An optimal migration algorithm for dynamic load balancing

Y. F. Hu, R. J. Blake and D. R. Emerson

Daresbury Laboratory

Daresbury

Warrington WA4 4AD

United Kingdom

SUMMARY The problem of redistributing the work load on parallel computers is considered. An optimal redistribution algorithm, which minimises the Euclidean norm of the migrating load, is derived. The relationship between this algorithm and some existing algorithms is discussed and the convergence of the new algorithm is studied. Finally, numerical results on randomly generated graphs as well as on graphs related to real meshes are given to demonstrate the effectiveness of the new algorithm.

1. INTRODUCTION

To achieve good performance on a parallel computer, it is essential to establish and maintain a balanced work load among all the processors. Sometimes the load can be balanced statically, but in many cases the load on each processor can not be predicted *a priori*.

One example that demonstrates the need for both static and dynamic load balancing strategies is the parallel finite element solution of PDE's based on unstructured meshes. To achieve high precision while minimising computational work and memory requirement, adaptive meshing techniques can be used. An adaptive finite element code starts from a relatively coarse initial mesh, but gradually refines the mesh every few iterations.

Assume that the amount of work on each processor is proportional to the number of mesh nodes on the processor. The static load balancing problem seeks to partition the initial mesh into subdomains, the number of which equals the number of processors, such that:

- each subdomain has an equal number of nodes, so as to balance the load;
- the number of shared edges (*edge cuts*) between subdomains and the number of neighbouring subdomains are as small as possible, so as to minimise the communication cost.

The static mesh partitioning problem has been studied extensively by many workers (see, e.g., [1, 2]). Among all the algorithms, the recursive spectral bisection algorithm [2, 3], which uses an eigenvector of the Laplacian matrix of the graph (or the dual graph, for element based applications) of a mesh as a separator, has been found to give partitions of good quality (i.e., small number of *edge cuts*). A multilevel implementation of the algorithm [4] was suggested to reduce the computational cost of finding the eigenvector. The multilevel idea can also be combined with local optimization strategies to derive partitioning algorithms that are able to partition very large meshes in a reasonable amount of time [5, 6, 7, 8, 9]. For very large meshes, the sheer memory requirement means that parallel mesh partitioning algorithms will have to be used. This is an active area of research [7, 10, 11].

Once the initial mesh has been partitioned, either sequentially or in parallel, and migrated to the processors, calculations can then be carried out. After an interval of computation, the mesh may be refined at some locations, usually based on an estimate of the discretisation error. The refinement process might generate widely varying numbers of mesh nodes on the processors, thus the need for dynamic load balancing. As an example, Figure 1 shows part of a mesh around a three element airfoil, partitioned into 8 subdomains. Due to mesh refinement the number of nodes on each subdomain is different. Figure 2 shows, for instance, subdomain 8 has 754 nodes, while subdomain 4 has only 465 nodes.

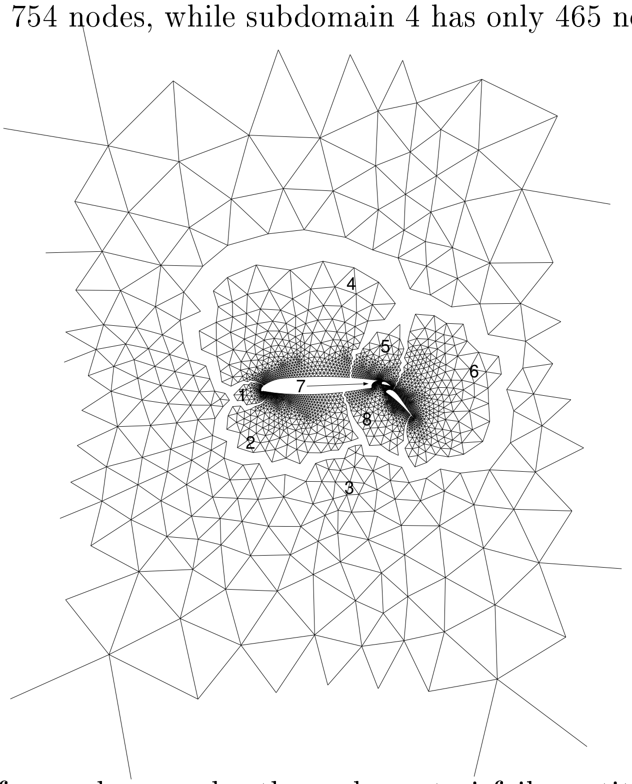

Figure 1: Part of a mesh around a three element airfoil, partitioned into 8 subdomains.

