LOAD BALANCING FOR UNSTRUCTURED MESH APPLICATIONS

Y. F. HU AND R. J. BLAKE*

Abstract. The efficient use of distributed memory parallel systems requires the load on each processor to be well balanced. In cases where the load changes unpredictably during the computation, a dynamic load balancing strategy is needed. Load balancing problems have been studied extensively in recent years, particularly in the context of unstructured mesh based applications. Static load balancing can be approximated by a graph partitioning problem and many efficient algorithms have been developed. Significant progress has also been made in the development of dynamic load balancing algorithms. This paper looks at the history and the state of the art of both classes of algorithms, with a particular emphasis on mesh based applications. However the underlying algorithms, including those for graph partitioning and flow calculation, are sufficiently generic to be applicable to other applications.

Key words. graph partitioning, dynamic load balancing, diffusion algorithms, graph theory

AMS subject classifications. 65Y05

1. Introduction. The numerical solution of partial differential equations usually involves dividing up the physical domain into small elements or volumes to form a mesh. To solve the problem on a distributed memory parallel computer, the mesh should be decomposed into subdomains, the number of which usually equals the number of processors, with each subdomain assigned to a unique processor.

The *static load balancing* problem is that of how to decompose the mesh into subdomains so that each processor has about the same amount of computation and so that the communication cost between processors is minimized, with the overall aim of minimizing the runtime of the calculation.

After the grid is partitioned and the parallel solver code has been executed for a number of iterations, the mesh may need to be refined (or coarsened) at certain locations, based on an estimate of the discretization errors. As a consequence of the refinement (or coarsening), some processors may have significantly more (or less) elements than others, thus it becomes necessary to re-balance the load in order to maintain the parallel efficiency. This is the *dynamic load balancing* problem.

Usually the load of each processor remains fixed for many iterations in between mesh adaptations, and the load balancing step is only needed just after a mesh refinement or coarsening. Such quasi-dynamic load balancing problems are relatively easy to solve, although still far more difficult than the static load balancing problem.

In this paper developments over recent years and the state of the art in static and dynamic load balancing algorithms will be reviewed. The discussion is biased towards unstructured mesh based applications, although the techniques discussed here may be used in many other areas where static and dynamic load balancing problems appear, for example, task scheduling in operating systems. Section 2 introduces some useful notation, followed by Section 3 which looks at the classical as well as recent graph partitioning algorithms. Section 4 discusses dynamic load balancing algorithms. The paper finishes with some future research topics in Section 5 and conclusions in Section 6.

2. Graph Theory Notation. Much of the load balancing problem can be described using terminology from graph theory. In fact some of these problems had

*Daresbury Laboratory, Daresbury, Warrington WA4 4AD, United Kingdom (Y.F.Hu@dl.ac.uk).

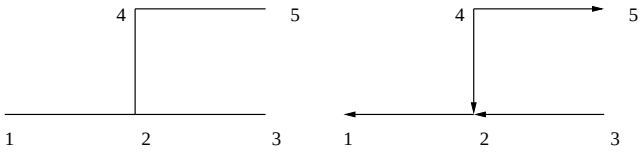

FIG. 2.1. *An undirected graph (left) and a directed graph (right)*

already been studied in graph theory before they appeared in the context of parallel computing.

A *graph* G has two key components:

- The vertex set V, which is just a list of indices;
- The edge set E. Each edge consists of two vertices.

The number of vertices and edges of a graph are denoted by $|V|$ and $|E|$. The number of edges connected to a vertex is called the degree of the vertex. The degree of the graph is the largest degree of all of its vertices.

A graph can be *directed*, in which case there is a beginning and an end vertex (also called the *head* and the *tail*, respectively) for each edge. A graph can also be *undirected*. For example, Figure 2.1 illustrates an undirected graph (left) and a directed graph (right). The undirected graph has 5 vertices, $V = \{1, 2, 3, 4, 5\}$, and 4 edges, $E = \{(1, 2), (2, 3), (2, 4), (4, 5)\}$. The number of vertices and edges are therefore $|V| = 5$ and $|E| = 4$ respectively. The directed graph also has 5 vertices and 4 edges, although each edge has a direction associated. For instance, the edge between vertex 4 and 2 has vertex 4 as the head and vertex 2 as the tail. All graphs in this paper are undirected, unless specified otherwise.

If two vertices i and j form an edge, this will be denoted as $(i, j) \in E$, or $i \leftrightarrow j$. A graph is *connected*, if for any two vertices i and j, there exists a *path* connecting them. Here a path is defined as a list of vertices $\{i, i_1, i_2, \ldots, i_k, j\}$ such that any two subsequent vertices in this list form an edge.

The distance between two vertices of a graph is defined as the minimum number of edges among all paths connecting these two vertices. For the undirected graph in Figure 2.1, the distance between vertex 1 and vertex 5 is 3. This is different to the physical distance between the two points in the Euclidean space.

3. The Static Load Balancing Problem. The static load balancing problem for a mesh based application involves partitioning the mesh into subdomains. The subdomains can then be distributed over the processors and calculation carried out in parallel. Different partitions of the mesh may result in different times to completion for the calculation. It is therefore necessary to examine the quality of the partitioning based on its effect on the application code. There are a number of factors.

The first factor is that of the load balance. The computational work of each processor should be balanced, so that no processor will be waiting for others to complete. Assuming that the computational work per processor is proportional to the number of mesh nodes in the subdomain, then to achieve load balance it is necessary for the number of nodes in each subdomain to be the same.

The second factor is that of the communication cost. When forming the dis-

cretized equations on a node of the mesh, the contributions from its nearest neighbor nodes will usually be needed. Depending on the order of the discretization scheme, contributions from more distant neighboring nodes may be necessary. On a parallel computer, accumulating the contributions from nodes that are not on the current processor will incur communication cost. It is known that on distributed memory parallel computers the cost of accessing remote memory is far higher than that of accessing local memory (typically a ratio of between 10 to 1000). It is therefore important to minimize the communication cost.

There are other factors that affect the time to completion of a parallel calculation. Often, to maximize the parallelism, the solver algorithm implemented on a parallel computer is not chosen to be mathematically equivalent to the sequential algorithm. Therefore the number of iterations needed for a parallel code to converge on multiprocessors may not be the same as that on a single-processor. This is particularly true when implicit algorithms are used, and the way a mesh is partitioned can affect the rate of convergence considerably. However for now, we will in essence assume that a linear explicit solver is being used and seek to meet two criteria

- Each subdomain should have a roughly equal number of nodes;
- The communication cost is minimized.

It is useful to introduce the idea of a communication graph, and to distinguish the computational mesh from its communication graph. The communication graph characterizes the data dependency of the computation on the mesh. For instance, when constructing a communication graph for the purpose of partitioning a finite element mesh,

- If the finite element algorithm has a node based data structure, then computation on each node requires data from neighboring nodes (those nodes which share an edge with this node). Therefore data dependency is along the edges of the mesh, in which case vertices of the communication graph are simply the nodes of the mesh, and the edges of the graph coincide with the edges of the mesh.
- Alternatively, if the finite element algorithm has an element based data structure, then computation on each element requires data from neighboring elements (those elements which share a face with this element). Therefore the data dependency is across the faces of the mesh. Vertices of the communication graph are now at the centroid of the elements, and two vertices of the graph form an edge if the corresponding elements share a face. Such a graph is called the *dual graph* of the mesh.

The partitioning of a mesh is usually based on the partitioning of its corresponding communication graph. The number of edges of the communication graph cut by a partitioning is called the *number of edges-cut* (or *edge-cut* for short) of the partitioning. The communication cost of a subdomain is a function of its edge-cut as well as the number of neighboring subdomains that share edges with it. In practice the edge-cut of a partitioning is usually used as an important indicator of the quality of the partitioning. Figure 3.1 shows a mesh of 788 elements for an element based finite element calculation, and the dual graph of the mesh. The partitioning of the dual graph (Figure 3.1 (right)) into 4 subdomains, using the recursive coordinate bisection (RCB) algorithm described in Section 3.1.1, results in the partitioning of the mesh shown in Figure 3.1 (left).

When there are only two subdomains or processors, the graph partitioning problem becomes a graph bisection problem, where given a graph $G = (V, E)$ with vertices

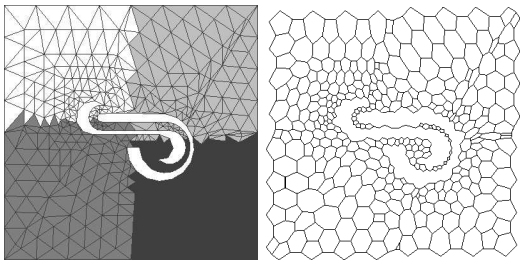

FIG. 3.1. *A mesh of 788 elements (left), and its dual graph with 788 vertices (right)*

V and edges E, it is required to find a partition $V = V_1 \bigcup V_2$ such that $V_1 \bigcap V_2 = \emptyset$, $|V_1| \simeq |V_2|$ and the edge-cut $|E_c|$:

$$|E_c| = |\{h \mid h \in E; h = (v_1\ v_2);\ v_1 \in V_1,\ v_2 \in V_2\}|$$

is minimized. Here for a set S, one denotes $|S|$ as the number of elements in the set.

The graph bisection problem has been studied in the past by many authors (see, e.g., [53, 63, 11]) in the context of graph theory as well as VLSI circuit layout. Advances in parallel computing hardware and software has renewed interest in the problem. The graph bisection problem is an NP hard problem, so there are no known algorithms that can find the exact solution to the problem in polynomial time. Most of the graph bisection methods therefore seek a good approximation to the optimal partitioning that can be calculated efficiently.

In the following a number of partitioning algorithms are described. Many of them are bisection based. A partitioning for an arbitrary number of processors requires the recursive use of the bisection algorithm (for a number that is not a power of two, the bisection will give two subdomains of different sizes, see [42]). We shall denote $n = |V|$ the number of vertices of the graph, and p the number of processors (subdomains).

3.1. Geometric Based Algorithms. Since in practice all meshes are within a physical domain and have some coordinates associated with them, it is intuitive to classify the nodes based on their physical orientation. The shortcomings of this approach are obvious. Among others, the result of such a partitioning is not invariant to geometric transformation of the mesh, such as a perturbation of the mesh nodes.

3.1.1. Coordinate bisection. The *recursive coordinate bisection* (RCB) algorithm [87] partitions the graph according to the coordinates of the vertices. For this purpose the coordinate of a vertex of the communication graph is taken as that of its corresponding node of the mesh, or that of the centroid of the corresponding element in the case of a dual graph. The RCB algorithm divides the graph into two halves according to the x-coordinates of the vertices, then further into four according to the y-coordinates, etc.. This works well when the mesh is evenly spread over a simple domain. Otherwise it can create long boundaries with disconnected subdomains. This situation can be alleviated by the so called unbalanced recursive bisection approach [46], where instead of dividing the vertices into equal sets, one chooses the partitioning that minimizes the aspect ratio and divides the graph into subgraphs of nk/p and $n(p-k)/p$ vertices respectively. Here n is the number of vertices in the graph, p the number of processors and $k \in \{1, 2, \ldots, p-1\}$. By suitable choice of k at each bisection, this approach leads to subdomains of equal number of vertices but with better aspect ratios.

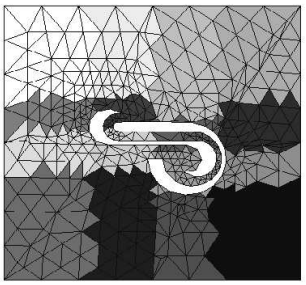

FIG. 3.2. *Partitioning a mesh into 16 subdomains using RCB*

Figure 3.2 shows the result of applying the RCB to the dual graph of a mesh with 788 elements. It creates 142 inter-boundary edges.

3.1.2. Inertia bisection. Partitioning using coordinate bisection is susceptible to the orientation of the mesh, a simple rotation of the mesh would result in different partition. *Inertia bisection* [54] remedies this by using a procedure that is invariant to rotation. It finds the principal axis of the communication graph. Here the coordinates of the vertices of the communication graph are defined as explained in Section 3.1.1. If one assumes that each vertex has a unit mass associated with it, then the principal axis is the axis that minimizes the angular momentum when the graph rotates around it.

