described in [48, 49] only permits those processors in the sender group which are face adjacent to a processor in the receiver group to send elements. The particular elements that are migrated are selected in a different manner than in [53] however. This is based upon a generalization of the notion of a “gain” that is used in many local algorithms (see [31] for example). The gain in migrating a vertex of a partitioned graph from one partition to another is defined to be the decrease in the weight of the edges of the graph that are cut by the partition boundary should that migration take place. In [48, 49] the “gain density” (i.e. the gain divided by the weight of the node) is used to determine which particular elements should be migrated at each stage.

In assessing the quality of these recursive algorithms we again observe that there is a trade-off between the quality of the final partition and the amount of data that we wish to migrate. This is especially true if the initial partition is quite poor: immediately after heavy local mesh refinement has occurred on just one or two processors for example. In comparison with the optimal algorithm

of Hu and Blake [21] (see subsection 3.3) for example, these techniques result in significantly more data migration. However, as demonstrated in [48, 49], the quality of the new partition is often superior. If, on the other hand, the initial partition is not that poor then this may usually be improved with very little data migration and the cheaper algorithm is generally just as effective.

4 Discussion

This paper attempts to review quite a large amount of material in order to introduce the reader to the main features of dynamic load-balancing for adaptive computational mechanics codes. It is inevitable therefore that most of the topics covered have not been discussed in very much depth and the reader is encouraged to make use of the bibliography below in order to follow up any work that is of particular interest to them.

There is also one extremely important issue that has not yet been addressed at all in this paper. This is the question of assessing when it is worth the expense of repartitioning a mesh, as opposed to continuing with an existing partition which may be a little unbalanced (as a result of some local mesh refinement for example). This is an extremely difficult question to answer in general since it depends on such a large number of different factors (some of which are impossible to determine in advance). Clearly the benefits of obtaining a new partition must outweigh the cost of achieving this partition. Hence it is necessary to have some computational model which measures the impact on the parallel solver of altering such things as the number of elements per processor, the number of edges each processor has on the partition boundary, the total number of edges on the partition boundary, the number of neighbours each processor has in the WPCG, etc. In turn, each of these factors will depend not only upon the parallel solver being used but also on the computer architecture it is being executed on. It will therefore be necessary to know the computational power of each processor, the physical connectivity of the processors and the communication latency and bandwidth. In addition it will be necessary to have a computational model for the repartitioning algorithm so that one may assess the cost of invoking this. Nevertheless, there is no obvious way of knowing in advance just how much data will be transferred (and therefore the communication overhead of repartitioning) or how much better the final partition will be than the original (to compute the improvement in the parallel solver). In general therefore most parallel adaptive codes that have so far been written have tended to be rather cautious about deciding when to repartition ([8, 53] for example). There is clearly considerable scope for research in this area so as to help with this important decision.

Finally, we conclude the paper with the comment that nearly all of the algorithms discussed in section 3 have their own strengths and weaknesses, and that it certainly is not possible to say that any one of them is better than the others. All of the algorithms are heuristics and they are all trade-offs between the competing factors enumerated at the end of section 2. This has been recognized by those who have attempted to produce software for this class of problem (including [30, 41, 55]), since most of their algorithms are hybrids of a number of

techniques; including coarsening, global repartitioning and local repartitioning algorithms. It is my belief that any robust dynamic load-balancing software of the future will need to be able to (dynamically) choose the most appropriate choice of algorithm from a wide selection available to it, based upon the configuration and performance of the parallel architecture and the way in which the finite element solution and the adaptivity are developing.