One way to re-balance the load is to *repartition* the mesh using one of the mesh partitioning algorithms mentioned above. But it is difficult to ensure that the new partitioning will be "close" to the original partitioning. Should the new partitioning deviates considerably from the old then the cost of transferring large amounts of data, in addition to that of the mesh partitioning, will be incurred. An alternative strategy is to *migrate* the nodes among neighbouring processors (neighbouring in the sense that these processors share boundaries), effectively shifting the boundaries to achieve a balanced load. This should involve far less movement of data compared with repartitioning, although the number of *edge cuts* after the migration could possibly be larger than that given by the reparti-

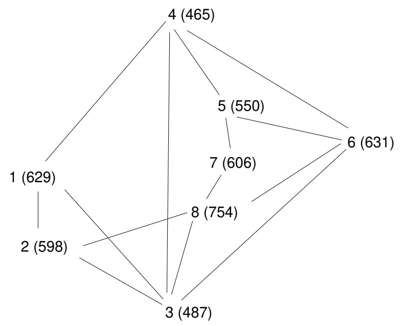

Figure 2: The processor graph associated with the partitioned mesh in Figure 1, and the load on each processor (in bracket).

tioning. Therefore care must be taken to keep the number of *edge cuts* down when choosing the nodes to be migrated. Nonetheless migration is normally preferred to repartitioning.

In this paper we shall concentrate on the migration strategy. The process of migrating loads between processors to achieve a balanced load can be broken down into two distinctive steps [6, 7]:

- Step 1 (scheduling): Each processor works out a schedule for the exact amount of load that it should send to (or receive from) its neighbouring processors;

- Step 2 (migration): Once the above schedule is worked out, each processor decides which particular nodes it should send to or receive from its neighbouring processors. The migration then takes place.

Step 2 (migration) has been studied by a number of authors. The popular strategy is to start from the boundary nodes and gradually move to the interior of the mesh, until enough nodes are marked for migration (see, e.g., [5, 6, 7, 11]).

Scheduling algorithms for Step 1 are mostly iterative. Note that there is a startup cost for communication on parallel computers – the latency, which is usually very high compared with the subsequent cost of transmitting a word. Thus for many applications it is better if the migration of load does not physically take place until the scheduling algorithm has converged. The final schedule can then be used for the load migration.

The most popular scheduling algorithms are diffusion type iterative algorithms. These algorithms are asynchronous and are therefore suitable for applications where the parallelism is fine grained and the load transfer takes place alongside the iterations of the scheduling algorithm. However, in applications such as finite element calculation with adaptive meshes, where it is beneficial not to carry out the load migration until the scheduling algorithm has converged, diffusion type algorithms may not be very suitable. Because convergence for such algorithms can be very slow [5].

The motivation of this paper is therefore to propose a more efficient algorithm for scheduling. For the rest of the paper we will study the following dynamic load balancing problem.

The dynamic load balancing problem: Find a schedule for the number of nodes (work load) to be migrated between processors, such that each processor will have the same load if load migration based on the schedule is carried out.

In devising scheduling algorithms for dynamic load balancing, a number of considerations are relevant. First, it is hoped that the schedule will balance the load with minimal data movement between processors, because communication is expensive compared with computation. Second, from the point of view of reducing the number of *edge cuts*, it is more desirable for the data movement to be restricted to the neighbouring processors. Here it is important to differentiate between the graph induced by a particular partitioning of a mesh (termed "processor graph", or simply "graph" later on), and that of the processor topology. Figure 2 gives the processor graph associated with the partitioned mesh of Figure 1, together with the load (in brackets) on each processor. Two processors are linked with an edge and are thus neighbours if they share a boundary. For instance, processor 1 is linked with processors 2, 3 and 4, but not to processor 6. By restricting the data movement to neighbouring processors, the processor graph will stay the same. This is important because, if there is no such restriction, then after a few dynamic load balancing steps all processors may well share boundaries with each other. Finally, the scheduling algorithm itself should take little time in comparison with the time taken by the application code in between mesh refinement.

In the next section, a brief review of existing algorithms for the dynamic load balancing problem is given. In section 3, a dynamic load balancing algorithm, which minimises the Euclidean norm of the data movement, is derived. In Section 4, the algorithm is viewed in the light of the unsteady heat conduction equation. The relationship between this algorithm and other algorithms is discussed. In Section 5, theoretical results of the convergence of this algorithm on special graphs are given. In Section 6, numerical tests are carried out to demonstrate the effectiveness of the algorithm. Section 7 concludes the paper with some discussions.