Consider a graph of n vertices associated with a 3D mesh. Let the principal axis be $y_0 + \alpha * y$ with y_0 a point in space, α real numbers and y a unit vector. Then any vertex i with coordinates $x_i \in R^3$ has an angular momentum equal to the square of its distance to the axis, or

$$||x_i - y_0||^2 - \frac{(x_i - y_0)^T y}{||y||^2} = ||x_i - y_0||^2 - (x_i - y_0)^T y, \quad i = 1, \ldots, n$$

The principal axis minimizes the sum of these momentums:

$$\sum_{i=1}^{n} \left(||x_i - y_0||^2 - (x_i - y_0)^T y \right)$$

subject to $||y||^2 = 1$. It is possible to prove, using the necessary condition of constrained optimization, that $y_0 = \bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i$ and that y is the eigenvector that corresponds to the largest eigenvalue of the 3×3 matrix

$$I = \sum_{i=1}^{n} (x_i - \bar{x})(x_i - \bar{x})^T \tag{3.1}$$

Once this eigenvector has been found, every vertex can be sorted based on its projection onto the principal axis, that is, the value $(x_i - \bar{x})^T y$, or simply $x_i^T y$.

The main computational cost of the inertia algorithm is the formation of the inertia matrix (3.1), which requires of the order of $O(n)$ operations. The algorithm is easy to parallelize [18, 74].

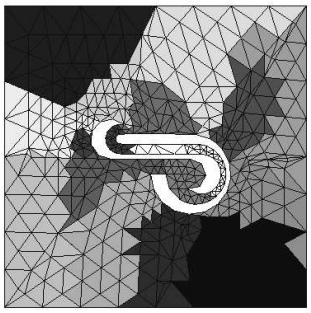

FIG. 3.3. *Partitioning a mesh into 16 subdomains using RGB*

3.2. Graph Theory Based Algorithms. One of the reasons for the relatively low quality of geometric based algorithms is that they make no use of any connectivity information about the graph.

3.2.1. Graph bisection. The *recursive graph bisection* (RGB) algorithm [87] attempts to remedy this. Assume that the graph is connected. The RGB algorithm first finds the two vertices that are the furthest apart (their distance is called the diameter of the graph). Then, starting from one of them (the root), the half of the vertices that are closer to the root form one subdomain, the rest form the other. This process is then recursively executed on each of the subdomains.

To find the two vertices that are the furthest apart, a heuristic procedure can be employed. Starting from any vertex as the potential root R, label its neighbors with 1, and the neighbors' neighbors as 2, and so on. The last labeled vertex has a label of m which is its distance from R. Assign this vertex as R and repeat the process. When the distance m does not increase any more, R is taken as the root and m the diameter of the graph. In practice this procedure converges quickly in a few iterations.

The graph bisection algorithm has a complexity of $O(n)$. Figure 3.3 shows the result of applying the RGB algorithm to the dual graph of the mesh of 788 elements, resulting in 142 shared edges, the same as RCB by coincidence.

3.2.2. Greedy algorithm. Starting with a vertex with the smallest degree, mark its neighbors, and then the neighbors' neighbors. The first n/p marked vertices are taken to form one subdomain and the procedure is applied to the remaining graph until all of the vertices are marked. This algorithm [20] is similar to the RGB algorithm, although it is not a bisection algorithm. Like RGB it has a low complexity of $O(n)$.

3.2.3. Spectral bisection. The *recursive spectral bisection* (RSB) algorithm [68, 75, 87] is based on the following consideration. Let each vertex of the graph be assigned a value of either 1 or -1, and let the value assigned to vertex i be denoted by x_i. This creates a bisection of the graph in which the vertices having value 1 form one subdomain and those with value -1 form the other. The edge-cut for the bisection can be expressed as

$$|E_c| = \frac{1}{4} \sum_{i \leftrightarrow j,\ i,j \in V} (x_i - x_j)^2, \tag{3.2}$$

where $i \leftrightarrow j$ means that there is an edge connecting the vertices i and j. In order to keep the load balanced, each of the two subdomains is required to have the same number of vertices. Thus the sum of all the values associated with the vertices is zero, that is

$$\sum_{i=1}^{n} x_i = 0. \tag{3.3}$$

The graph bisection problem is equivalent to minimizing (3.2) subject to (3.3) with x_i taking the value of either 1 or -1. This is an integer programming problem which is difficult to solve when the number of vertices n is large. Ignoring the integer constraints, the quadratic (3.2) can be minimized subject to the linear constraint (3.3) only. This gives a continuous minimization problem. But without an extra constraint the solution is simply the vector of zeros. Remembering that when all the variables take the value 1 or -1, the sum of the squares should be n, the number of vertices. This gives the extra constraint

$$\sum_{i=1}^{n} x_i^2 = n. \tag{3.4}$$

The quadratic (3.2) can now be minimized subject to (3.3) and (3.4).

The quadratic (3.2) can be written as

$$|E_c| = \frac{1}{4} x^T L x, \tag{3.5}$$

where $x = (x_i)$ is the vector composed of all the values to be assigned to vertices and L is an $n \times n$ matrix known as the Laplacian matrix of the graph. It has the simple form

$$(L)_{ij} = \begin{cases} -1, & \text{if } i \neq j \text{ and } i \leftrightarrow j, \\ deg(i), & \text{if } i = j, \\ 0, & \text{otherwise.} \end{cases} \tag{3.6}$$

Here $\deg(i)$ is the degree of the vertex i, defined in Section 2 as the number of edges connected with the vertex i. The matrix L satisfies $Le = 0$ with e the vector of all ones. Applying the necessary condition for the constrained minimization problem gives

$$Lx = \mu e + \lambda x, \tag{3.7}$$

with μ and λ two Lagrange multipliers to be decided. Multiplying both sides of the equation by e^T gives $\mu = 0$. Thus x has to be an eigenvector of L and the edge-cut is $|E_c| = \frac{1}{4} n \lambda$. In order to minimize the edge-cut, x needs to be the eigenvector of the smallest possible eigenvalue of L that satisfies the two constraints (3.3) and (3.4).

The matrix L is positive semi-definite with smallest eigenvalue $\lambda_1 = 0$ and corresponding eigenvector e. Clearly e is not a solution to the problem since it does not satisfy the load balancing constraint (3.3). If the graph is connected, then it can be shown that the next smallest eigenvalue λ_2 is positive [63]. The eigenvector associated with this eigenvalue satisfies (3.3) because it is orthogonal to the first eigenvector e. It also satisfies (3.4) by proper scaling, and thus is the solution of the constrained

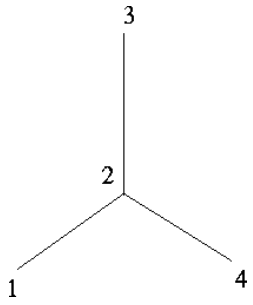

FIG. 3.4. *A graph that can not be bisected into 2 connected subgraphs with an equal number of vertices*

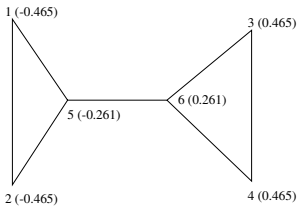

FIG. 3.5. *A simple graph and its Fiedler vector*

minimization problem. This eigenvector (also called the Fiedler vector) gives a value to each vertex of the graph, and the graph can be bisected by separating those with smaller values from those with larger values. The procedure can then be repeated on each of the subdomains. This gives the RSB algorithm. In practice the RSB algorithm almost always gives connected subdomains if the original graph is connected, however there can be exceptions, as Figure 3.4 shows.

Figure 3.5 gives a simple graph and the eigenvector associated with the second smallest eigenvalue of the Laplacian. Here the values of the eigenvector are marked beside each vertex in brackets. Sorting the vertices based on the values of the eigenvector bisects the graph into two triangles, with the edge-cut of one.

To find the eigenvector corresponding to the second smallest eigenvalue, the Lanczos algorithm can be employed. This is an iterative algorithm which requires $O(n)$ operations per iteration. The algorithm normally converges in m iterations, with m typically less than a few hundred. Thus the algorithm has a complexity of $O(mn)$. However the Lanczos algorithm requires the storage of m Lanczos vectors of length n, unless the entire Lanczos recurrence is run through a second time in order to accumulate the eigenvector. An alternative algorithm [56] for finding the eigenvector is to solve the minimization problem (3.2) using the conjugate gradient algorithm, which requires the storage of only four vectors. This algorithm was only tested on regular grids.

Figure 3.6 shows the result of applying the RSB algorithm to the dual graph of the mesh with 788 elements, the partition has 108 shared edges, significantly fewer than the edge-cut of 142 generated by the RCB and RGB algorithms, although the RSB algorithm is far more expensive. In this case all of the subdomains are connected.

Two recursive spectral bisections do not necessarily generate an optimal quadri-

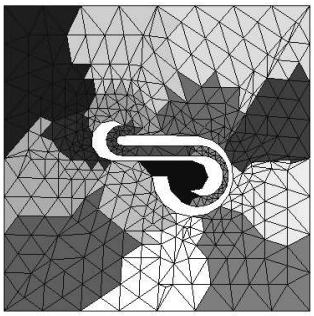

FIG. 3.6. *Partitioning a mesh into 16 subdomains using RSB*

section [76]. Direct quadri-section, even octa-section, using two or three eigenvectors of the Laplacian matrix has been explored [34]. Another interesting way of utilizing multiple eigenvectors is to treat them as spectral coordinates of the vertices. Using these coordinates in the context of inertia bisection can generate partitions of better quality than RSB [77].

3.2.4. K-L algorithm. The K-L (Kernighnan-Lin) algorithm [53, 24, 69] was first suggested in 1970 for bisecting graphs in relation to VLSI layout. It is an iterative algorithm. Starting from a load balanced initial bisection, it first calculates for each vertex the gain in the reduction of edge-cut that may result if that vertex is moved from one partition of the graph to the other. At each inner iteration, it moves the unlocked vertex which has the highest gain, from the partition in surplus (that is, the partition with more vertices) to the partition in deficit. This vertex is then locked and the gains updated. The procedure is repeated even if the highest gain may be negative, until all of the vertices are locked. The last few moves that had negative gains are then undone and the bisection is reverted to the one with the smallest edge-cut so far in this iteration. This completes one outer iteration of the K-L algorithm and the iterative procedure is restarted. Should an outer iteration fail to result in any reductions in the edge-cut or load imbalance, the algorithm is terminated. The initial bisection is generated randomly and for large graphs, the final result is very dependent on the initial choice. The K-L algorithm is a local optimization algorithm, with a limited capability for getting out of local minima by way of allowing moves with negative gain.

By using appropriate data structures, it is possible to implement the K-L algorithm so that each outer iteration has a complexity of $O(|E|)$ [24]. Figure 3.7 shows the best result among 3 runs of the algorithm on the mesh with 788 elements, it has 133 inter boundary edges.

The K-L algorithm is a 2-way local refinement algorithm. It is able to reduce the edge-cut of an existing bisection and can be combined with any of the other partitioning algorithms. It has been extended to p-way refinement (see Section 3.4).

3.3. Other Partitioning Algorithms.

3.3.1. Global optimization algorithms. Since graph partitioning is a global optimization problem, a number of popular global optimization algorithms have been explored. Among these are simulated annealing [87, 61, 79, 62] and genetic algorithms (GA) [60, 59, 10]. These algorithms in their sequential form tend to take significantly more computational time (and more memory in the case of GA) compared to other

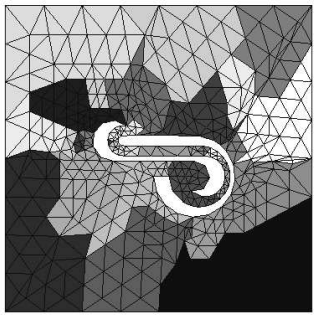

FIG. 3.7. *Partitioning a mesh into 16 subdomains using K-L*

partitioning algorithms. On the other hand they are easy to parallelize and they also have the potential to give partitions of better quality. In particular, these algorithms may be useful in fine tuning an existing partition created by a faster algorithm [79]. Besides, with these algorithms, a more complicated and accurate model, in the form of a cost function that takes into account the effect of communication cost and load balance, is usually used. It is noted that the K-L algorithm is also able to use more complicated cost functions [84]. However in order to retain the efficiency advantage of the algorithm such a cost function should be of a localized nature [84].