References

- [1] S.T. Barnard and H.D. Simon, “*A Fast Multilevel Implementation of Recursive Bisection*”, in Proc. of the Sixth SIAM Conf. on Parallel Processing for Scientific Computing (R.F. Sincovec *et al*, eds.), SIAM, Philadelphia, 1993.
- [2] S.T. Barnard and H.D. Simon, “*A Parallel Implementation of Multilevel Recursive Spectral Bisection for Application to Adaptive Unstructured Meshes*”, in Proc. of the Seventh SIAM Conf. on Parallel Processing for Scientific Computing (D.H. Bailey *et al*, eds.), SIAM, Philadelphia, 1995.
- [3] A. Bahreininejad, B.H.V. Topping and A.I. Khan, “*Finite Element Mesh Partitioning Using Neural Networks*”, Advances in Engineering Software, 27, 103–115, 1996.
- [4] J.E. Boillat, “*Load balancing and Poisson Equation in a Graph*”, Concurrency: Practice and Experience, 2, 289–313, 1990.
- [5] G. Cybenko, “*Dynamic Load Balancing for Distributed Memory Multiprocessors*”, J. of Parallel and Distributed Computing, 7, 279–301, 1989.
- [6] L. Demkowicz, J.T. Oden and W. Rachowicz, “*A New Finite Element Method for Solving Compressible Navier-Stokes Equations Based on an Operator Splitting Method and h-p Adaptivity*”, Computer Methods in Applied Mechanics and Engineering, 84, 275–326, 1990.
- [7] L. Demkowicz, J.T. Oden, W. Rachowicz and O. Hardy, “*An h-p Taylor-Galerkin Finite Element Method for Compressible Euler Equations*”, Computer Methods in Applied Mechanics and Engineering, 88, 363–396, 1991.
- [8] K.D. Devine and J.E. Flaherty, “*Parallel Adaptive HP-Refinement Techniques for Conservation Laws*”, Applied Numerical Mathematics, 20, 367–386, 1996.
- [9] P. Diniz, S. Plimpton, B. Hendrickson and R. Leland, “*Parallel Algorithms for Dynamically Partitioning Unstructured Grids*”, in Proc. of the Seventh SIAM Conf. on Parallel Processing for Scientific Computing (D.H. Bailey *et al*, eds.), SIAM, Philadelphia, 1995.
- [10] C. Farhat, “*A Simple and Efficient Automatic FEM Domain Decomposer*”, Computers and Structures, 28, 579–602, 1988.

- [11] C.M. Fiduccia and R.M. Mattheyses, “*A Linear Time Heuristic for Improving Network Partitions*”, in Proc. 19th IEEE Design Automation Conference, IEEE, 1982.
- [12] G. Globisch, “*PARMESH: A Parallel Mesh Generator*”, *Parallel Computing*, 21, 509–524, 1995.
- [13] G.H. Golub and C.F. Van Loan, “*Matrix Computations*”, John Hopkins Press, 2nd edition, 1989.
- [14] B. Hendrickson and R. Leland, “*An Improved Spectral Graph Partitioning Algorithm for Mapping Parallel Computations*”, *SIAM J. on Scientific Computing*, 16, 452–469, 1995.
- [15] B. Hendrickson and R. Leland, “*A Multilevel Algorithm for Partitioning Graphs*”, Technical Report SAND 93-1301, Sandia National Laboratories, 1993.
- [16] B. Hendrickson and R. Leland, “*The Chaco Users’ Guide*”, Technical Report SAND 93-2339, Sandia National Laboratories, 1993.
- [17] D.C. Hodgson and P.K. Jimack, “*Efficient Mesh Partitioning for Parallel Elliptic Differential Equation Solvers*”, *Computing Systems in Engineering*, 6, 1–12, 1995.
- [18] D.C. Hodgson and P.K. Jimack, “*Efficient Generation of Partitioned Unstructured meshes*”, *Advances in Engineering Software*, 27, 59–70, 1996.
- [19] G. Horton, “*A Multi-Level Diffusion Method for Dynamic Load Balancing*”, *Parallel Computing*, 19, 209–218, 1993.
- [20] P. Houston and E. Suli, “*Adaptive Lagrange-Galerkin Methods for Unsteady Convection-Dominated Diffusion Problems*”, Technical Report NA 95/24, Oxford University Computing Laboratory, 1995.
- [21] Y.F. Hu and R.J. Blake, “*An Optimal Dynamic Load Balancing Algorithm*”, Preprint DL-P-95-011, Daresbury Laboratory, 1995.
- [22] T.J.R. Hughes, L.P. Franca and M. Mallet, “*A New Finite Element Formulation for Computational Fluid Dynamics: VI, Convergence Analysis of the Generalized SUPG Formulation for Linear Time-Dependent Multi-Dimensional Advective-Diffusive Systems*”, *Computer Methods for Applied Mechanics and Engineering*, 63, 97–112, 1987.
- [23] T.J.R. Hughes, I. Levit and J. Winget, “*An Element-by-Element Solution Algorithm for Problems of Structural and Solid Mechanics*”, *Computer Methods for Applied Mechanics and Engineering*, 36, 241–254, 1983.
- [24] P.K. Jimack, “*A New Approach to Finite Element Error Control for Time-Dependent Problems*”, in *Numerical Methods for Fluid Dynamics 4* (M.J. Baines and K.W. Morton, eds.), Oxford University Press, 1993.