2. EXISTING ALGORITHMS

The dynamic load balancing problem is analogous to the diffusion process, where an initial uneven temperature or concentration distribution in space drives the movement of heat (or chemicals), and eventually reaches equilibrium. It is thus not surprising that a number of algorithms based on this analogy have been proposed. Cybenko [12] assumed that work load was infinitely divisible and suggested a diffusion algorithm where each processor exchanges load with its neighbours, the amount of which is proportional to the difference in their loads. The algorithm is iterative and converges to a steady state. Similar algorithms have been suggested independently by Boillat [13], and linked to the Poisson equation for the graphs. Cybenko [12] also suggested a so-called dimension exchange algorithm, in which processors were grouped in pairs and processors i and j with loads l_i and l_j will exchange work and result in a mean load of $(l_i + l_j)/2$. The algorithm converges in d steps, if the graph considered is a hypercube with dimension d. Xu and Lau [14, 15] extended the dimension exchange algorithm so that after the exchange processor i will have load $l_i * \lambda + l_j * (1 - \lambda)$. If $\lambda = 0.5$ this is equivalent to Cybenko's algorithm. Based on the eigenvalue analysis of the underlining iterative matrices, they argued that for some graphs a factor λ of other than 0.5 will converge more quickly. Song [16] suggested an asynchronous algorithm and proved its convergence based on the theory given by Bertsekas and Tsitsiklis [17]. The work load is assumed to be integer and the algorithm gave a maximum of $[d/2]$ load difference between processors, with d being the diameter of the processor graph. Other modified diffusion type algorithms have also been suggested [18, 19] and applied in areas such as molecular dynamic simulation.

One of the disadvantages of diffusion like algorithms is their possible slow convergence, particularly near equilibrium, for reasons analogous to the slow convergence of the Jacobi algorithm when solving linear systems. The rate of convergence of the diffusion algorithm on a graph is related to the value of the smallest positive eigenvalue of its Laplacian [13], which in turn is related to the number of *edge cuts* that can be obtained from partitioning the graph. For graphs that have a small number of *edge cuts*, the convergence can be slow. Boillat [13] proved that the worst case happens when the graph is 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 processors.

To speedup the process, Horton [20] 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 work 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-neighbouring processors and should be avoided, as explained in Section 1.

All the aforementioned algorithms do not take into account one important factor, namely that the data movement resulting from the load balancing schedule should be kept to a minimum. As discussed before, this is important because data movement between processors is expensive. Furthermore, for irregular mesh applications, by keeping the number of nodes migrated between processors small, it is more likely that the resulting number of *edge cuts* will not increase significantly.

3. AN OPTIMAL DYNAMIC LOAD BALANCING ALGORITHM

Let p be the number of processors. Let $(V,\ E)$ be the processor graph, where $V = (1, 2, \ldots, p)$ is the set of vertices each representing a processor, and E is the set of edges. The graph is assumed to be connected. Two vertices i and j form an edge if processors i and j share a boundary of the partitioning. Associated with each processor i is a scalar l_i representing the load on the processor. The average load per processor is

$$\bar{l} \;=\; \frac{\sum_{i=1}^{p} \; l_i}{p}.$$

Each edge (i,j) of the graph also has a scalar δ_{ij} associated with it, where δ_{ij} is the amount of load to be sent from processor i to processor j. The variables δ_{ij} are directional, that is,

$$\delta_{ij} = -\delta_{ji}. \tag{1}$$

This represents the fact that if processor i is to send the amount δ_{ij} to processor j, then processor j is to receive the same amount (to send $-\delta_{ij}$).

In reality, of course, the work load will be an integer number. In the case of finite element applications this can be the number of nodes on each processor. However we shall assume for the moment that the work load on each processor is a real number that is infinitely divisible. The case when the work load is an integer will be discussed in Section 6.

A load balancing schedule should make the load on each processor equal to the average load, that is,

$$\sum_{\{j \;|\; (i,j)\in E\}} \delta_{ij} \;=\; l_i \;-\; \bar{l}, \quad i = 1, 2, \ldots, p. \tag{2}$$

If $i > j$ and $(i,j) \in E$, vertex i will be called the head of the edge (i,j), and j the tail. Because of (1), we shall only keep δ_{ij} as a variable if i is the head of edge (i,j), but replace δ_{ij} with $-\delta_{ji}$ if i is the tail.

If $p-1$ equations of (2) are satisfied, the remaining one equation will be satisfied automatically. Thus the number of independent equations is no more than the number of vertices minus one. The number of variables in the system of equations (2), on the other hand, is equal to the number of edges in the graph. There are usually far more edges in a graph than vertices, and in any case for a connected graph $|E| \geq |V| - 1$, where $|E|$ and $|V|$ are the number of edges and vertices of the graph (E, V) respectively. Therefore (2) is likely to have infinitely many solutions. We shall choose amongst these solutions one that minimises the data movement.

Let A be the matrix associated with (2), x the vector of δ_{ij}'s and b the right hand side. Assuming that the Euclidean norm of the data movement is used as a metric and that the communication cost between any two processors is the same (which is roughly the case for many modern parallel computers such as the Cray T3D), the problem becomes

$$\begin{aligned} \text{Minimise} \quad & \tfrac{1}{2}x^T x, \\ \text{subject to} \quad & Ax = b. \end{aligned} \tag{3}$$

Here A is the $|V| \times |E|$ matrix, 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 [21] on (3) gives

$$x = A^T \lambda, \tag{4}$$

where λ is the vector of Lagrange multipliers. Substituting back into (2) gives

$$L\,\lambda \;=\; b, \tag{5}$$

with $L = A\,A^T$ a matrix of size $|V| \times |V|$.