3.3.2. Reducing the bandwidth of the matrix. As we have seen in the discussion of the RSB algorithm (equation (3.6)), an undirected graph can be uniquely represented by an $n \times n$ matrix, where the entry (i, j) is nonzero if vertices i and j are connected by an edge and zero if the two vertices are not connected. A good bisection of the graph is equivalent to an ordering, through symmetric permutation, of the matrix such that the resulting matrix has very few nonzero off-diagonal entries corresponding to the two principal diagonal blocks of size $\frac{n}{2} \times \frac{n}{2}$. This is because the off-diagonal entries correspond to the edges between the two subdomains. Although such an ordering algorithm is not readily available, there are a number of algorithms that order the matrix to minimize its bandwidth. A partitioning algorithm [58] was suggested which orders the vertices of the graph using matrix ordering algorithms, and partitions the graph by taking the first n/p vertices as the first subdomain, etc.. The algorithm is inexpensive and can usually result in a smaller number of neighbors for each processor. The edge-cut can be improved if the algorithm is used recursively as a bisection algorithm. Nonetheless in terms of edge-cut its is not as good as algorithms such as the RSB or the greedy algorithm [30].

3.3.3. Index based algorithms. In a way many of the partitioning algorithms mentioned so far work by generating an ordered list of the vertices and bisect this list into equal segments. In the case of spectral bisection for example, this ordering is induced by the value of the Fiedler vector. There are many classic ways of mapping a d dimensional regular grid to a one-dimensional list, such that proximate vertices in the original grid are mapped close to each other on the one-dimensional list. These techniques include the Hilbert space-filling curve [39] and the bit-interleaving transformation [66]. An irregular grid can be so indexed by embedding the grid into a d-dimensional regular grid. The advantage of such index based methods is that the ordering can be calculated very quickly. Index based algorithms can also be parallelized and efficiently adopted for situations where vertices are added or deleted as

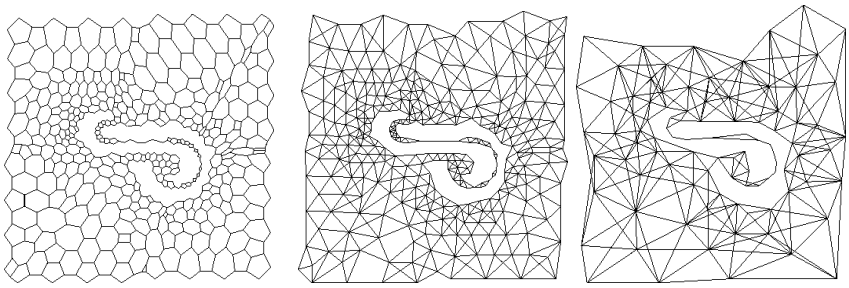

FIG. 3.8. *Graph coarsening based on the independent vertex set: the original Graph G with 788 vertices (left); G_1 with 332 vertices (center); G_2 with 94 vertices (right)*

the result of mesh adaptation. The quality of the partitioning is comparable to that of the RCB algorithm [66].

3.4. The State of the Art. Much of the recent development in the area of grid partitioning has been in the efficient implementation and improvement of existing algorithms. Three important developments are: the combination of existing algorithms; multilevel approaches and parallel implementations of the algorithms.

3.4.1. Hybrid approach. In the hybrid approach, algorithms are combined together to achieve better results. In particular, the K-L algorithm, which is a "steepest decent" type optimization algorithm with a limited capability for getting out of a local minima, has been combined with other more "global" algorithms and with the multilevel approach [42, 79, 35, 36].

3.4.2. Multilevel approach. The idea of the multilevel approach is to encapsulate the connectivity information of a very large graph (of, say, over 1 million vertices) by a series of coarser and coarser graphs. An algorithm based on the multilevel approach normally has 3 phases

- coarsening phase: the original graph G is reduced into a series of successively coarser graphs $G_1, G_2, \ldots, G_k$.
- partitioning phase: partition the coarsest graph G_k into p parts;
- uncoarsening and refinement phase: the partitioning of G_k is interpolated to G_{k-1} and refined, the process repeated and terminated at G.

This approach is in a way similar to the multigrid idea for grid based linear equation solvers [9]. The multilevel approach was first introduced [1] to speedup the recursive spectral bisection (RSB) algorithm, and was soon used in combination with K-L type refinement algorithms [36, 47, 82].

The coarsening phase –

There are a number of ways to coarsen a graph. In [1] the maximal independent set of a graph is chosen as the vertices of the coarse graph. An independent set of the graph G is a set of its vertices, with no two of the vertices in the set connected by an edge of G. An independent set is a maximal independent set if the addition of an extra vertex makes it no longer independent. Figure 3.8 illustrates a graph of 788 vertices (which is the dual graph of the mesh in Figure 3.2), with 2 levels of coarse graphs of 332 and 94 vertices respectively, generated using this method.

The most popular method for generating the multilevel of coarse graphs is based on edge collapsing [36, 47]. Selected edges are collapsed so that two vertices connected by one of these edges form a multi-node. Each vertex of the resulting coarse graph

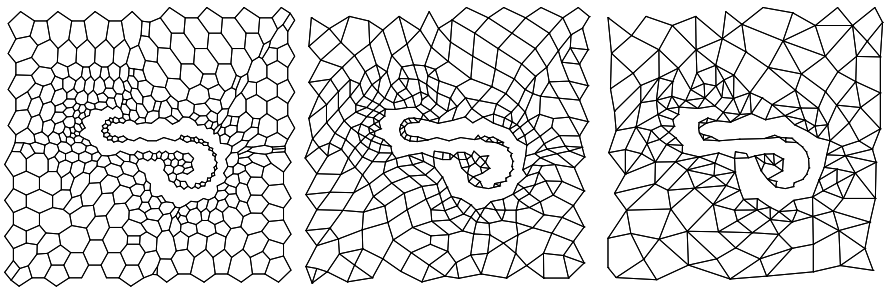

FIG. 3.9. *Graph coarsening based on edge collapsing: the original Graph G with 788 vertices (left); G_1 with 403 vertices (center); G_2 with 210 vertices (right)*

has a weight associated with it, which indicates the number of original vertices it contains. Each edge of the coarse graph also has a weight associated with it that indicates the number of original edges it represents. This edge collapsing approach has the distinctive advantage that the edge-cut (generalized here as the sum of the edge weights cut by the partitioning) on the coarsest graph equals the edge-cut of the finest (original) graph, if the partitioning on the coarsest graph were inherited by the finer graphs without further refinement. Therefore to some extent the original problem is encapsulated well by a problem of much smaller size. Figure 3.9 illustrates the graph of 788 vertices, with 2 levels of coarse graphs of 403 and 210 vertices respectively, generated using this method.

The edges are usually selected using the idea of *matching*. A matching of a graph is a set of edges, such that no two of them share the same vertex. The maximal match is a matching of the largest size. In a *heavy edge matching* [47], vertices are visited randomly. For each vertex, the heaviest unmatched edge from the vertex is selected. Heavy edge matching has the advantage that the resulting coarse graph has a relatively small total edge weight, and therefore the partitioning of it is more likely to give a small edge-cut. As an alternative to edge collapsing based on maximal matching, a more aggressive coarsening approach has been employed [80], where the greedy algorithm (Section 3.2.2) was used to form small clusters of, say, 10 vertices for collapsing. The coarsening process is applied until the graph has less than a preset minimum number of vertices.

The partitioning phase –

As the coarsest graph is of very small size, any partitioning algorithms will be able to partition it rapidly. The partition is executed in a manner such that the summation of the vertex weights of each subgraph should be roughly the same. When edge collapsing approach is used for graph coarsening, the edge-cut of the partitioning is the same as that for the finest graph, should the partitioning be inherited all the way to the finest graph without refinement. The choice of partitioning algorithms at this coarsest level is not very important [47, 36] to the overall quality of the partition because refinement will be carried out at the subsequent finer levels of the graph.

The uncoarsening and refinement phase –

During the uncoarsening and refinement phase, the partitioning of a graph G_{k+1} is inherited by the finer graph G_k.

For the multilevel spectral bisection (MRSB) algorithm [1], the eigenvector for the coarse graph G_{k+1} is interpolated to the graph G_k as the first approximation of the

eigenvector for G_k. From there, instead of using the Lanczos algorithm to recalculate the eigenvector, Rayleigh iterations are used to improve the approximation. The resulting positive semi-definite linear systems are solved using SYMMLQ. Because a good initial approximation exists, the number of matrix-vector products involved in finding the Fiedler vector using the Rayleigh quotient algorithm is normally no more than ten, as opposed to up-to several hundred if the Lanczos algorithm is used (here the matrix refers to the Laplacian matrix of the corresponding graph). Thus the multilevel spectral bisection algorithm is about an order of magnitude faster than the single level Lanczos implementation of RSB.

But there are two disadvantages to the multilevel spectral bisection approach. First, this is still a bisection algorithm and the partitioning of a graph into many subdomains would involve recursive use of the algorithm. Second, even though the Rayleigh quotient procedure is substantially faster than the Lanczos algorithm, each iteration is still of the order $O(n)$, where n is the number of vertices in the graph (but see [81] for a partial improvement of the situation by contracting the interior of each of the two subdomains into super-nodes). However all that may be needed on the finer graph G_k is some refinement of the subdomain boundary to reduce edge-cut and to maintain the load balance. For a large graph with a good existing partition, the size of the boundary should be only a small fraction of that of the interior. Therefore multilevel algorithms that adopt K-L type algorithms during the uncoarsening and refinement phase have proved themselves to be superior to MRSB [36, 82, 47]. The boundary refinement approach is not restricted to bisection and p-way refinement to the existing partition could be used. In [36] vertices of any subdomain are allowed to move to any of the other $p-1$ subdomains, while in [52, 82] boundary vertices are allowed to move to the neighboring subdomains only.

3.4.3. Parallel partitioning algorithms. Although the multilevel approach reduces the computing time significantly, for a very large mesh it can prove to be memory intensive – often exceeding the limits of single CPU memories. Furthermore as the ultimate purpose of partitioning is for the subsequent implementation of the application code on parallel machines, it makes sense to have a parallel partitioning code. Besides, a fast parallel partitioning algorithm can also be used for the purpose of dynamic load balancing. There have been a number of efforts in this area.

Parallel multilevel recursive bisection algorithm PMRSB –

In [2, 3], the multilevel spectral bisection was parallelized specifically for the Cray T3D architecture using the Cray SHMEM facility. The linear algebra algorithms (Lanczos, Rayleigh iteration and SYMMLQ) involved are easy to parallelize. Difficulties arose in the parallel graph coarsening, in particular, in the parallel generation of the maximal independent set. These were tackled by using a number of parallel graph theory algorithms from the literature. On a 256 processor Cray T3D, the resulting algorithm PMRSB is able to partition a graph of 1/4 million vertices into 256 subdomains, 140 times faster than a workstation (of similar specification to one processor of the Cray T3D) using an equivalent serial algorithm.

