A Parallelisable Algorithm for Optimising Unstructured Mesh Partitions

C. Walshaw, M. Cross and M. G. Everett *

Mathematics Research Report

January '95

Abstract

A new method is described for optimising graph partitions which arise in mapping unstructured mesh calculations to parallel computers. The method employs a combination of iterative techniques to both evenly balance the workload and minimise the number and volume of interprocessor communications. It is designed to work efficiently in parallel as well as sequentially and can be applied directly to dynamically refined meshes. In addition, when combined with a fast direct partitioning technique (such as the Greedy algorithm) to give an initial partition, the resulting two-stage process proves itself to be both a powerful and flexible solution to the static graph-partitioning problem. A clustering technique can also be employed to speed up the whole process. Experiments, on graphs with up to a million nodes, indicate that the resulting code is up to an order of magnitude faster than existing state-of-the-art techniques such as Multilevel Recursive Spectral Bisection, whilst providing partitions of equivalent quality.

Key words. graph-partitioning, unstructured meshes, load-balancing, parallel scientific computation.

1 Introduction

The use of unstructured mesh codes on parallel machines is one of the most efficient ways to solve large Computational Fluid Dynamics (CFD) and Computational Mechanics (CM) problems. Completely general geometries and complex behaviour can be readily modelled and, in principle, the inherent sparsity of many such problems can be exploited to obtain excellent parallel efficiencies. However, unlike their structured counterparts, one must carefully address the problem of distributing the mesh across the memory of the machine at runtime so that the computational load is evenly balanced and the amount of interprocessor communication is minimised. It is well known that this problem is NP complete, [8], so in recent years much attention has been focused on developing suitable heuristics, and some powerful methods, many based on a graph corresponding to the communication requirements of the mesh, have been devised, e.g. [5]. Closely related to this graph partitioning problem is the problem of optimising existing mesh partitions and in this paper we discuss the partition optimisation problem and its bearing on the graph partitioning problem.

1.1 Dynamic partitioning

An extremely important area for partition optimisation arises from time-dependent unstructured mesh codes which adaptively refine the mesh. This is a very efficient way to track phenomena which traverse the

* *School of Mathematics, Statistics & Scientific Computing, University of Greenwich, London, SE18 6PF, UK.* **e-mail:** C.Walshaw@gre.ac.uk

solution domain but means that the position and density of the mesh points may vary dramatically over the course of an integration.

From the point of view of partitioning this has three major influences on possible solution techniques; cost, reuse and parallelism. Firstly, the unstructured mesh may be modified every few time-steps and so the load-balancing must have a low cost relative to that of the solution algorithm in between remeshing. This may seem to restrict us to computationally cheap algorithms but fortunately help is at hand if the mesh has not changed too much, for in this case it is a simple matter to interpolate the existing partition from the old mesh to the new and use this as the starting point for repartitioning, [24]. In fact, not only is the load-balancing likely to be unnecessarily computationally expensive if it fails to use this information, but also the mesh elements will be redistributed without any reference to their previous 'home processor' and heavy data migration may result. The third factor to take into account is that the data is distributed and so should be repartitioned *in situ* rather than incurring the expense of transferring it back to some host processor for load-balancing and Löhner *et al.* have advanced some powerful arguments in support of this proposition, [14]. Collectively these issues call for a parallel partition optimisation algorithm.

1.2 Static Partitioning

The development of successful partition optimisation algorithms also has an important side effect on the static partitioning problem. Previously, authors have concentrated on direct methods, [1, 4]; however, the combination of a cheap direct algorithm together with a powerful optimisation technique can often derive partitions faster and better than those given by more sophisticated direct methods, [5, 20, 21]. Also this sort of two-stage process can be made far more flexible by varying the amount of optimisation carried out dependent on the problem parameters.

In addition, as mesh and machine sizes grow the need for parallel mesh partitioning becomes increasingly acute. For small meshes and machines, an $O(N)$ overhead for the mesh partitioning may be considered reasonable. However, for large N, this order of overhead will rapidly become unacceptable if the solver is running at $O(N/P)$. Clearly, to carry out partitioning in parallel, the data must first be distributed across the machines memory and so deriving a partition as quickly as possible, distributing the data and then *optimising* the partition *in parallel* is a natural strategy.