To illustrate the matrices A and L, consider a simple graph of three vertices linked by a line and let $(1, 2)$ and $(2, 3)$ be the first and second edges, then

$$A = \begin{pmatrix} -1 & 0 \\ 1 & -1 \\ 0 & 1 \end{pmatrix}$$

and

$$L = AA^T = \begin{pmatrix} 1 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 1 \end{pmatrix}.$$

The problem of finding an optimal load balancing schedule therefore becomes that of solving the linear equation (5). It is not difficult to confirm that the matrix L is in fact the Laplacian matrix of the graph with dimension $|V| \times |V|$, defined as

$$(L)_{ij} = \begin{cases} -1, & \text{if } \ i \neq j \text{ and edge } \ (i,j) \in E, \\ deg(i), & \text{if } \ i = j, \\ 0, & \text{otherwise.} \end{cases} \tag{6}$$

Here $deg(i)$ is the degree of vertex i in the graph.

Once the Lagrange vector λ is solved from (5), then by equation (4) and due to the special form of A^T (each row of the matrix has only two non-zeros of 1 and -1), the amount of load to be transferred from processor i to processor j is simply $\lambda_i - \lambda_j$, where λ_i and λ_j are the Lagrange multipliers associated with processors i and j respectively.

Thus the new load balancing algorithm is:

The new load balancing algorithm

- Step 1: Find the average work load, and thus the right hand side of (5);
- Step 2: Solve $L\lambda = b$ to obtain λ;
- Step 3: Determine the amount of load to be transferred. The amount processor i will send to processor j is $\lambda_i - \lambda_j$.

As a simple example, consider the processor graph in Figure 2. The load for each processor is given in brackets. The average load is 590 and the largest load imbalance is $(754 - 590)/590 = 28\%$. The Laplacian system is now

$$\begin{pmatrix} 3 & -1 & -1 & -1 & 0 & 0 & 0 & 0 \\ -1 & 3 & -1 & 0 & 0 & 0 & 0 & -1 \\ -1 & -1 & 5 & -1 & 0 & -1 & 0 & -1 \\ -1 & 0 & -1 & 4 & -1 & -1 & 0 & 0 \\ 0 & 0 & 0 & -1 & 3 & -1 & -1 & 0 \\ 0 & 0 & -1 & -1 & -1 & 4 & 0 & -1 \\ 0 & 0 & 0 & 0 & -1 & 0 & 2 & -1 \\ 0 & -1 & -1 & 0 & 0 & -1 & -1 & 4 \end{pmatrix} \begin{pmatrix} \lambda_1 \\ \lambda_2 \\ \lambda_3 \\ \lambda_4 \\ \lambda_5 \\ \lambda_6 \\ \lambda_7 \\ \lambda_8 \end{pmatrix} = \begin{pmatrix} 629 - 590 \\ 598 - 590 \\ 487 - 590 \\ 465 - 590 \\ 550 - 590 \\ 631 - 590 \\ 606 - 590 \\ 754 - 590 \end{pmatrix} = \begin{pmatrix} 39 \\ 8 \\ -103 \\ -125 \\ -40 \\ 41 \\ 16 \\ 164 \end{pmatrix}.$$

The solution of this linear equation is

$$(\lambda_1, \ldots, \lambda_8) = (-2.49, 11.03, -17.49, -40.48, -19.19, 2.34, 21.12, 45.15).$$

These Lagrange multipliers are illustrated in Figure 3 in brackets. The amount of load to be transferred between two neighbouring processors is the difference between their Lagrange multipliers, and is shown along the edges in Figure 3. For example, processor 8 needs to send to processor 6 a load of $45.15 - 2.34 = 42.81$.

It is important to note that it is not necessary to explicitly form and store the Laplacian matrix, and that the new algorithm can be implemented efficiently on parallel computers. Implementation details will be given in Section 6.

4. RELATIONSHIP WITH DIFFUSION ALGORITHMS

It is instructive to look at the dynamic load balancing problem as a diffusion or heat conduction process. Let ρ, c and k denote the density, specific heat and thermal conductivity respectively and assume that these coefficients are constant. Let $D = \frac{k}{c\rho}$, then the one dimensional heat conduction process can be described by the following equation

$$\frac{\partial u}{\partial t} - D \frac{\partial^2 u}{\partial x^2} = 0,$$

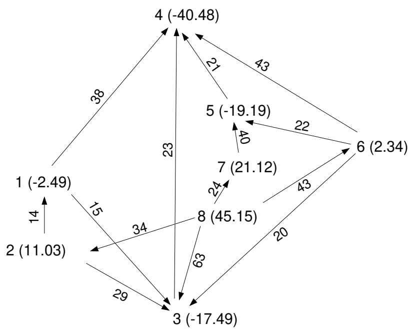

Figure 3: The Langrange multiplier (in bracket) associated with each processor, and the amount of load to be transferred (shown along the edges).

where u is the temperature, t is the time and x the space location. The boundary condition is assumed to be periodic.