Parallel multilevel p-way partitioner ParMETIS –

In [49], a parallel multilevel p-way partitioning scheme was developed. This is based on work in the sequential p-way partitioning algorithm METIS [52], but with some interesting modification to facilitate parallelization. In particular, as in PMRSB [2, 3], a parallel algorithm due to Luby [57] for finding the maximal independent set was employed. However unlike MRSB, where the maximal independent set is used

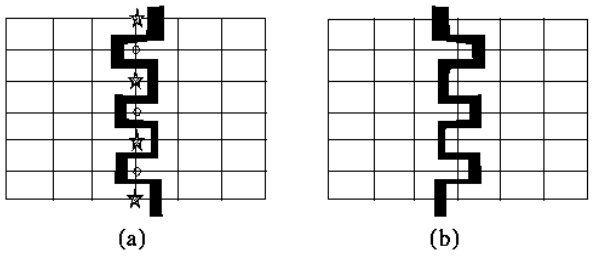

FIG. 3.10. *a) a graph partitioned into 2 subdomains, with subdomain 1 on the left and subdomain 2 on the right; b) after a naive K-L refinement*

directly to form the coarse graph, here the maximal independent set is used for a different purpose. By considering one independent set at one time, it is possible to avoid conflicts during the coarsening stage when the vertices are matched, as well as during the uncoarsening stage when the boundaries are refined. To illustrate this consider the graph of Figure 3.10 (a) distributed across two processors. Suppose one hopes to refine the partition to reduce the edge-cut using K-L type algorithms. To do this in parallel without proper scheduling, processor one finds all the vertices that have a positive gain if moved to processor two (those marked with a star), and sends them to processor two; likewise processor two sends all the vertices marked with a circle to processor one. This results in the graph shown in Figure 3.10 (b) which has the same edge-cut of 13. Any subsequent use of the K-L algorithm in parallel would result in an oscillation between these two graphs. This situation is a result of the fact that moving vertices marked with a star to processor two affects the gains of those marked with a circle, therefore the moves should not be carried out in isolation. One way of avoiding this conflict is for the two processors to communicate with each other and to collaborate on a migration strategy. When there are more than two processors, this may necessitate a coloring of the processor graphs so that processors are grouped in pairs and refinement is carried out on boundaries of the paired processors [18].

An alternative coloring strategy is however used in [49], as it is believed that pairing the processors lacks the global view of the sequential p-way partitioning where each vertex is free to move to the subdomain that leads to the maximum reduction in the edge-cut.

The new strategy is based on coloring the vertices in a number of colors. Vertices of the same color form an independent set. Each colored vertex set is considered one at a time when doing the refinement. By the definition of an independent set (Section 3.4.2), no two vertices in the set are linked by an edge, therefore moving a vertex in this colored independent set to another processor does not affect the gain of other vertices of the same color. Consequently, each processor may decide to send vertices which it identifies as having a positive gain, without the need to communicate with other processors, knowing that such a gain will be realized. This procedure can be repeated on vertices of each color in turn. Using this strategy Figure 3.11 (a) is colored using two symbols (either with or without a Δ). When applying K-L refinement on vertices colored with Δ, Figure 3.11 (a) becomes Figure 3.11 (b), which has a smaller edge-cut of 7, instead of the original edge-cut of 13.

The resulting parallel algorithm was implemented on a Cray T3D using its SHMEM communication facility [49]. Table 3.1 [49] gives a rough idea of the capability of the

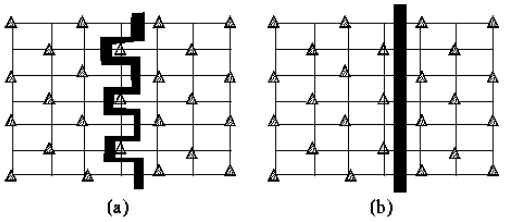

FIG. 3.11. *a) a graph partitioned into 2 subdomains, vertex colored; b) K-L refinement of vertices colored with* Δ

TABLE 3.1
Edge-cut and time for ParMETIS and PMRSB on a Cray T3D.

	598a				MDUAL			
	16 PEs		128 PEs		16 PEs		128 PEs	
	edgecuts	time	edgecuts	time	edgecuts	time	edgecuts	time
PMRSB	37583	67.45	103928	22.478	37235	65.49	54692	18.971
ParMETIS	31211	2.436	90724	0.805	13144	2.637	35457	0.795

algorithm and its performance in relation to PMRSB [3]. The two algorithms were applied on two meshes, 598a (a 3D finite element mesh of 144649 nodes and 1074393 edges) and MDUAL (the dual graph of a 3D finite element mesh, with 258569 vertices and 523132 edges). ParMETIS is significantly faster and gives smaller edge-cut. The scalability of both approaches is similar with approximately three to four fold improvement in performance as the number of processor elements (PEs) is increased by a factor of eight.

A coarse grained modification suitable for MPI implementation was introduced more recently [50]. In this version, the vertex coloring is not used to avoid the need for global point to point communication when each color is considered during the coarsening and refinement phase. Graph folding to part of the parallel processors is also carried at coarse levels to reduce the communication. During the coarsening stage, each processor examines its vertices, and chooses their heavy edge neighbors for matching. A match request of vertex u with a local neighbor v is granted immediately while a request for match with a remote neighbor v is sent to the remote processor for consideration only if $u < v$. During the refinement phase, conflict is avoided by allowing flow in one direction at a time. That is, a vertex u belongs to partition i is considered for moving to partition j only if $i < j$ (or $i > j$ at the next round). Applying this strategy to Figure 3.11 (a), assuming that the flow is from the right to the left, results in Figure 3.11 (b). Using MPI, this version of the ParMetis algorithm was demonstrated [50] to lead to much better performance than that based on vertex coloring.

Parallel multilevel p-way partitioner JOSTLE-MD –

The parallel partitioning algorithm in [83] used a different refinement strategy to that of vertex coloring. As this algorithm is designed also for dynamic load balancing, the initial partitioning is assumed to be unbalanced. For any two neighboring subdomains p and q, the flow (the amount of load to be migrated to achieve global load balance) is first calculated. The flow from p to q is denoted as f_{pq}. Let g_{pq} denote the total weight of the vertices on the boundary of p which have a preference to migrate to q. Let $d = \max(g_{pq} - f_{pq} + g_{qp} - f_{qp}, 0)$, which represents the total weight of all boundary vertices with a positive gain after the flow is satisfied. Then the load to be migrated from p to q is given by

$$a_{pq} = f_{pq} + d/2. \tag{3.8}$$

This allows the flow to be satisfied and at the same time an additional load of equal amount is exchanged between the two processors to optimize the edge-cut.

Having decided on the amount of load to be migrated, the idea of *relative gain* is then used to work out which vertices are to be migrated. The relative gain of a vertex v on processor p is defined as $g(v,q) - \sum_{u \in \Gamma_q(v)} g(u,p)/|\Gamma_q(v)|$, where $g(v,q)$ denotes the gain of moving v to processor q, and $\Gamma_q(v)$ those vertices in processor q that are connected with v. The vertices on the boundaries are sorted by their relative gain and a weight of a_{pq} is migrated. This avoids the collisions that arise when the gain of vertices varies. But in a situation when each boundary vertex has exactly the same gain, a tie break strategy is necessary. This was not detailed in [83]. The algorithm has been shown to be very efficient in partitioning and re-balancing the meshes resulting from adaptive refinement. The parallel JOSTLE algorithm has been compared with ParMETIS. It was found to give partitioning of better quality, although it runs slower than ParMETIS [85].

Other parallel graph partitioning algorithms –

In [18] a parallel single level algorithm, combining inertia bisection with K-L refinement, was implemented. Possible conflict during the refinement stage was avoided by the pairing of processors based on the edge-coloring of the processor graph. The quality of the partition was not as good as multilevel algorithms.

In [77] a spectral inertia bisection algorithm was introduced. The spectral basis set of eigenvectors for the coarsest graph was first calculated and serves as the spectral coordinates of the vertices. The graph was partitioned with the inertia bisection algorithm (Section 3.1.2), based on these spectral coordinates. Part of the algorithm was parallelized. This algorithm is also suitable for dynamic load balancing, on applications where the mesh is enriched by refining the individual elements. In such a case the refinement can be captured by updating the vertex weights of the dual graph of the mesh, without changing the graph itself. The mesh is repartitioned quickly after refinement, using the spectral information originally calculated for the top level coarse mesh. Since the size of the dual graph does not change, the repartitioning time does not change with the increase of the mesh size, as the refinement steps are carried out. However a possible disadvantage of this approach is that in order to reuse the spectral information, only the vertex weights, not the edge weights, are allowed to be updated. Thus the edge-cut may not be a good representation of the actual edge-cut of the mesh after many levels of refinement.

3.5. Existing Partitioning Softwares. In choosing a partitioning algorithm, it is necessary to balance the benefit of the reduction in communication cost, against

that of the cost of generating the partitioning. If the parallel computer has slow communication, or the application is fine-grained where there is no large chunk of computation to "shield" the cost of communication, it is very important to have a partition of good quality. On the other hand, if the parallel computer has very fast communication, or the application is coarse grained where there is a lot of computation in between communication, a big reduction of the communication cost will only result in a fractional reduction of the overall cost, and in such a case a simple and cheap partitioning algorithm, such as the greedy algorithm, is quite adequate. Some of the most recent parallel partitioning codes [49, 83] give very good quality partitions. They are scalable in memory and yet are comparable in cost to the very simple algorithms such as the greedy algorithm.

A number of packages are publicly available. Many of them are constantly being updated. Further details should be checked from the contact addresses listed.

- **CHACO** (*http://www.cs.sandia.gov/CRF/chac.html*)
  perhaps the first publicly available package. The latest version CHACO 2.0 uses multilevel combined with a number of available methods such as K-L refinement, spectral bisection. Available by request.
- **JOSTLE** (*http://www.gre.ac.uk/jostle/*)
  multilevel graph partitioning using p-way refinement. Both the sequential and parallel version are available free for academic use as an executable, after signing an agreement.
- **METIS** (*http://www.cs.umn.edu/~karypis/metis/metis.html*)
  multilevel graph partitioning using p-way refinement. Both the sequential and the parallel version are available free for down-load.
- **PARTY** (*http://www.uni-paderborn.de/fachbereich/AG/monien/RESEARCH/PART/party.html*)
  sits on top of existing packages, offers greedy, multilevel spectral bisection, coordinate bisection, inertia bisection and other algorithms. Version 1.1 is available after signing a license.
- **PMRSB** (*stb@renaissance.cray.com*)
  parallel multilevel spectral bisection. Available only on CRAY platforms.
- **S-HARP** [77] (*http://www.cs.njit.edu/sohn/sharp/*)
  multilevel graph partitioning using multiple eigenvectors and (spectral) inertia bisection on the coarsest level.
- **SCOTCH** (*http://www.labri.u-bordeaux.fr/Equipe/ALiENor/membre/pelegrin/scotch/*)
  multilevel implementation of a number of algorithms including K-L. The code is free for down-load for academic use.
- **TOP/DOMDEC** [21] (*charbel@boulder.colorado.edu*)
  Greedy algorithm, recursive coordinate and graph bisection, recursive inertia bisection, multilevel spectral bisection algorithm with tabu search, simulated annealing and stochastic evolution as smoothers.
- **WGPP** [31, 32]
  multilevel graph partitioning. Similar to METIS with some new features.

4. Dynamic Load Balancing Algorithms. When adaptive algorithms are used, after an interval of computation, the mesh may be refined (or coarsened) at some locations, usually based on an estimate of the discretization error. The refinement (or coarsening) process can generate widely varying numbers of mesh nodes on the processors. Subsequently, there is a need for dynamic load balancing. Load imbalance

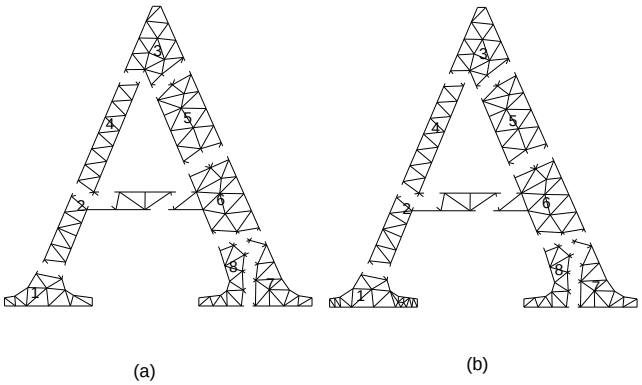

FIG. 4.1. *(a) A mesh of "A" shape partitioned into 8 subdomains; (b) the mesh refined at subdomain 1*

may also be caused by the use of local time stepping, local spatial approximation schemes of varying orders [26], or non-linear material properties.

As a simple example, Figure 4.1 (a) shows a mesh of shape "A", partitioned into 8 subdomains. It has been refined in Figure 4.1 (b). Due to the mesh refinement subdomain 1 has more nodes than the other subdomains.