This multi-stage approach is similar to the work of Vanderstraeten, [20], although the techniques vary in that we employ deterministic heuristics to optimise the partition. The philosophy behind the optimisation algorithm is close to that of Löhner *et al.*, [14], although in addition we employ a heuristic which attempts to improve the 'shape' of the subdomains.

2 Notation and Definitions

We use a graph to represent the data dependencies in the mesh arising from the discretisation of the domain. Thus, let $G = G(N, E)$ be an undirected graph of N nodes & E edges and P be a set of processors. In a slight abuse of notation we use N, E & P to represent both the *sets* of nodes, edges & processors and the *number* of nodes, edges & processors, with the meaning clear from the context. We assume that both nodes and edges are weighted (with positive integer values) and that $|n|$ denotes the weight of a node n; $|(n,m)|$ the weight of an edge; $|S| := \sum_{n \in S} |n|$ the weight of a subset $S \subset N$; and $|T| := \sum_{(n,m) \in T} |(n,m)|$ the weight of a subset $T \subset E$. We define $\mathcal{P} : N \to P$ to be a partition of N and denote the resulting subdomains by S_p, for $p \in P$. The optimal subdomain weight is given by $W := \lceil |N|/P \rceil$. We denote the set of cut (or inter-subdomain) edges by E_c and the border of each subdomain, B_p, is defined as the set of nodes in S_p which have an edge in E_c.

We shall use the notation $\leftrightarrow$ to mean 'is adjacent to'. For example, for $n, m \in N$, $n \leftrightarrow m$ if $\exists (n,m) \in E$. Also if $n \in N$ and $S, T \subset N$, $n \leftrightarrow S$ if $\exists (n,m) \in E$ with $m \in S$ and similarly $S \leftrightarrow T$ if $\exists (n,m) \in E$ with $n \in S$ &

Notation	Name	Definition
$\lvert.\rvert$	weight function	
$\lceil.\rceil$	ceiling function	$\lceil x \rceil :=$ smallest integer $\geq x$
$\leftrightarrow$	adjacency operator	$n \leftrightarrow m \Leftrightarrow \exists(n,m) \in E$
$\Gamma(S)$	neighbourhood function	$\{m \in N - S : m \leftrightarrow S\}$
$\mathcal{P} : N \rightarrow P$	partition function	
E_c	cut edges	$\{(n,m) \in E : \mathcal{P}(n) \neq \mathcal{P}(m)\}$
S_p	subdomain p	$\{n \in N : \mathcal{P}(n) = p\}$
B_p	border of subdomain p	$\{n \in S_p : \exists(n,m) \in E_c\}$
H_p	halo of subdomain p	$\Gamma(S_p) = \{m \in N - S_p : m \leftrightarrow S_p\}$
W	optimal subdomain weight	$\lceil \lvert N\rvert / P \rceil$

Table 1: Definitions

$m \in T$. We then denote the neighbourhood of a set of nodes S (the nodes adjacent to S) by $\Gamma(S)$ and thus the halo of subdomain S_p by $H_p := \Gamma(S_p)$. Table 1 gives a summary of these definitions.

We assume throughout the rest of this paper that the graph is connected (i.e. for all $n, m \in N$ there exists a path of edges from n to m). If this is not the case there are several possible options, such as running the partitioning code on each connected subgraph or even adding edges of zero weight to the graph to make it connected.

2.1 The graph-partitioning problem

The definition of the graph-partitioning problem is to find a partition which evenly balances the load or node weight on each processor whilst minimising the communications cost. More precisely we seek $\mathcal{P}$ such that $S_p \leq W$ for $p \in P$ and such that $|E_c|$ is approximately minimised. The definition of the partition optimisation problem is the same but working from an initial partition $\mathcal{P}_0$. It is a matter of contention whether $|E_c|$ is the most important metric for minimisation but see §5.1 for a further discussion on this topic.