If the above equation is discretised using forward difference in time and central difference in space, then

$$\frac{u(x,t+\Delta t)-u(x,t)}{\Delta t} = D\,\frac{u(x+\Delta x,t)+u(x-\Delta x,t)-2u(x,t)}{(\Delta x)^2},$$

or

$$u(x,t+\Delta t) = u(x,t) - \frac{D\Delta t}{(\Delta x)^2}\,[u(x,t)-u(x-\Delta x,t)+u(x,t)-u(x+\Delta x,t)].$$

This is just the equation in [12] applied to a simple graph of a ring. The above iterative process is stable and convergent only if $D\Delta t/(\Delta x)^2 \le 1/2$. The generalised form of this iterative process applied to a graph is

$$u \;\leftarrow\; u \;-\; R\,u, \tag{7}$$

with u the vector defined over the vertices and R a matrix that closely resembles the Laplacian matrix of the graph. Boillat [13] considered the convergence of this iterative scheme in detail.

The general form of the heat conduction equation is

$$c\rho\,\frac{\partial u}{\partial t} \;+\; div\ \mathbf{F} = 0, \tag{8}$$

where $\mathbf{F}$ is the heat flux defined as $\mathbf{F} = -k \bigtriangledown u$.

As mentioned in Section 1, it is the accumulated load (heat) transfer, rather than the actual history of load (temperature) from the initial state to the steady state, that is of interest. The accumulated heat flux is given by

$$\mathbf{Q} = \int_0^\infty \mathbf{F}\ dt.$$

Since $\mathbf{Q}$ is irrotational ($curl\ \mathbf{Q} = 0$), there exists a scalar field q, the potential, such that

$$\mathbf{Q} = \bigtriangledown q. \tag{9}$$

Integrating (8) over time $[0, \infty]$ gives

$$\frac{1}{c\rho}\int_0^\infty div\mathbf{F}\ dt = \frac{1}{c\rho} div\mathbf{Q} = \frac{1}{c\rho}\bigtriangledown^2 q = u|_{t=0} - u|_{t=\infty}. \tag{10}$$

The Poisson equation (10) is just the continuous form of equation (5), while equation (9) is the continuous form of equation (4).

The optimal algorithm suggested here is therefore closely related to the d-iffusion type algorithms in the sense that the underlining equation for both is (8). The difference is that the diffusion type algorithms integrate (8) directly by discretising over time as well as space, while the new algorithm suggested solves the time-integrated equation (10) and only spatial discretisation is needed.

5. CONVERGENCE RESULTS

The Poisson equation (10), or its discretised form (5), can be solved by many standard numerical algorithms. For example, stationary type algorithms, such as

Jacobi or Gauss-Seidel algorithms, can be used. The processor graph can also be coarsened to form a series of graphs, each coarser than the other, and multigrid type acceleration techniques can be employed [4, 11]

We propose to solve the Laplacian system (5) by the conjugate gradient algorithm [22] in parallel, because of its simplicity and fast convergence. It is well known (see, e.g., [3, 23]) that the Laplacian matrix L is positive semi-definite. It has an eigenvalue of zero associated with the eigenvector of all ones and if the graph is connected, the rest of the eigenvalues are all positive. Starting with a vector of all zeros, the iterates for the conjugate gradient algorithm will stay orthogonal to e, the vector of all ones, because $L\ e \ = \ 0$. Thus the conjugate gradient algorithm will converge in k iterations, where k is the number of distinct positive eigenvalues of the Laplacian matrix L (see [21] for the theory of conjugate gradient algorithms). Clearly $k \leq p - 1$.

For some special graphs, it is possible to work out the number of distinct eigenvalues of their Laplacian matrices and therefore the maximum number of iterations needed for the conjugate gradient algorithm to converge.

Theorem 1 The Laplacian of a hypercube of dimension d has d distinct positive eigenvalues.

Proof: Let C be the node adjacency matrix of the hypercube, defined in the same way as the Laplacian matrix L except that the entries on the diagonal are set to zero. It is known [12] that C has $d + 1$ distinct eigenvalues

$$-d,\ -d+2,\ -d+4,\ \ldots,\ d-4,\ d-2,\ d.$$

For a hypercube the degree for every vertex is d, thus the Laplacian matrix is simply

$$L = C \ + \ d\ I,$$

where I is the unit matrix. Thus L has $d+1$ distinct eigenvalues

$$0,\ 2,\ 4,\ \ldots,\ 2d-4,\ 2d-2,\ 2d,$$

of which d of them are positive. $\square$

By this theorem, for a hypercube of p vertices, the conjugate gradient algorithm converges in no more than $d = log_2(p)$ steps.

Theorem 2 The Laplacian of a complete graph of p vertices has only one positive eigenvalue of multiplicity $p-1$.

Proof: For a complete graph there is an edge between any two vertices, thus the Laplacian matrix is full and has entries of one on the off-diagonal positions and $p-1$ on the diagonal. It is easy to confirm that a vector with 1 and -1 as the only two non-zero entries is an eigenvector with an eigenvalue of $p+1$. As there are $p-1$ such independent vectors and the Laplacian has another eigenvalue of 0 associated with the vector of all ones, then the positive eigenvalue is $p+1$ with the multiplicity of $p-1$. $\square$

In this case the conjugate gradient algorithm will converge in 1 step.