In general dynamic load balancing algorithms should satisfy the following objectives

- Re-balance the load of each processor with speed and scalability.
- Minimize the edge-cut (or more precisely, the communication cost of the application after the re-balance)

In order to satisfy the first objective, the dynamic load balancing algorithm should not only identify what to migrate efficiently, the amount of data required to be migrated should also be kept to a minimum. Various metrics such as TotalV and MaxV [70] have been used to model and minimize the data migration cost.

One way to re-balance the load is to *repartition* the mesh using one of the partitioning algorithms of Section 3. Indeed parallel algorithms such as JOSTLE or ParMETIS are able to partition large mesh very rapidly [83, 70]. For example, ParMETIS was able to partition a mesh of the order of 1 million nodes in less than 2 seconds on 128 PEs of a Cray T3D [70]. However it is important, but difficult, to ensure that the new partitioning will be "close" to the original partitioning. Should the new partitioning deviate considerably from the old one then the cost of transferring large amounts of data will be incurred. It has been found that repartitioning is more appropriate when there has been a substantial localized refinement on the mesh [72, 5].

An alternative strategy is to *migrate* the excessive nodes to neighboring processors, effectively shifting the boundaries to achieve a balanced load. This approach may potentially cause less movement of data than repartitioning, although the edge-cut after the migration could possibly be larger than that given by a global repartitioning. Therefore care must be taken to keep edge-cut down when choosing the nodes to be migrated. It has been found [72] that this strategy is more suitable when the load imbalances caused by the refinement are low, or when localized high imbalances

occur throughout the mesh. This is because in such cases the optimal partition will be relatively close to the initial partition.

4.1. Dynamic Load Balancing by Repartitioning. Dynamic load balancing through repartitioning is relatively straightforward. Any of the fast parallel graph partitioning algorithms of Section 3 may be employed. These include the parallel multilevel schemes. Algorithms that allow quick reordering of a refined or coarsened mesh through a slight shift of the original ordering, such as the index based algorithms of Section 3.3.3, can also be attractive due to their simplicity and data locality. For adaptive mesh based applications where the relationship between the "parents" and "offspring" elements is known, the octree partitioning approach [25, 26] offers an efficient way of repartitioning. The tree representation (the octree) is used to form a one-dimensional list of elements, through a depth-first search. This list can be divided, in parallel, into segments to give nearly equal load. Members of the same segment tend to be spatially close to each other (because they are likely to have the same parent or parents close to each other), and thus form a good partition. This partition can be further smoothed [25] by migrating boundary elements based on a K-L like gain calculation. Incidentally a similar idea was used in the parallel generation of partitioned unstructured meshes [40], where a coarse background mesh was partitioned.

Standard graph partitioning algorithms however only have edge-cut as the sole objective. To minimize the data movement resulting from the repartitioning, a number of techniques have been used to modify the graph partitioning algorithms, so that the new partition is as close to an existing partitioning as possible. The idea of a virtual vertex was used in [81, 19, 37]. A virtual vertex is associated with each subdomain and is connected to each vertex in the subdomain by a virtual edge. The weight of this edge reflects the communication cost of migrating the vertex. By partitioning the combined graph, the dual objectives of minimizing edge-cut and data movement are considered at the same time. In [73], the idea of local matching was used where matchings were restricted to vertices that have the same home processor. This strategy biases the multilevel graph partitioner towards an existing partitioning, thus reducing data movement resulting from the repartitioning.

A re-mapping algorithm can be applied after the repartitioning to allocate the subdomains to the most appropriate processors, in order to reduce the data movement [5, 65, 72].

4.2. Dynamic Load Balancing Through Node Migration. The strategy of migrating boundary nodes to achieve load balancing is more complicated and is therefore discussed in detail in the following.

The process of migrating loads between processors to achieve load balance can be broken down into two distinctive steps [81, 82, 83]:

- *Flow calculation:* Each processor works out a schedule for the amount of load that should be sent to (or received from) its neighboring processors. This is referred to as a flow calculation in [83].
- *Node selection:* Once the flow is worked out, each processor decides which mesh nodes should be sent or received, to satisfy the flow as well as to minimize the edge-cut.

For real applications, there is also the third step to carry out the actual migration of the meshes and fields. This step is very complex and the exact details depend on the applications and the data structures used. In many adaptive computations, the amount of data associated with each mesh node may be quite large. The time for

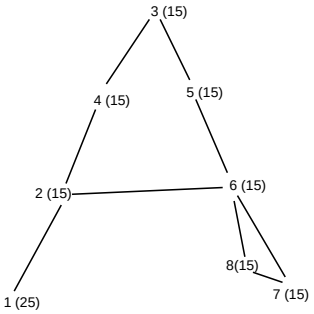

FIG. 4.2. *processor graph associated with the partitioned mesh in Figure 4.1 (b), and the load (in brackets) on each processor*

the migration of the data, and subsequent book-keeping cost of updating the data structures, can dominate overall run time, specially if there is a need to re-balance the load frequently. So far there exist few large scale applications that have a full dynamic load balancing capability (but see, e.g., [26]). We shall therefore concentrate on each of these first two steps, which are less application dependent.

4.3. Flow Calculation. The problem can be formally stated as follows:

Definition Given a graph $G = (V, E)$, let each vertex $i \in V$ have a load l_i associated with it. The *flow calculation* problem is to find the amount of migrating load δ_e along each edge $e \in E$, such that after the migration, the load on each vertex is the same. That is

$$l_i + \sum_{j \leftrightarrow i} \delta_{ji} = \bar{l}, \quad i \in V. \tag{4.1}$$

Here $\bar{l}$ is the average load over all vertices, defined as $\bar{l} = (\sum_{i \in V} l_i)/|V|$, and δ_{ji} is the amount of load to be migrated from node j to node i. □

The graph here is the so called *processor graph* of the partitioning. Each vertex of the processor graph corresponds to a subdomain (or a processor) and two vertices are linked by an edge if the two subdomains share a boundary. Figure 4.2 shows the processor graph corresponding to the partitioning in Figure 4.1 (b). The load l_i can simply be taken as the number of nodes in the subdomain i. Figure 4.2 shows that subdomain 1 has a load of 25 nodes, whilst others have 15 nodes each. In the following the terms *vertex* and *processor* are used interchangeably.

It is noted that the flow calculation problem does not have a unique answer. This is because the solutions are given by a linear system (4.1), with δ_{ij} as the unknown. There are $|V|$ equations (out of these, $|V| - 1$ are independent), and $|E|$ unknowns. As there are usually more edges than vertices in a graph (that is, $|E| > |V|$), there are usually more unknowns than equations and so the answer to the problem is not unique.

This flow calculation problem has been studied by many authors [4, 6, 14, 28, 43, 44, 41, 78, 88, 89]. Apart from parallel unstructured mesh based applications, it has appeared in areas such as parallel molecular dynamic simulation [7, 55]. Some of the algorithms are discussed in the following.

4.3.1. Diffusion algorithm. One of the most popular approaches to the flow calculation problem is to use diffusion based algorithms [6, 14]. In a heat diffusion process, the initial uneven temperature distribution in space causes the movement of heat, and the system eventually reaches a steady-state temperature.

The diffusion algorithm, as described in [6], is given as follows. At each iteration $k+1$ of the algorithm, processor i will send an amount proportional to the difference between its load and its neighbor's load, $c_{ij}(l_i^{(k)} - l_j^{(k)})$, to its neighbor j. Assume $c_{ij} = c_{ji}$, the new load $l_i^{(k+1)}$ of the processor i is given by the combination of its own load $l_i^{(k)}$ and contributions from/to its neighboring vertices, namely

$$l_i^{(k+1)} = l_i^{(k)} - \sum_{i \leftrightarrow j} c_{ij}(l_i^{(k)} - l_j^{(k)}), \quad i,\ j \in V,\ k = 1, 2, \ldots \tag{4.2}$$

Initially the load for vertex $i \in V$ is $l_i^{(1)} = l_i$. In matrix form, the above equation can be rewritten as

$$l^{(k+1)} = (I - A\ W\ A^T)\ l^{(k)}, \tag{4.3}$$

where W is a diagonal matrix of the size $|E| \times |E|$, that consists of the coefficients c_{ij}, and A is a matrix of size $|E| \times |V|$, to be defined in Section 4.3.4. For the choice of the coefficients, Boillat [6] suggested

$$c_{ij} = \frac{1}{\max\{deg(i), deg(j)\} + 1}, \quad i \leftrightarrow j, \quad i,\ j \in V.$$

The diffusion algorithm, being a stationary iterative method of the form (4.3), can converge quite slowly on graphs with small connectivity. Boillat [6] proved that the worst case happens when the graph is, say, a line, and in such a case the number of iterations needed to reach a given tolerance is $O(p^2)$, with p the number of vertices. There are other variations of the diffusion algorithm (see [33, 28]). For example instead of (4.3), a special case of the following equation,

$$(I + A\ W\ A^T)\ l^{(k+1)} = l^{(k)},$$

is solved in [33]. The convergence of the diffusion algorithm can also be improved using the Chebyshev polynomial [44, 17, 64].

Many investigations of dynamic load balancing algorithms have used a diffusive approach, although the details vary. For example, in both the tiling algorithm and the iterative tree balancing algorithm [13, 25, 26], a processor selects amongst its neighbors the one with the highest load and posts a request. In the tiling algorithm the amount of load to be sent is decided by looking at the average of the loads in the neighborhood. In the iterative tree balancing algorithm the requests are viewed as a forest of trees. The flow along the branches of the tree is then calculated using a logarithmic time parallel scan operation.

4.3.2. Dimension exchange algorithm. Cybenko [14] suggested a dimension exchange algorithm, in which the edges of the graph are colored so that no two edges of the same color share a vertex. Pairs of processors having the same color were grouped and a processor pair $(i,\ j)$ with load l_i and l_j exchange their load, after which each has the load $(l_i + l_j)/2$. The algorithm was proved to converge in d steps if the graph considered was a hypercube with dimension d. Xu and Lau [88, 89]

extended the dimension exchange algorithm so that after the exchange processor i has load $l_i * a + l_j * (1 - a)$. If $a = 0.5$ this is equivalent to Cybenko's algorithm. Based on an eigenvalue analysis of the underlining iterative matrices, they argued that for some graph a factor a other than 0.5 gives better convergence. On a graph with small connectivity, this algorithm suffers in convergence in the same way as the diffusion algorithm.

4.3.3. Multilevel algorithm. To speedup the diffusion algorithm, Horton [41] suggested a multilevel diffusion method. The processor graph was bisected and the load imbalance between the two subgraphs was determined and transferred. This process was repeated recursively until the subgraphs could not be bisected any more. The advantage of the algorithm is that it is guaranteed to converge in $log(p)$ bisections, and the final load will be almost exactly balanced even if the loads are integers. However, because it is not always possible to bisect a connected graph into two connected subgraphs, it was not clear from the paper how to proceed for such a case. Connectivity can of course be restored by adding new edges to a disconnected subgraph. However this is equivalent to moving data between non-neighboring processors and should be avoided.

4.3.4. The method of potential. As mentioned before, the solution of the system of linear equations (4.1) is not unique. The motivation of the algorithm [43], called the method of potentials here for reasons explained later, is to choose amongst these solutions one that minimizes the data movement. For this section (Section 4.3.4), the processor graph is assumed to be *directed*, with the direction of each edge going from the vertex with higher index to the one with lower index.

Let A be the matrix associated with the linear system (4.1), let x be the vector of δ_{ij}'s and b the right hand side of (4.1). Assuming that the Euclidean norm of the data movement is used as a measure of data movement, and that the unit communication cost between any two processors i and j is $1/(c_{ij})^{1/2}$. Let W denote the diagonal matrix of size $|E| \times |E|$ consisting of the c_{ij}, then one needs to find a solution x which solves the following minimization problem:

$$\begin{aligned} &\text{Minimise } \frac{1}{2} x^T W^{-1} x, \\ &\text{subject to } \quad Ax = b. \end{aligned} \tag{4.4}$$

Here A is the $|V| \times |E|$ matrix associated with system (4.1). It is called the vertex-edge incident matrix [68, 8] of the directed processor graph, given by

$$(A)_{ik} = \begin{cases} 1, & \text{if vertex } i \text{ is the head of edge } k, \\ -1, & \text{if vertex } i \text{ is the tail of edge } k, \\ 0, & \text{otherwise.} \end{cases}$$

Applying the necessary condition for the constrained optimization problem (see [27]), after some algebraic manipulation, one has

$$x = W A^T d, \tag{4.5}$$

where d is the vector of Lagrange multipliers given by the equation

$$L\, d \;=\; b, \tag{4.6}$$

with $L = AWA^T$.