Note that in general, for graphs arising from finite element or finite volume discretisations, the degree of each node is low. As a result the surface of solutions of the mesh-partitioning problem tends to have many local minima and can have several optimal solutions which are far apart.

2.2 Localisation and the processor graph

An important aim for any parallel algorithm is to keep communication as localised as possible to avoid contention and expensive global operations; this issue becomes increasingly important as machine sizes grow. Throughout the optimisation algorithm we localise the communication with respect to the partition $\mathcal{P}$ by only requiring subdomains to communicate with neighbouring subdomains. Thus a processor p will only need to communicate to q if there is an edge $(n, m) \in E_c$ with $\mathcal{P}(n) = p$ and $\mathcal{P}(m) = q$.

In this way the partition $\mathcal{P}$ induces a processor graph on G which we shall refer to as $\mathcal{P}(G) = \mathcal{P}(G)(P, C)$, an undirected graph of P processors (nodes) and C connections (edges). There is an edge $(p, q) \in C$ if $S_p \leftrightarrow S_q$. In addition, the weight on each processing node $p \in P$ is given by $|p| = |S_p|$ and the weight of a connection is given by $|(p,q)| = \sum_{\{(n,m)\in E: n\in S_p, m\in S_q\}} |(n,m)|$. Thus $|P| = \sum_{p\in P} |S_p| = |N|$ and $|C| = |E_c|$. Figure 1 gives an example of a partitioned graph (with unit node and edge weights) and the resulting processor graph (with node and edge weights as shown).

Note that this processor graph may bear no direct relationship to the physical processor topology of the

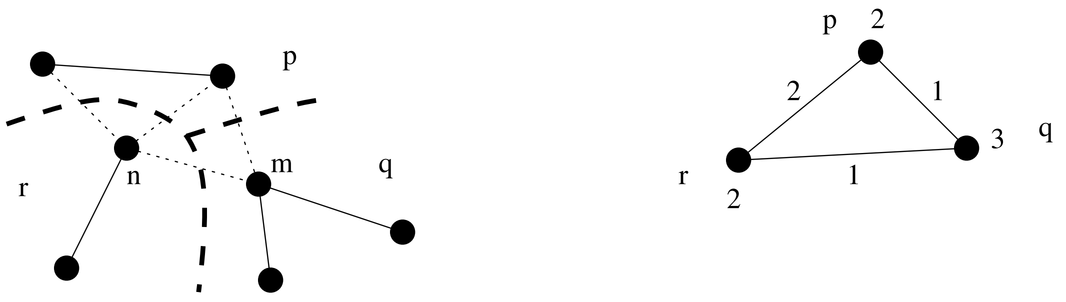

Figure 1: An example graph and the resulting processor graph (node & edge weights as shown)

parallel machine, although the algorithm should be more efficient if this is the case. For the purposes of this paper, however, we shall assume that the processors are uniformly connected – that is, point-to-point communication carries a uniform latency no matter which processors are communicating. This is a realistic assumption for certain machines with small or moderate numbers of processors but for other situations it may be possible to optimise the partition for the machine topology, [23].

3 The parallel optimisation method

In this section we present three complementary iterative algorithms which combined together form a powerful and flexible technique for optimising unstructured mesh partitions. Initially the subdomain heuristic attempts to 'improve' the 'shape' of the subdomains. However, this heuristic cannot guarantee load-balance and so a second heuristic is applied to share the workload equally between all processors. Finally a localised version of the Kernighan-Lin algorithm, [12], is applied to minimise the communication cost.

3.1 Migration and the gain and preference functions

Throughout each of the three phases, it is assumed that the final partition will not deviate too far from the initial one. Thus, in general, only border nodes are allowed to migrate to neighbouring subdomains.