Theorem 3 The Laplacian of a ring of p vertices has $[p/2]$ distinct positive eigenvalues.

Proof: The eigenvalue for the Laplacian matrix of a ring is (see, e.g., [13])

$$\lambda_k = \frac{2\ -\ 2\cos\frac{2(k-1)\pi}{p}}{3}, \quad k = 1, 2, \ldots, p. \tag{11}$$

As $\cos(x_1) = \cos(x_2)$ for any real number satisfying $x_1 + x_2 = 2\pi$, thus applied to equation (11) one has

$$\lambda_i = \lambda_j, \quad \text{if} \ \ i + j = p + 2.$$

If p is even, the matrix has λ_1 and $\lambda_{\frac{p}{2}} + 1$ as the eigenvalues of multiplicity one, while the rest of the eigenvalues have multiplicity of two because

$$\lambda_{\frac{p}{2}+1-k} = \lambda_{\frac{p}{2}+1+k}, \quad k = 1, 2, \ldots, \frac{p}{2} - 1.$$

Thus there are, in total, $(p/2 - 1) + 2 = [p/2] + 1$ distinct eigenvalues.

When p is odd, λ_1 is the eigenvalue of multiplicity of one while the other eigenvalues are all of multiplicity two because

$$\lambda_{\frac{p+1}{2}+k} = \lambda_{\frac{p+1}{2}-k+1}, \quad k = 1, 2, \ldots, \frac{p+1}{2} - 1.$$

Thus there are, in total, $(p+1)/2 = [p/2] + 1$ distinct eigenvalues.

So the number of distinct positive eigenvalues is $[p/2]$. $\square$

Therefore, the conjugate gradient algorithm on a ring will converge in no more than $[p/2]$ steps.

Theorem 4 The Laplacian on a 2-D torus of $p = n_1 \times n_2$ vertices has at most $([n_1/2] + 1) \times ([n_2/2] + 1)$ distinct eigenvalues.

Proof: Immediate from the proof of Theorem 3, and the fact that the eigenvalues for the 2-D torus are

$$\frac{4 \; - \; 2\,\cos\frac{2(k_1-1)\pi}{n_1} - 2\,\cos\frac{2(k_2-1)\pi}{n_2}}{5}, \quad k_1 = 1, \ldots, n_1; \; k_2 = 1, \ldots, n_2.$$

$\square$

In the special case when $n_1 = n_2 = n$, the number of distinct eigenvalues is further reduced by a factor of 2, and therefore for a 2-D torus of $p = n \times n$ processors, the conjugate gradient algorithm will converge in around $p/8$ or fewer iterations. For a 3-D torus of $p = n \times n \times n$, it will converge in around $p/48$ or fewer iterations.

6. NUMERICAL RESULTS

On graphs that do not have special structures, it is difficult to predict the convergence of the conjugate gradient algorithm. In this section the new method, combined with conjugate gradient algorithm, is therefore implemented numerically on a parallel computer and compared with a diffusion algorithm.

6.1 Parallel implementation of the new algorithm

On a parallel computer, each iteration of a standard conjugate gradient algorithm applied to the Laplacian system involves three global summations (or global maximum) of scalars, one matrix-vector multiplication and a few scalar floating point operations. The conjugate gradient algorithm is well known and will not be listed here (see [22]). The only operation that requires attention is the matrix-vector multiplication. On processor i, this gives

$$(L\lambda)_i \; = \; deg(i)\lambda_i \; - \sum_{\{j\,|\,(i,j)\in E\}} \lambda_j .$$

Here λ is the vector of the current estimated Lagrange multipliers. This is implemented on a parallel computer in two steps. On each processor:

- Send its Lagrange multiplier to its neighbour processors; receive neighbouring processors' Lagrange multipliers;
- Multiply its Lagrange multiplier by the number of neighbours, subtracting this by neighbouring processors' Lagrange multipliers.

The solution of (5) is not unique, because if λ is a solution of (5), then $\lambda+\alpha e$ is also a solution, where e is the vector of all ones and α any real number. However, for a connected graph, the Laplacian is of rank $p-1$. The amount of load transferred between two neighbouring processors, which is the difference between their λ's, is therefore unique.

In many applications, the load on each processor is an integer. For example, in finite element calculations it can be set to the number of nodes on the processor. Since the conjugate gradient algorithm works with real numbers, we suggest that the amount of load to be transferred is rounded to the nearest integer once the algorithm has converged. By doing so, the final load of any processor i will be no more than $deg(i)/2$ away from the average load. It is noted that this possible unbalance of final load is equally suffered by most of the existing algorithms, including the diffusion algorithms. The processor graph produced by good quality partitioning should have a small degree, and in any case $deg(i) \leq p-1$. Furthermore, for large calculations the number of nodes on each processor will be much larger than the number of processors, the new algorithm should therefore give a good balanced load.