Thus the problem of finding an optimal load balancing schedule is transformed to that of solving the linear equation (4.6). Once the Lagrange multipliers are found, then the load transfer vector is $x = WA^T d$.

For any graph, each row k of the matrix A^T only has two non-zeros, 1 and -1, corresponding to the head and tail vertices of the edge k. Therefore the amount of load to be transferred from processor i to processor j (assuming i is the head and j the tail), along the edge $e = (i, j)$, is simply

$$\delta_{ij} = c_{ij}(d_i - d_j),$$

where d_i and d_j are the Lagrange multipliers associated with vertices i and j respectively. The quantities d_i and d_j can also be looked upon as the potentials at the vertices i and j and their (weighted) difference gives the flow between the two vertices. That is why we call this method *the method of potentials*.

The matrix $L = AWA^T$ is a generalized form of the Laplacian matrix (3.6). For many parallel computers the unit communication cost is roughly the same between any two processors ($W = I$), in which case this matrix is the same as (3.6).

The linear system (4.6) can be solved by many standard numerical algorithms. The conjugate gradient algorithm was used in [29] because it is simple, easy to parallelize and converges quickly. For preconditioning, the diagonal of the Laplacian may be used.

As a simple example, considering the processor graph in Figure 4.2. The load for each processor is given in brackets. The average load is 16.25 and the largest load imbalance is $(25 - 16.25)/16.25 = 53.8\%$. The Laplacian system (assuming a weighting of $W = I$) is now

$$\begin{pmatrix} 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 3 & 0 & -1 & 0 & -1 & 0 & 0 \\ 0 & 0 & 2 & -1 & -1 & 0 & 0 & 0 \\ 0 & -1 & -1 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 2 & -1 & 0 & 0 \\ 0 & -1 & 0 & 0 & -1 & 4 & -1 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & -1 & 2 \end{pmatrix} \begin{pmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_5 \\ d_6 \\ d_7 \\ d_8 \end{pmatrix} = \begin{pmatrix} 25 - 16.25 \\ 15 - 16.25 \\ 15 - 16.25 \\ 15 - 16.25 \\ 15 - 16.25 \\ 15 - 16.25 \\ 15 - 16.25 \\ 15 - 16.25 \end{pmatrix} = \begin{pmatrix} 8.25 \\ -1.25 \\ -1.25 \\ -1.25 \\ -1.25 \\ -1.25 \\ -1.25 \\ -1.25 \end{pmatrix}.$$

The solution of this linear equation is

$$(d_1, \ldots, d_8) = (11.28, 2.53, -2.22, -0.47, -2.72, -1.97, -3.22, -3.22).$$

These Lagrange multipliers (the potentials) are illustrated in Figure 4.3 in brackets. The amount of load to be transferred between two neighboring processors is the difference between their potentials, and is shown along the edges in Figure 4.3. For example, processor 1 needs to send to processor 2 a load of $11.28 - 2.53 \approx 9$.

Table 4.1 gives the result of the method of potentials and that of the diffusion algorithm on random graphs of varying connectivity. The algorithms have been implemented on a Cray T3D with MPI. It can be seen that in general the method of potentials is faster, and that for graphs of small connectivity it is considerably faster. The same conclusion can be drawn when applying the two algorithms on processor graphs from real applications [43].

When examining the Euclidean norm of migrating flow (Figure 4.4), it was surprising to find that the norm for the diffusion algorithm was usually as small as that

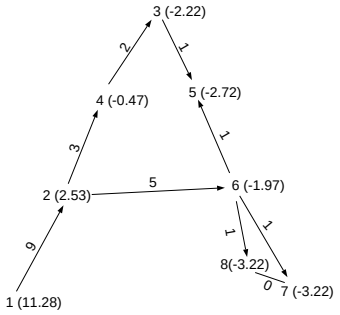

FIG. 4.3. *The potentials and the amount to be migrated along each edge*

TABLE 4.1
The number of iterations (and in brackets, time for convergence, in milli-seconds) for the algorithm of potentials and the diffusion algorithm, on randomly generated graphs

p	diameter	degree	potential	diffusion
64	9	3	25 (19.3)	177 (51.8)
64	5	5	15 (12.5)	57 (21.3)
64	4	7	11 (10.5)	38 (17.6)
64	3	9	8 (9.1)	23 (13.0)
128	87	2	116 (86.7)	11507 (2915.5)
128	11	3	27 (22.9)	168 (55.8)
128	6	5	15 (14.5)	65 (27.0)
128	5	7	13 (13.4)	43 (22.4)
128	4	9	10 (11.4)	25 (16.1)
256	155	2	223 (182.1)	32243 (8624.0)
256	14	3	34 (31.0)	155 (53.6)
256	7	5	19 (19.0)	65 (28.3)
256	6	7	15 (17.2)	48 (26.4)
256	4	9	12 (15.2)	45 (28.1)

for the method of potentials, and much smaller than that of algorithms such as the dimension exchange algorithm. It has now been proved [44] that diffusion type algorithms also have a minimal norm property. In fact it is possible to prove a rather general result [45] that if a flow calculation algorithm is based on the polynomial of the weighted Laplacian, $l^{(k+1)} = p_k(L)l^{(1)}$, where p_k is a k-th order polynomial such that $p_k(0) = 1$, then the flow generated, provided that the algorithm converges, solves (4.4). In other words, all polynomial based flow calculation algorithms, including the diffusion algorithm (4.3), generate exactly the same minimal flow as the algorithm of potentials, provided that the same edge weights c_{ij}'s are used. In terms of the rate of

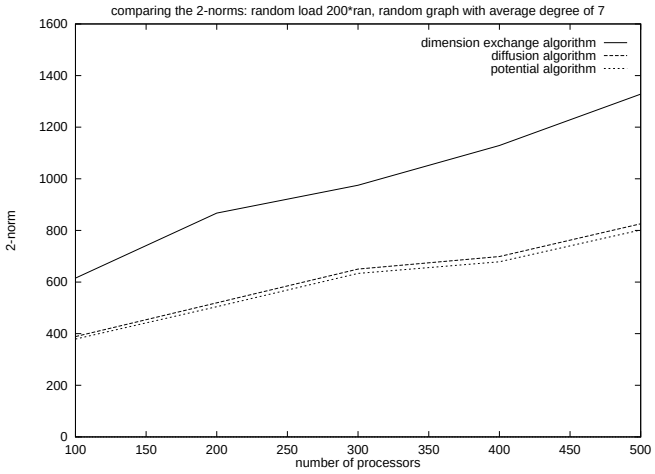

FIG. 4.4. *Comparing the Euclidean norm of the flow from three algorithms: dimension exchange algorithm (top solid line), diffusion algorithm (middle broken line) and the algorithm of potentials (bottom broken line)*

convergence, the algorithm of potentials is of course better. It also converges for any positive edge weights.

4.3.5. Linear programming based algorithms. A possibly better model [43] of the communication cost in the migration process, as opposed to (4.4), is the maximum cost of load migration over all processors, that is

$$\text{cost} = \max_{i \in E} \ (t_0 + \alpha \ |x_i|).$$

Here t_0 is the communication latency and α is the subsequent cost of communication per word. The flow calculation problem then becomes

$$\begin{aligned} &\text{Minimise} \quad \{\max_{i \in E} \ (t_0 + \alpha \ |x_i|)\} \ , \\ &\text{subject to} \quad Ax = b, \end{aligned}$$

which is equivalent to

$$\begin{aligned} &\text{Minimise} \quad c, \\ &\text{subject to} \quad Ax = b, \ c \geq (t_0 + \alpha \ |x_i|), \ i \in E. \end{aligned} \tag{4.7}$$

However it is not clear that an efficient parallel algorithm exists for such a linear programming problem.

Similar linear programming based flow calculation were proposed in [67], where the problem

$$\begin{aligned} &\text{Minimise} \quad \sum_{i \leftrightarrow j} \delta_{ij} \\ &\text{subject to} \ \sum_{i \leftrightarrow j} (\delta_{ij} - \delta_{ji}) = l_j - \bar{l}, \quad j \in V \\ &\qquad 0 \leq \delta_{ij} \leq \alpha_{ij}, \quad i, j \in V; \ i \leftrightarrow j \end{aligned}$$

was solved. Here α_{ij} is the number of vertices on subdomain i that may be moved to subdomain j, using a node selection strategy based on layering (see the next section). This linear programming problem was solved using the simplex method to give the flow. The problem has $2|E|$ variables and $|V| + |E|$ constraints. A multilevel approach was used to group subdomains into super-partitions, thereby breaking the linear programming problem into smaller ones to be solved by subsets of processors. This reduced the overall complexity of solving the linear programming problem.

4.4. Node Selection for Migration. Once the flow is calculated, it is necessary to identify the nodes to be migrated. In doing so, it is essential that the edge-cut as well as the amount of data migration are minimized in addition to restoring the balance, and this process has to be in parallel.

In [83] this was achieved by using formula (3.8) to decide the amount of load to be moved, taking into account the minimization of edge-cut and load balancing. Nodes are selected using the idea of relative gain (Section 3.4.3). Results in [83] on an adaptive finite element problem with a mesh of 31172 nodes refined in 8 steps to 224843 nodes showed that this approach was successful. On the Cray T3D, the time for each dynamic load balancing associated with each refinement was less than a second for 64 processors. Furthermore, the edge-cut was competitive with those produced by repartitioning the meshes. The most telling fact of all was that the number of nodes migrated was as little as a few percent of the total number of nodes, compared with near 100% for repartitioning.

In [70, 71], the parallel METIS algorithm [50] was extended to add the capability of dynamic load balancing. One of the differences compared to JOSTLE-MD is that dynamic load balancing was not applied on the original graph. The proposed algorithm took a two stage approach – the multilevel diffusion stage, followed by the multilevel refinement stage. When applied to a partitioned mesh with unbalanced load, the algorithm first generated a multilevel of meshes, one coarser than the other. Dynamic load balancing was then carried out on the coarse meshes, using either "directed diffusion" or "undirected diffusion". The former used the algorithm of potentials of Section 4.3.4 to calculate the flow, while the latter was close to the classical diffusion algorithm of Section 4.3.1. Boundary vertices were visited in a random order and moved to the neighbor to achieve load balance. After the load was balanced, multilevel refinement started and the graph was refined. Boundary vertices were again visited randomly and checked to see if migration of them to a neighboring processor would result in reduced deviation from the original partitioning; reduced edge-cut or improved load balance. The algorithm was compared against JOSTLE-MD and was shown to be very competitive.

In [67] a layer process was used for node selection. Each boundary node is assigned a label equal to the partition that has the maximum number of nodes that are adjacent to this node. Nodes that are adjacent to the boundary nodes are then marked based on the labels of the boundary nodes. The process is recursively carried out until all of the nodes are marked.

In [13, 25, 26], element selection for the so called tiling algorithm and the iterative tree balancing algorithm is based on the idea of priority, similar to the concept of gain used in the K-L algorithm. Elements are assigned priorities (initially zero) based on the locality of their neighbors. An element's priority is decreased by one for each neighbor in its own processor, increased by two for each neighbor in an importing processor, and decreased by two for each neighbor in any other processor.

An important aspect of the node selection process is that nodes need to be selected

to minimize the data migration in addition to minimizing the edge-cut. In [70], the *size* and *weight* of a vertex are defined as the cost of migrating this vertex and the work load of this vertex, respectively. A vertex is *dirty* if it is displaced (no longer on its original subdomain) in the process of balancing. Two heuristics for reducing the TotalV (the total sum of the size of dirty vertices after load balancing) and MaxV (maximum of the sums of the sizes of vertices which migrate into or out of any one partition after load balancing) were proposed. This is necessary partly because in ParMETIS, load balancing was carried out on coarse graphs, where each vertex represents a varying number of original vertices of the graph. Nonetheless the technique can be useful for applications where the mesh nodes have heterogeneous loads due to, say, the use of the local time stepping or local spatial approximation schemes.