- [25] P.K. Jimack and D.C. Hodgson, “*Parallel Preconditioners Based Upon Domain Decomposition*”, in *Parallel and Distributed Processing for Computational Mechanics I* (B.H.V. Topping, ed.), Saxe-Coburg Publications, 1997.
- [26] C. Johnson “*Numerical Solution of Partial Differential Equations by the Finite Element Method*”, Cambridge University Press, 1987.
- [27] C. Johnson “*Adaptive Finite Element Methods for Diffusion and Convection Problems*”, *Computer Methods for Applied Mechanics and Engineering*, 82, 301–322, 1990.
- [28] Y. Kallinderis and A. Vidwans, “*Generic Parallel Adaptive-Grid Navier-Stokes Algorithm*”, *AIAA Journal*, 32, 54–61, 1994.
- [29] G. Karypis and V. Kumar, “*A Fast High Quality Multilevel Scheme for Partitioning Irregular Graphs*”, Technical Report TR 95-035, Department of Computer Science, University of Minnesota, 1995.
- [30] G. Karypis and V. Kumar, “*A Coarse Grain Parallel Formulation of Multilevel k -way Graph Partitioning Algorithm*”, in *Proc. of the Eighth SIAM Conf. on Parallel Processing for Scientific Computing* (M. Heath *et al*, eds.), SIAM, Philadelphia, 1997.
- [31] B. Kernighan and S. Lin, “*An Efficient Heuristic Procedure for Partitioning Graphs*”, *Bell System Technical Journal*, 49, 291–307, 1970.
- [32] A.I. Khan and B.H.V. Topping, “*Parallel Adaptive Mesh Generation*”, *Computing Systems in Engineering*, 2, 75–101, 1991.
- [33] G.A. Kohring, “*Dynamic Load Balancing for Parallel Particular Simulation on MIMD Computers*”, *Parallel Computing*, 21, 683–693, 1995.
- [34] J.D. Lambert, “*Numerical Methods for Ordinary Differential Systems*”, Wiley, 1991.
- [35] C.A. Leete, B.W. Peyton and R.F. Sincovec, “*Toward a Parallel Recursive Spectral Bisection Mapping Tool*”, in *Proc. of the Sixth SIAM Conf. on Parallel Processing for Scientific Computing* (R.F. Sincovec *et al*, eds.), SIAM, Philadelphia, 1993.
- [36] R. Lohner, “*An Adaptive Finite Element Method for Transient Problems in CFD*”, *Computer Methods in Applied Mechanics and Engineering*, 61, 323–338, 1987.
- [37] R. Lohner, R. Camberos and M. Merriam, “*Parallel Unstructured Grid Generation*”, *Computer Methods in Applied Mechanics and Engineering*, 95, 343–357, 1992.
- [38] B. Monien and R. Diekmann, “*A Local Graph Partitioning Heuristic Meeting Bisection Bounds*”, in *Proc. of the Eighth SIAM Conf. on Parallel Processing for Scientific Computing* (M. Heath *et al*, eds.), SIAM, Philadelphia, 1997.