For these purposes, at each iteration we calculate the gain and preference values for every border node. Loosely, the gain $g(n,q)$ of a node $n \in B_p$ can be calculated for each adjacent subdomain, S_q, and expresses some 'estimate' of how much the partition would be 'improved' were n to migrate to S_q. The preference $f(n)$ is then just the *adjacent* subdomain which maximises the gain of a node – i.e. $f(n) = q$ where q attains $\max_{r \in P} g(n,r)$. Most nodes in B_p will only be adjacent to one subdomain, q say, and in this case the preference is simply q. Where 3 or more subdomains meet, the preferences need to be explicitly calculated and if more than one subdomain attains the maximum gain, a pseudo-random choice is made, §3.4.

Note that the definition of the gain differs for each phase of the algorithm. For example, when applying load-balancing and local refinement, §3.3 & §3.4, the gain $g(n,q)$ just expresses the reduction in the cut-edge weight. Thus in Figure 1 (assuming unit edge weights) there would be one less edge cut if node n migrated to subdomain p and so $g(n,p) = 1$. Similarly $g(n,q) = 0$ and hence $f(n) = p$. For node m, $g(m,p) = -1$ and $g(m,r) = -1$ and so a pseudo-random choice must be made for $f(m)$. Note that here we do not allow the preference of a node to be its current processor since we may have to migrate nodes even if the net gain is negative.

The only time that internal nodes migrate is in the first phase, §3.2, when the heuristic can detect small subsets of the subdomain which are disconnected from the main body. In this case the gain of internal and border nodes in the disconnected subset is just some arbitrarily large number; the preference is the nearest adjacent subdomain (in a graph sense).

3.2 The subdomain heuristic

3.2.1 Motivation

One of the problems of partition optimisation is that of local minima traps. Algorithms such as the Kernighan-Lin heuristic, [12], which can be very successful at minimising the number of cut edges at a local level, do not fully address the global problem.

A partial solution to this problem has been the introduction of non-deterministic heuristics such as simulated annealing, e.g. [13, 25]. They address local minima traps by allowing transfers of graph nodes between subdomains *even if* the communication cost is increased, the transfer being accepted according to some probability based on a cooling schedule. These algorithms can be very effective, especially if the initial partition is good but they can suffer from high run times.

The new heuristic introduced here has been designed according to two constraints. We want the algorithm to:

(a) address the optimisation at a subdomain level to try and attain a *global* minimum;

(b) carry out the optimisation *locally* to give an efficient parallel algorithm.

At first these two constraints may appear to frustrate each other. However, a useful analogy with a simple schoolroom experiment can be drawn. The experiment consists of bubbling a gas through a pipette into a detergent solution and the result is that the bubbles make a regular (hexagonal) pattern on the surface of the liquid. They achieve their regularity without any global 'knowledge' of the shape of the container or even of any bubbles not in immediate contact; it is simply achieved by each bubble minimising its own surface energy.

Translating this analogy to a partitioned graph we see that each subdomain must try to minimise its own surface area. In the physical 2D or 3D world the object with the smallest surface to volume ratio is the circle or sphere. Thus the idea behind the domain-level heuristic is to determine the centre of each subdomain (in some graph sense) and to then measure the radial distance from the centre to the edges and attempt to minimise this by migrating nodes which are furthest from the centre.

3.2.2 Implementation

Determining the 'centre' of the subdomain is relatively easy and can be achieved by moving in level sets inwards from the subdomain border until all the nodes in the subdomain have been visited. More precisely, for each subdomain S_p, define a series of (disjoint) level sets, $L_0 = B_p$; $L_1 = \Gamma(L_0) \bigcap S_p$; $\ldots L_i = \Gamma(L_{i-1})$; etc. ... The final non-empty set, L_c, of this series defines the centre of the subdomain. Note that if the graph is connected (assumed) the level sets will completely cover the subdomain, i.e. $S_p = \bigcup_0^c L_i$, but that the centre may not be a connected set of nodes.

The reverse of this process can then be used to determine the radial distance of nodes from the centre. Thus $M_0 = L_c$; $M_1 = \Gamma(M_0) \bigcap S_p$; $\ldots M_i = \Gamma(M_{i-1}) \bigcap S_p$; etc. ... It is easy to see (for example, Figure 2) that the mapping to level sets is not necessarily the same as that given by the inwards search for the centre. Also, if S_p is not connected, then the level sets may not cover the subdomain, i.e. $\bigcup M_i \neq S_p$.