6.2 Comparison of the new algorithm with a diffusion algorithm

The new dynamic load balancing algorithm, combined with the conjugate gradient solver, has been implemented in parallel. As a comparison, the diffusion algorithm, as described in [13], has also been implemented in parallel. At each iteration in the diffusion algorithm, the new work load on processor i is given by

$$l_i \;\leftarrow\; l_i \;-\; \sum_{\{j \;|\; (i,j)\in E\}} c_{ij}(l_i - l_j),$$

with c_{ij} chosen to be $1/(1 + max\{\ deg(i), deg(j)\ \})$. In implementing the diffusion algorithm the work load is also assumed to be a real number and the final accumulated load transfer between processors is rounded to the nearest integer.

The convergence criterion for both algorithms is

$$\text{load imbalance} \;=\; \max_{i\in V} \left\{ \frac{l_i - \bar{l}}{\bar{l}} \right\} \;<\; \epsilon, \qquad (12)$$

where ϵ is set to 10^{-3}. This is checked at each iteration of the conjugate gradient algorithm. For the diffusion algorithm, in order to reduce the synchronisation time, this is only checked every 5 iterations.

Randomly generated graphs were first used as processor graphs to test the two algorithms. The reason for testing on random graphs is that it is easy to control the average degree of the graphs. This allows a thorough comparison of the two algorithms over a wide range of graph connectivity.

A random graph generator has been written. Given p vertices, the generator randomly links vertices until the average degree of the graph reaches the preset value. The graph is then checked for its connectivity, and extra edges added if the graph is found to be disconnected. The final degree of the graph can therefore be slightly larger than the preset value due to the extra edges. The load on each processor was randomly set to be between 1000 and 5000.

The two dynamic load balancing algorithms were tested on a Cray T3D parallel computer for up to 256 processors, using PVM for message passing and a hand coded global summation (and global maximum) routine. Table 1 shows the number of iterations and the elapsed times of the two algorithms against the average degree and diameter of the graphs. The preset values for the average degree were chosen as 1, 3, 5, 7, 9. For each value, three random graphs were generated and the averaged results of the two algorithms over the three random graphs are given in the table. As can be seen from Table 1, the number of iterations for the new algorithm is always less than p, and decreases with the increase of the average degree. The number of iterations for the diffusion algorithm, on the other hand, can be very large if the degree of the graphs is small. It decreases rapidly with the increase of the degree of the graphs. As the degree increases, the number of iterations for the two algorithms finally converges. In terms of elapsed time, similar trends is observed. For almost all the graphs tested, the new algorithm takes less time to converge than the diffusion algorithm, even though the cost per iteration for the new algorithm is higher.

Table 1 here

It is also interesting to see how the two algorithms scale with the number of processors. Table 1 clearly shows that for the processor graphs with small average degrees, the number of iterations for the diffusion algorithm increases quadratically with the number of processors, while that for the new algorithm increases linearly. For graphs with a high average degree, both algorithms scale sub-linearly.

Both algorithms give migration schedules with a good load balance. The worst load imbalance recorded was 0.24%. The Euclidean norm of the load migration was also looked at and in most cases the new algorithm gives smaller norms. As previously discussed, if the amount of load migration is assumed to be a real number, then the new algorithm should always give smaller or equal Euclidean norms. But after rounding to integers this may not be the case. The difference between the norms of the two algorithms is small, however, indicating that the diffusion process might also possesses some minimal energy property.

The algorithms were further tested on processor graphs and loads related to two meshes, Tri60K and Tet100K [5], generated using the dynamic mesh partitioning package JOSTLE [5, 7]. The initial load imbalance ranges from 10-50%. The results of the two algorithms are listed in Table 2. In general the new algorithm performs better than the diffusion algorithm.

Table 2 here

In conclusion, the new dynamic load balancing algorithm is, for almost all cases, faster than the diffusion algorithm. This is true for random processor graphs with up to 256 vertices and with an average degree of up to 9, as well as for processor graphs related to the partitioning of real meshes. In particular, the new algorithm is superior to the diffusion algorithm for graphs with a small degree.

As the average degree of the processor graphs from good quality partitioning of finite element meshes is usually small, this makes the new algorithm very suitable for such applications. The algorithm has recently been incorporated [7] in the parallel version of the JOSTLE package to replace a diffusion type algorithm.

The new algorithm is expected to perform even better, in comparison with the diffusion type algorithms, for future massively parallel computers with thousands, rather than hundreds, of processors.

7. DISCUSSIONS

In this paper, an optimal dynamic load balancing algorithm has been suggested and was demonstrated to be able to generate a good load balancing schedule in very little time. The algorithm is synchronous and is more suitable for applications where the parallelism is coarse grained, or the load does not change very rapidly between iterations. Finite element calculation is one such example. For other applications, where the parallelism is fine grained and the load changes rapidly every iteration, an asynchronous diffusion type algorithm may be more suitable because it has lower or no synchronisation cost.