The first heuristic was proposed to minimize MaxV. It was applied in the multilevel diffusion stage and involved a *suppression factor*. The density of the vertex is its weight divided by its size. What flow calculations give is the amount of load (the sum of the vertex weights) that need to be migrated to restore the load balance. Clearly to satisfy a given flow, it is cheaper to send vertices of higher density. Thus it was proposed [70] that only vertices with a density higher than the average vertex density of the graph multiplied by the suppression factor qualified for migration. The second heuristic was proposed to reduce TotalV. Here priority was given to the migration of dirty vertices by the use of a *cleanness factor* in the multilevel refinement stage. Only those clean vertices whose gain is greater than their size times the cleanness factor were considered for migration. The use of either heuristic may increase the edge-cut [70], nonetheless it was possible to improve both TotalV and MaxV with only a slight deterioration of edge-cut.

5. Future Research Topics. Algorithms for load balancing of unstructured mesh based applications have seen rapid development in recent years. However, some problems remain.

Although packages such as JOSTLE and ParMETIS can partition and load balance large meshes very rapidly [83, 70], the scalability of these algorithms remains a concern, particularly for applications where the load on each processor changes frequently. As pointed out in [83], these algorithms use only integer operations, the number of which is proportional to the size of the subdomain boundary, with no "chunky" floating point operations to "hide behind". Also they may not be load balanced and the communication cost may dominate. These factors make it difficult to achieve high scalability.

Almost all algorithms use edge-cut as the cost function for minimization. But this is not necessarily appropriate for all applications. There are a number of limitations of edge-cut based cost functions [38, 12]. First of all, edge-cut is not necessarily an accurate measure of the communication time. Factors such as the number of neighbors, the size of the largest inter-processor boundary among all subdomains, as well as the start up cost of the communications all come into play. Second, for some applications, there may be more than one objective that needs to be minimized.

For applications where domain decomposition type solvers or preconditioners are used, factors such as aspect ratio of the subdomains affect the convergence, through the interface problem [22, 23]. There have been some studies of special partitioning algorithms that take this into account [16, 84]. In [16], the reduction of the aspect ratio of subdomains was achieved through the migration of vertices that were far away from the geometric center of the subdomain. In [84], the aspect ratio was simplified to a localized function of the interface surface areas, which was associated with the edges

of the dual graph. The problem of minimizing the aspect ratio of the subdomains was transformed to that of minimizing the sum of edge weights cut by the partitioning, which can be dealt with in a similar way to minimizing the edge-cut. In both studies the minimization of aspect ratio is balanced against the minimization of edge-cut so that the communication cost resulting from the partitioning will be kept in check. It remains to be seen that these aspect ratio based partitioning algorithms do indeed improve the convergence of the solvers.

In other applications such as multi-physics calculations involving solids and liquids, the computation is carried out in phases and the mesh that is partitioned to balance the load of one phase of computation does not necessarily balance the other phases. Conventional graph partitioning algorithms can be viewed as optimization algorithms that minimize the edge-cut, subject to the load balancing constraint. For multi-physics calculations there is a need to accommodate two or more load balancing constraints. In [51] this multi-constraints optimization problem was looked at and a procedure suggested to solve the problem heuristically. This involved the modification of the different stages of the multi-level K-L like algorithm. In particular, in the graph coarsening stage the criterion for matching vertices was a combination of heavy edge weights and a balanced sum of vertices weights, assuming that each vertex has a vector of weights associated with it that represents the amount of work on this vertex during different phases of the computation. For the refinement stage the K-L algorithm was modified to take into account the need for minimal edge-cut and to satisfy the load balancing constraints, by favoring the migration of vertices that will improve the edge-cut as well as the load balance of the partitioning. The natural extension of the above is to multi-objective optimization problem, where instead of minimizing the edge-cut, other objectives such as aspect ratio and load balancing for multi-phase computation are included.

The limitations of the edge-cut based model for graph partitioning and efforts to overcome them have resulted in modifications of some of the more popular software packages. For example in the recent version of METIS (Version 4.01), a number of new features have been added. These include the ability to handle multi-constraints, the minimization of the total number of boundary vertices and the minimization of the maximum connectivity of subdomains.

Apart from modifications of the standard undirected graph based algorithms, alternative graph models have also been suggested to represent the underlying problems more accurately. A multilevel algorithm based on hypergraphs has been suggested to minimize communication involved in symmetric and unsymmetric matrix vector multiplications on parallel platforms [12], while bipartite graphs have been used as a better model for the problem of minimizing communication cost in parallel iterative algorithms and preconditioners for unsymmetric systems [38].

The actual migration step of the dynamic load balancing process also deserves further investigation. Few large scale applications that have full dynamic load balancing capability exist (but see, e.g., [26])), due to the difficulties involved with regards to the data structures and book-keeping, in actually moving entities such as meshes, fields and other physical quantities around the processors. During the dynamic load balancing process, the mesh on each processor will have to be split and part of the mesh, together with the field, will need to be sent to the appropriate neighboring processors. At the same time each processor will receive meshes and fields and combine them with what is left to form a single entity. There can be a lot of book keeping involved and such tasks are best fulfilled with a code written in an object oriented

manner (see, e.g., [86]).

6. Conclusions. Algorithms for load balancing of unstructured application has reached a degree of maturity. The static load balancing problem (mesh partitioning) can be deemed as solved. A number of fast and high quality parallel mesh partitioning codes are now available. Flow calculation and the node selection part of the dynamic load balancing problem are also more or less solved.

Some problems still remain. For some applications factors such as aspect ratio of the subdomains affect the convergence and there are few partitioning algorithms that take this into account [16, 84]. For multi-phase calculations, mesh partitioning algorithms that satisfy multiple constraints are needed. Implementation of dynamic load balancing on large scale applications also deserves further investigation. These could be areas where new results will emerge and conclusions will be drawn as to the types of applications where dynamic load balancing is effective.

Acknowledgments. The authors would like to thank the referees for very constructive and detailed reviews of the paper.

REFERENCES

[1] S. T. Barnard and H. D. Simon, *Fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems*, Concurrency: Practice and Experience, 6 (1994), pp. 101-117.

[2] S. T. Barnard and H. D. Simon, *A parallel implementation of multilevel recursive spectral bisection for application to adaptive unstructured meshes*, in: SIAM Proceedings Series 195, D.H. Bailey, P. E. Bjorstad, Jr Gilbert, M. V. Mascagni, R. S. Schreiber, H. D. Simon, V. J. Torczon, J. T. Watson, eds., SIAM, Philadelphia, 1995, pp. 627-632.

[3] S. T. Barnard, *PMRSB: parallel multilevel recursive spectral bisection*, Manuscript, 1996.

[4] Bertsekas and Tsitsiklis, *Parallel and Distributed Computation: Numerical Methods*, Prentice-Hall, Englewood Cliffs, NJ, 1989.

[5] R. Biswas and L. Oliker, *Experiments with repartitioning and load balancing adaptive meshes*, Technical Report NAS-97-021, NASA Ames Research Center, Moffett Field, CA, 1997.

[6] J. E. Boillat, *Load balancing and Poisson equation in a graph*, Concurrency: Practice and Experience, 2 (1990), pp. 289-313.

[7] J. E. Boillat and F. Brugé, *A dynamic load-balancing algorithm for molecular dynamics simulation on multi-processor systems*, Journal of Computational Physics, 96 (1991), pp. 1-14.

[8] N. Briggs, *Algebraic Graph Theory*, Cambridge University Press, Cambridge, 1974.

[9] W. L. Briggs, *A Multigrid Tutorial,* SIAM, Philadelphia, 1987.

[10] T. N. Bui and B. R. Moon, *Genetic algorithm and graph partitioning*, IEEE Transactions on Computers, 45 (1996), pp. 841-855.

[11] T. N. Bui, S. Chaudhuri, F. T. Leighton, M. Sipser, *Graph bisection algorithms with good average case behavior*, Combinatorica, 7 (1987), pp. 171-191.

[12] U. V. Çatalyürek and C. Aykanat, *Decomposing Irregularly Sparse Matrices for Parallel Matrix-Vector Multiplication,* Lecture Notes in Computer Science, 1117 (1996), pp. 75-86.

[13] H.L. de Cougny, K. D. Devine, J. E. Flaherty, R. M. Loy, C. Özturan and M. S. Shephard, *Load balancing for the parallel adaptive solution of partial differential equations*, Applied Numerical Mathematics, 16 (1994), pp. 157–182.

[14] G. Cybenko, *Dynamic load balancing for distributed memory multi-processors*, Journal of Parallel and Distributed Computing, 7 (1989), pp. 279-301.

[15] K. Devine, J. Flaherty, S. Wheat and A. Maccabe, *A massively parallel adaptive finite element method with dynamic load balancing*, Supercomputing, pp.2–11, 1993.

[16] R. Diekmann, D. Meyer and B. Monien, *Parallel decomposition of unstructured FEM-meshes*, Concurrency: Practice and Experience, 10 (1998), pp. 53-72.

[17] R. Diekmann, S. Muthukrishnan and M. V. Nayakkankuppam, *Engineering diffusive load balancing algorithms using experiments* Lecture Notes in Computer Science, 1253 (1997), pp. 111-122.

[18] P. Diniz, S. Plimpton, B. Hendrickson and R. Leland, *Parallel algorithms for dynami-*

cally partitioning unstructured grids, in: SIAM Proceedings Series 195, D.H. Bailey, P. E. Bjorstad, Jr Gilbert, M. V. Mascagni, R. S. Schreiber, H. D. Simon, V. J. Torczon, J. T. Watson, eds., SIAM, Philadelphia, 1995, pp. 627-632.

[19] R. Van Driessche and D. Roose, *Dynamic load balancing with a spectral bisection algorithm for the constrained graph partitioning problem*, Lecture Notes in Computer Science, 919 (1995), pp. 392-397.

[20] C. Farhat, *A simple and efficient automatic FEM domain decomposer*, Computer and Structures, 28 (1988), pp. 579-602.

[21] C. Farhat, S. Lanteri and H. D. Simon, *TOP/DOMDEC - a software tool for mesh partitioning and parallel-processing*, Computing Systems in Engineering, 6 (1995), pp. 13-26.

[22] C. Fahat, N. Maman and G. W. Brown, *Mesh partitioning for implicit computations via iterative domain decomposition - impact and optimization of the subdomain aspect ratio*, International Journal for Numerical Methods in Engineering, 38 (1995), pp. 989-1000.

[23] D. Vanderstraeten, C. Farhat, P. S. Chen, R. Keunings and O. Ozone, *A retrofit based methodology for the fast generation and optimization of large-scale mesh partitions - beyond the minimum interface size criterion*, Computer Methods in Applied Mechanics and Engineering, 133 (1996), pp. 25-45.

[24] C. M. Fiduccia and R. M. Mattheyses, *A linear-time heuristic for improving network partitions*, ACM Ieee Nineteenth Design Automation Conference Proccedings, vol.1982, ch.126, pp. 175-181, 1982.

[25] J. E. Flaherty, R. M. Loy, M. S. Shepard, B. K. Szymanski, J. D. Teresco and L. H. Ziantz, *Adaptive local refinement with octree load balancing for the parallel solution of three- dimensional conservation laws*, Journal of Parallel and Distributed Computing, 47 (1997), pp. 139-152.

[26] J. E. Flaherty, R. M. Loy, C. Özturan, M. S. Shepard, B. K. Szymanski, J. D. Teresco and L. H. Ziantz, *Parallel structures and dynamic load balancing for adaptive finite element computation*, Applied Numerical Mathematics, 26 (1998), pp. 241-263.

[27] R. Fletcher, *Practical Methods of Optimization*, John Wiley and Sons, Chichester, 1987.