Having derived these sets each node n can be marked by its radial distance, $c - i$ for $n \in M_i$. Nodes in S_p which are not connected to L_c are not marked. This is useful for migrating small disconnected parts of

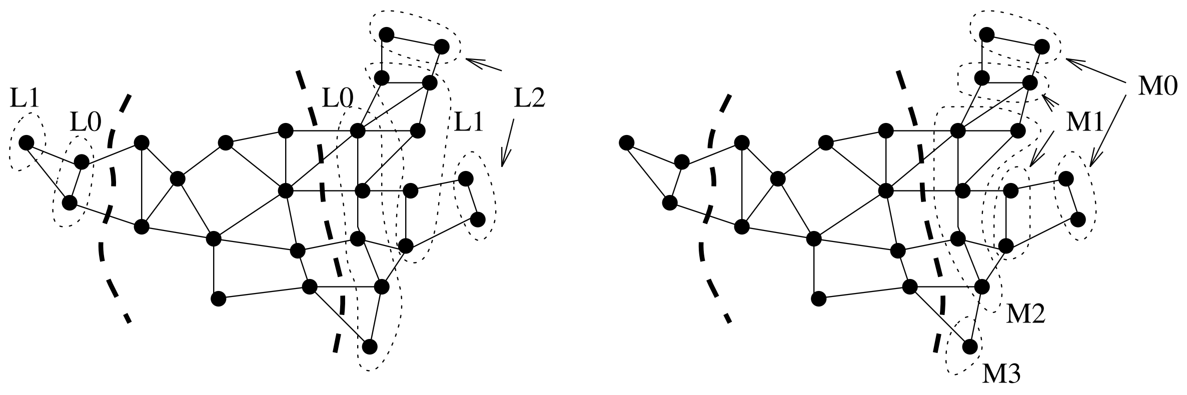

Figure 2: An example of the level sets moving in from the boundary, L0, L1 & L2, and moving out from the centre, M0, M1, M2 & M3

the subdomain to more appropriate processors, although it should be noted that the centre is essentially determined by the diameter of each connected subset and not its size. Thus it is possible that a subset of the subdomain may contain more than half the weight of the subdomain but still be unmarked. Figure 2 shows a simple example of these level sets for a disconnected subdomain.

Whilst it is fairly easy to determined the radial distance for each node, the gain function must be derived empirically. Although we use separate load-balancing and local refinement heuristics, §3.3 & §3.4, it is inefficient to achieve some sort of steady-state with the subdomain heuristic only to destroy it later. For this reason we encode imbalance and communication minimisation information in the gain at this stage. We first define an imbalance adjustment for every domain by

$$\delta_p := 20\frac{(W - |S_p|)}{|B_p|} \times \frac{|N|}{|E|}$$

and add it to the radial distance to give an adjusted radial distance, $\phi(n) = c - i + \delta_p$ for $n \in M_i \subset S_p$. This imbalance adjustment has been arrived at by experimentation and loosely expresses the amount of weight a subdomain can hope to gain or lose according to the size of its border and the average connectivity of the graph. We then define the gain on a node $n \in B_p$ by

$$g(n,p) = \phi(n) + \sum_{m \in S_p \cap \Gamma(n)} \phi(m) + \theta|(n,m)|$$

$$g(n,q) = \sum_{m \in S_q \cap \Gamma(n)} \phi(m) + \theta|(n,m)| \qquad p \neq q.$$

Again the preference $f(n)$ is just the *adjacent* subdomain which maximises the gain of a node and for the subdomain heuristic we migrate all nodes with positive gain.

Here θ expresses the relative influence between the radial distance, which is used to 'improve' domain shape, and the edge weight, which is used to minimise $|E_c|$. We have found a very successful policy is to gradually increment θ throughout the run. This has a dual effect, strongly reminiscent of the cooling schedule in simulated annealing, [13], which both slows down the migration towards a steady-state and which transfers the emphasis from global to local optima. For the experiments described in §5.3 we started each run with $\theta = 1$ and incremented it by 1 after every 4 iterations (except after using graph coarsening and having interpolated to the full graph – configuration 'JOSTLE/reduction' – when we incremented θ by 1 every iteration).