The algorithm was derived by minimising the Euclidean norm of the load transfer. An alternative measure of the cost of load migration, is probably 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 optimal scheduling problem (3) then becomes

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

which is equivalent to

$$\begin{aligned} & \text{Minimise} \quad c, \\ \text{subject to} \quad & Ax = b, \; c \geq (t_0 + \alpha\,|x_i|), \; i \in E. \end{aligned} \tag{13}$$

However, we are not aware of a way of solving (13) efficiently in parallel.

The optimal model (3) can be generalised by assigning a weight w_{ij} to each edge of the processor graph, where $w_{ij} > 0$ is a weighting factor representing the penalty of communication between processors i and j. The quantity to be minimised becomes $\frac{1}{2}x^T W^2 x$, with W the diagonal matrix of weights. Then equations (4) and (5) become

$$x = W^{-2}A^T\lambda$$

and

$$AW^{-2}A^T\lambda = b.$$

Here $AW^{-2}A^T$ can be computed as

$$(AW^{-2}A^T)_{ij} = \begin{cases} -\frac{1}{w_{ij}^2}, & \text{if } \; i \neq j, \; (i,j) \in E, \\ \sum_{\{k \;|\; (i,k)\in E\}} \frac{1}{w_{ik}^2}, & \text{if } \; i = j, \\ 0, & \text{otherwise.} \end{cases}$$

ACKNOWLEDGMENTS

We would like to thank Chris Walshaw for supplying processor graphs and loads related to the Tri60K and Tet100K meshes. We would also like to thank the referees for valuable remarks.

References

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

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

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

[4] S. T. Barnard and H. D. Simon, 'Fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems', *Concurrency: Practice and Experience*, **6**, 101-117 (1994).

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

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

[7] 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).

[8] B. Hendrickson and R. Leland, *The Chaco User's Guide*, Version 1.0, Technical Report SAND 93-2339, Sandia National Laboratories, Allbuquerque, NM (1993).

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

[10] P. Diniz, S. Plimpton, B. Hendrickson and R. Leland, 'Parallel algorithms for dynamically partitioning unstructured grids', in *SIAM Proceedings Series* 1995, ch. 195, eds. D.H. Bailey, P. E. Bjorstad, Jr Gilbert, M. V. Mascagni, R. S. Schreiber, H. D. Simon, V. J. Torczon, J. T. Watson, SIAM, Philadelphia, 615-620 (1995).

[11] G. Karypis and V. Kumar, 'Parallel multilevel graph partitioning', Technical Report, Department of Computer Science, University of Minnesota, Minneapolis, MN 55455 (1995).

[12] G. Cybenko, 'Dynamic load balancing for distributed memory multiprocessors', *Journal of Parallel and Distributed Computing* , **7**, 279-301 (1989).

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

[14] 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**, 385-393 (1992).

[15] 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**, 72-85 (1995).

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

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

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

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

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

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

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

[23] N. Briggs, *Algebraic Graph Theory*, Cambridge University Press, Cambridge (1974).

Table 1 The number of iterations (and in brackets, time for convergence, in milli-seconds) for the new algorithm and the diffusion algorithm, on randomly generated graphs

p	diameter	degree	new algorithm	diffusion algorithm
8	6	2	7 (3.3)	87 (16.0)
8	3	3	7 (4.0)	25 (6.5)
8	2	5	5 (3.2)	13 (4.5)
8	1	7	1 (1.4)	5 (1.9)
16	11	2	14 (7.5)	305 (60.0)
16	5	3	11 (7.0)	48 (13.1)
16	3	5	7 (5.2)	17 (5.9)
16	2	7	6 (5.3)	13 (6.3)
16	2	9	5 (5.0)	10 (5.6)
32	24	2	29 (17.0)	923 (182.4)
32	8	3	17 (11.5)	122 (33.3)
32	4	5	10 (8.3)	32 (11.4)
32	3	7	8 (7.1)	20 (9.2)
32	3	9	7 (6.8)	15 (8.4)
64	47	2	59 (39.6)	2842 (628.6)
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)

Table 2 The number of iterations (and in brackets, time for convergence, in milli-seconds) for the new algorithm and the diffusion algorithm, on processor graphs resulted from two meshes

p	diameter	degree	new algorithm	diffusion algorithm
Tri60K mesh				
16	8	3.25	11(7.3)	45(11.8)
32	9	3.88	17(13.1)	140(40.6)
64	14	4.44	12(11.0)	25(9.4)
128	20	4.77	30(28.6)	520(186.9)
256	30	5.03	41(43.8)	670(263.7)
Tet100K mesh				
16	6	3.75	14(9.5)	55(16.6)
32	7	5.88	16(14.1)	70(30.3)
64	10	5.94	22(21.4)	155(72.7)
128	11	7.63	27(32.8)	240(156.2)
256	14	7.23	34(41.6)	285(169.4)