- [39] B. Nour-Omid, A. Raefsky and G. Lyzenga, “*Solving Finite Element Equations on Concurrent Computers*”, in *Parallel Computations and Their Impact on Mechanics* (A.K. Nour, ed.), American Soc. Mech. Eng., 1986.
- [40] J.T. Oden, T. Strouboulis and P.H. Devloo, “*Adaptive Finite Elements for High Speed Compressible Flow*”, *International J. for Numerical Methods in Fluids*, 7, 1211–1228, 1987.
- [41] R. Preis and R. Diekmann, “*The PARTY Partitioning Library User Guide*”, Technical Report rsfb-96-024, Department of Computer Science, University of Paderborn, 1996.
- [42] P.M. Selwood, M. Berzins and P.M. Dew, “*3D Parallel Mesh Adaptivity: Data-Structures and Algorithms*”, in *Proc. of the Eighth SIAM Conf. on Parallel Processing for Scientific Computing* (M. Heath *et al*, eds.), SIAM, Philadelphia, 1997.
- [43] H.D. Simon, “*Partitioning of Unstructured Problems for Parallel Processing*”, *Computing Systems in Engineering*, 2, 135–148, 1991.
- [44] J. Song, “*A partially Asynchronous and Iterative Algorithm for Distributed Load Balancing*”, *Parallel Computing*, 20, 853–868, 1994.
- [45] W.E. Spears and M. Berzins, “*A Fast 3-D Unstructured Mesh Adaption Algorithm with Time-Dependent Upwind Euler Shock Diffraction Calculations*”, in *Proc. of the Sixth International Symposium on Computational Fluid Dynamics* (M. Hafex and K. Oshima, eds.), Vol III, 1995.
- [46] T. Strouboulis and J.T. Oden, “*A Posteriori Error Estimates of Finite Element Approximations in Fluid Mechanics*”, *Computer Methods in Applied Mechanics and Engineering*, 78, 201–247, 1990.
- [47] B.H.V. Topping and A.I. Khan, “*Parallel Finite Element Computations*”, Saxe-Coburg Publications, 1996.
- [48] N. Touheed and P.K. Jimack “*Parallel Dynamic Load-Balancing for Adaptive Distributed Memory PDE Solvers*”, School of Computer Studies Research Report 96.34, University of Leeds, 1996.
- [49] N. Touheed and P.K. Jimack “*Dynamic Load-Balancing for Adaptive PDE Solvers with Hierarchical Refinement*”, in *Proc. of the Eighth SIAM Conf. on Parallel Processing for Scientific Computing* (M. Heath *et al*, eds.), SIAM, Philadelphia, 1997.
- [50] N. Touheed and P.K. Jimack “*Improved Parallel Mesh Generation Through Dynamic Load-Balancing*”, in *Advances in Computational Mechanics for Parallel Processing* (B.H.V. Topping, ed.), Civil-Comp Press, 1997.
- [51] R. Van Driessche and D. Roose, “*An Improved Spectral Bisection Algorithm and its Application to Dynamic Load Balancing*”, *Parallel Computing*, 21, 29–48, 1995.

- [52] V. Venkatakrishnan and H.D. Simon, “*A MIMD Implementation of a Parallel Euler Solver for Unstructured Grids*”, Int. J. Supercomputer Applications, 6, 117–127, 1992.
- [53] A. Vidwans, Y. Kallinderis and V. Venkatakrishnan, “*Parallel Dynamic Load-Balancing Algorithm for 3-Dimensional Adaptive Unstructured Grids*”, AIAA Journal, 32, 497–505, 1994.
- [54] C. Walshaw and M. Berzins, “*Dynamic Load-Balancing for PDE Solvers on Adaptive Unstructured Meshes*”, Concurrency: Practice and Experience, 7, 17–28, 1995.
- [55] C. Walshaw, M. Cross and M.G. Everett, “*Dynamic Load-Balancing for Parallel Adaptive Unstructured Meshes*”, in Proc. of the Eighth SIAM Conf. on Parallel Processing for Scientific Computing (M. Heath *et al*, eds.), SIAM, Philadelphia, 1997.
- [56] C. Walshaw, M. Cross, M.G. Everett, S. Johnson and K. McManus, “*Partitioning and Mapping of Unstructured Meshes to Parallel Machine Topologies*”, in Irregular '95: Parallel Algorithms for Irregularly Structured Problems (A. Ferreira and J. Rolim, eds.), Springer, 1995.
- [57] R.D. Williams, “*Performance of Dynamic Load Balancing for Unstructured Mesh Calculations*”, Concurrency: Practice and Experience, 3, 457–481, 1991.
- [58] C.Z. Xu and F.C.M. Lau, “*Analysis of the Generalized Dimension Exchange Method for Dynamic Load Balancing*”, J. of Parallel and Distributed Computing, 16, 385–393, 1992.
- [59] C.Z. Xu and F.C.M. Lau, “*The Generalized Dimension Exchange Method for Load Balancing in K-ary Ncubes and Variants*”, J. of Parallel and Distributed Computing, 24, 72–85, 1995.