[28] B. Ghosh and S. Muthukrishnan, *Dynamic load balancing on parallel and distributed networks by random matchings*, in Proceeding of Sixth Annual ACM Symposium om Parallel Algorithms and Architechures, pp. 226-235, 1994.

[29] D. H. Golub and C. F. Van Loan, *Matrix Computations*, Johns Hopkins University Press, Baltimore, 1983.

[30] C. Greenough and R. F. Fowler, *Partitioning methods for unstructured finite element meshes*, RAL report RAL-94-092, Rutherford Appleton Laboratory, UK, 1994.

[31] A. Gupta, *Fast and effective algorithms for graph partitioning and sparse-matrix ordering*, IBM Journal of Research and Development, 41 (1996), pp. 171-183.

[32] A. Gupta, *WGPP: Watson Graph Partitioning (and Sparse Matrix Ordering) Package: Users' Manual*, Technical Report RC-20453 (90427), IBM Thomas J. Watson Research Center, Yorktown Heights, NY, May 6, 1996.

[33] A. Heirich and S. Tayler, *A parabolic load balancing method*, International Conference on Parallel Processing, 1995

[34] B. Hendrickson and R. Leland, *An improved graph partitioning algorithm for mapping parallel computations*, Sandia National Laboratories, Albuquerque, NM 87185, 1992.

[35] B. Hendrickson and R. Leland, *The Chaco User's Guide*, Version 2.0, Technical Report SAND 94-2692, Sandia National Laboratories, Allbuquerque, NM, 1995.

[36] B. Henderson and R. Leland, *A multilevel algorithm for partitioning graphs*, in Proceeding of Supercomputing '95, ACM, December 1995.

[37] B. Hendrickson, R. Leland and R. Van Driessche, *Enhancing Data Locality by Using Terminal Propagation*, Proc. 29th Hawaii Intl. Conf. System Science, 1996.

[38] B. Hendrickson, *Graph partitioning and parallel solvers: has the emperor no clothes?* Lecture Notes in Computer Science, 1457 (1998), pp. 218-225.

[39] D. Hilbert, *Uber die steitige abbildung einer linie auf ein flachenstuck*, Math. Ann., 38, 1891.

[40] D. C. Hodgson and P. K. Jimack, *Efficient parallel generation of partitioned, unstructured meshes*, Advances in Engineering Software, 27 (1996), pp. 59-70

[41] G. Horton, *A multi-level diffusion method for dynamic load balancing*, Parallel Computing, 9 (1993), pp. 209-218.

[42] Y .F. Hu and R. J. Blake, *Numerical experiences with partitioning of unstructured meshes*, Parallel Computing, 20 (1994), pp. 815-829.

[43] Y. F. Hu, R. J. Blake and D. R. Emerson, *An optimal dynamic load balancing algorithm*, Concurrency: Practice and Experience, 10 (1998), pp. 467-483.

[44] Y. F. Hu, *An Improved Diffusion Algorithm for Dynamic Load Balancing*, 1997. To appear in

Parallel Computing.

[45] Y. F. Hu and R. J. Blake, *The optimal property of polynomial based diffusion-like algorithms in dynamic load balancing*, in: Computational Dynamics'98, K. D. Papailiou, D. Tsahalis, J. Périaux, D. Knörzer, eds., John Wiley & Son, Chichester, 1998.

[46] M. T. Jones and P. E. Plassmann, *Parallel algorithms for the adaptive refinement and partitioning of unstructured meshes*, in Proceedings of the Scalable High-performance Computing Conference (SHPCC94), 1994, Ch.111, pp. 478-485. Available from *http://www-unix.mcs.anl.gov/~freitag/SC94demo/paper.html*

[47] G. Karypis and V. Kumar, *A fast and high quality multilevel scheme for partitioning irregular graphs*, Technical Report TR 95-035, Department of Computer Science, University of Minnesota, 1995.

[48] G. Karypis and V. Kumar, *Parallel multilevel graph partitioning*, Technical Report TR 95-036, Department of Computer Science, University of Minnesota, 1995.

[49] G. Karypis and V. Kumar, *Parallel Multilevel k-way Partitioning Scheme for Irregular Graphs*, Technical Report TR 96-036, Department of Computer Science, University of Minnesota, 1996.

[50] G. Karypis and V. Kumar, *A Coarse-Grain Parallel Formulation of a Multilevel k-way Graph Partitioning Algorithm*, Eighth SIAM Conference on Parallel Processing for Scientific Computing, 1997.

[51] G. Karypis and V. Kumar, *Multilevel algorithms for multi-constraint graph partitioning*, Technical Report TR 98-019, Department of Computer Science, University of Minnesota, 1998.

[52] G. Karypis and V. Kumar, *Multilevel k-way partitioning scheme for irregular graphs*, Journal of Parallel and Distributed Computing, 48 (1998), pp. 96-129.

[53] B. W. Kernighan and S. Lin, *An efficient heuristic procedure for partitioning graphs*, Bell Systems Tech. J., 49 (1970), pp. 291-308.

[54] J. De Keyser and D. Roose, *Grid partitioning by inertial recursive bisection*, Report TW 174, K.U.Leuven, Dept of Computer Science, Belgium, July 1992.

[55] G. A. Kohring, *Dynamic load balancing for parallel particular simulation on MIMD computers*, Parallel Computing, 21 (1995), pp. 683-693.

[56] N. P. Kruyt, *A conjugate gradient method for the spectral partitioning of graphs*, Parallel Computing, 22 (1997) pp. 1493-1502.

[57] M. Luby, *A simple parallel algorithm for the maximal independent set problem*, SIAM Journal on Computing, 15 (1986), pp. 1036-1053.

[58] J. G. Malone, *Automated mesh decomposition and concurrent finite element analysis for hypercube multiprocessor computers*, Computer Methods in Applied Mechanics and Engineering, 70 (1988), pp. 27-58.

[59] N. Mansour and G. C. Fox, *Allocating data to distributed-memory multiprocessors by genetic algorithms*, Concurrency: Practice and Experience, 6 (1994), pp. 485-504.

[60] N. Mansour, *Physical optimization algorithms for mapping data to distributed-memory multiprocessors*, Ph. D. thesis, School of Computer Science, Syracuse University, 1992.

[61] N. Mansour, *Allocating data the multicomputer nodes by physical optimization algorithms for loosely synchronous computations*, Concurrency: Practice and Experience, 4 (1992), pp. 557-574.

[62] O. C. Martin and S. W. Otto, *Partitioning of unstructured meshes for load balancing* Concurrency: Practice and Experience, 7 (1995), pp. 303-314.

[63] B. Mohar, *The Laplacian spectrum of graphs*, technical Report, Department of Mathematics, University of Ljubljana, Ljubljana, Yugoslavia, 1988.

[64] S. Muthukrishnan, B. Ghosh and M. H. Schultz, *First- and second-order diffusive methods for rapid, coarse, distributed load balancing*, Theory of Computing Systems, 31 (1998), pp. 331-354.

[65] L. Oliker and R. Biswas, *PLUM: Parallel Load Balancing for Adaptive Unstructured Meshes*, Technical Report NAS-97-020, NASA Ames Research Center, Moffett Field, CA, 1997.

[66] C. Ou, S. Ranka, and G. Fox, *Fast and parallel mapping algorithms for irregular problems*, Journal of Supercomputing, 10 (1996), pp. 119-140

[67] C.-W. Ou and S. Ranka, *Parallel incremental graph partitioning*, IEEE Transactions on Parallel and Distributed Systems, 8 (1997), pp. 884-896.

[68] A. Pothen, D. H. Simon and K. P. Liou, *Partitioning sparse matrices with eigenvectors of graphs*, SIAM Journal of Matrix Analysis and Applications, 11 (1990), pp. 430-452.

[69] P. Sadayappan, F.Ercal and J. Ramanujam, *Cluster partitioning approach to mapping parallel programs onto a hypercube*, Parallel Computing, 13 (1990), pp. 1-16.

[70] K. Schloegel, G. Karypis and V. Kumar *Multilevel diffusion schemes for repartitioning of*

adaptive meshes, Journal of Parallel and Distributed Computing, 47 (1997), pp. 109-124

[71] K. Schloegel, G. Karypis and V. Kumar, *Parallel multilevel diffusion algorithms for repartitioning of adaptive meshes*, Technical Report TR 97-014, Department of Computer Science, University of Minnesota, 1997.

[72] K. Schloegel, G. Karypis, V. Kumar, R. Biswas and L. Oliker, *A performance study of diffusive vs. remapped load-balancing schemes*, Technical Report TR 98-018, Department of Computer Science, University of Minnesota, 1998.

[73] K. Schloegel, G. Karypis, V. Kumar, *Wavefront diffusion and LMSR: algorithms for dynamic repartitioning of adaptive meshes* , Technical Report TR 98-034, Department of Computer Science, University of Minnesota, 1998.

[74] M. S. Shephard, J. E. Flaherty, H. L. de Cougny, C. Özturan, C. L. Bottasso and M. W. Beall, *Parallel automated adaptive procedures for unstructured meshes*, in Parallel Computing in CFD, AGARD-R-807, pp. 6.1–6.49, 1995.

[75] H. D. Simon, *Partitioning of unstructured problems for parallel processing*, Computer Systems in Engineering, 2 (1991), pp. 135-148.

[76] H. D. Simon and S. H. Teng, *How good is recursive bisection*, SIAM Journal on Scientific Computing, 18 (1997), pp. 1436-1445.

[77] H. D. Simon, A. Sohn, and R. Biswas, *HARP: A Dynamic Spectral Partitioner*, Journal of Parallel and Distributed Computing, 50 (1998), pp. 88-103.

[78] J. Song, *A partially asynchronous and iterative algorithm for distributed load balancing*, Parallel Computing, 20 (1994), pp. 853-868.

[79] D. Vanderstraeten and R. Keunings, *Optimized partitioning of unstructured finite-element meshes*, International Journal for Numerical Methods in Engineering, 38 (1995), pp. 433-450.

[80] C. Walshaw, M. Cross, S. Johnson and M. Everett, *A parallelisable algorithm for partitioning unstructured meshes*, in: Proceeding of Irregular '94: Parallel Algorithms for Irregular Problems: State of the Art, A. Ferreira and J. Rolim, eds, Kluwer Academic Publishers, Dordrecht, 1995, pp. 23-44.

[81] C. Walshaw and M. Berzins, *Dynamic load-balancing for PDE solvers on adaptive unstructured meshes*, Concurrency: Practice and Experience, 7 (1995), pp. 17-28.

[82] C. Walshaw, M. Cross and M. Everett, *Dynamic mesh partitioning: a unified optimisation and load-balancing algorithm*, Technical Report 95/IM/06, University of Greenwich, London SE18 6PF, UK, 1995.

[83] C. Walshaw, M. Cross and M. Everett, *PARALLEL dynamic load balancing for adaptive unstructured meshes*, Journal of Parallel and Distributed Computing, 47 (1997), pp. 102-108.

[84] C. Walshaw, M. Cross, R. Diekmann, and F. Schlimbach, *Multilevel mesh partitioning for aspect ratio*, in Proc. VecPar'98, Porto, Portugal, pages 381-394, Universidade do Porto, 1998.

[85] C. Walshaw and M. Cross, *Parallel optimisation algorithms for multilevel mesh partitioning*, Technical Report 99/IM/44, University of Greenwich, London SE18 6PF, UK, 1999.

[86] H. G. Weller, G. Tabor, H. Jasak and C. Fureby, *A tensorial approach to computational continuum mechanics using object oriented techniques*, Thermofluids Section Report TF-98/03, Department of Mechanical Engineering, Imperial College, UK, 1998.

[87] R. D. Williams, *Performance of dynamic load balancing algorithms for unstructured mesh calculations*, Concurrency: Practice and Experience, 3 (1991), pp. 457-481.

[88] C. Z. Xu and F. C. M. Lau, *Analysis of the generalized dimension exchange method for dynamic load balancing*, Journal of Parallel and Distributed Computing, 16 (1992), pp. 385-393.

[89] C. Z. Xu and F. C. M. Lau, *The generalized dimension exchange method for load balancing in K-ary ncubes and variants*, Journal of Parallel and Distributed Computing, 24 (1995), pp. 72-85.