3.3 Load-balancing

The load-balancing problem, i.e. how to distribute N tasks over a network of P processors so that none have more than $\lceil N/P \rceil$, is a very important area for research in its own right with a vast range of applications. The topic is comprehensively introduced in [17] and some common strategies described. In particular, much work has been carried out on parallel or distributed algorithms recently, e.g. [3], and an important feature of the problem is whether the information on the current load-state is gathered globally or locally – the former enables an algorithm to converge faster but may carry a high communication overhead.

Here we use a localised iterative algorithm for distributed load-balancing devised by Song, [18]. It builds on a proposition proved by Bertsekas and Tsitsiklis, [2, pp 521], which states that a transfer policy, devised under certain easily satisfied conditions, will converge as $t \to \infty$. In common with the rest of our optimisation, the localised nature means that it only requires information from, and migrates nodes to, its neighbours in the processor graph. The description of the algorithm assumes that the workload consists of independent tasks of equal size and is proven to iterate to convergence with a maximum load-difference of $d/2$ where d is the diameter of the processor graph.

Translating this to the mesh partitioning problem, we use Song's algorithm to determine the *number* of nodes to migrate from a given processor to each of its neighbours and then use the gain and preference functions, §3.1, to determine *which* nodes to move. Although the algorithm does not guarantee perfect load-balance, consider for example the Tri60K mesh in §5.3 of 60,005 nodes on 64 processors. The maximum diameter possible (which arises if the processor graph is a 1D linear array or chain) is 63. The optimal load is 938 and so the worst case deviation from this figure is 8/938 = 0.85%. Of course the finer the granularity the more exaggerated this effect can be, but experience suggests the imbalance is usually far from the worst possible and usually insignificant.

Conceptually this algorithm is ideally suited to our application; all information about the load-state is gathered locally and all transfers are between neighbouring processors. Unfortunately, however, it can suffer from very slow convergence, typically $O(P)$ and in particular the rate of convergence is directly proportional to the load-imbalance. For example, in one randomly generated processor graph of 233 nodes the algorithm took 100 iterations to go from 106.1% to 4.0% load-imbalance and then a further 200 iterations to converge while reducing the load-imbalance to 2.1%. We consider this to be too high an overhead, particularly as most of the final iterations only involved transfers of one or two nodes at a time.

Fortunately this effect can be easily countered by running the algorithm to completion based on the current load-state but without actually carrying out the transfers. The accumulated results can then be aggregated and used as a migration schedule. The only qualification is that if the processor graph loses a connection scheduled for migrating nodes, the algorithm must be rerun and the schedule revised.

Currently we use this technique for sequential partitioning and we also propose to use it in the parallel implementation, although we are investigating other alternatives such as diffusion type schemes, e.g. [3]. The problem of calculating the migration-schedule in parallel can be solved by broadcasting the processor graph around the network and replicating the calculation on every processor. This is not ideal because of the global communication involved, but each processor need only broadcast its current load plus a sparse vector listing its neighbours (or half of its neighbours since the connections are symmetric).

The gain and preference functions for load-balancing are as described in §3.1. The border nodes are sorted by gain and the load specified by the schedule migrated to preferred subdomains. Of course a subdomain may not have enough nodes along its boundary to satisfy the amount of migration chosen by the load-balancing algorithm, but this will not break the conditions of the proposition provided at least one node is transferred to the most lightly loaded neighbour at each iteration. Also the fact that, due to node migration, the processor graph may change from iteration to iteration does not seem to significantly affect the convergence.

3.4 Local partition refinement

Having achieved optimal (or near optimal) load-balance it may still be possible to move nodes around the processor network to further minimise the number of cut edges whilst retaining the load-balance. An algorithm which comes immediately to mind for this purpose is the Kernighan-Lin (KL) heuristic, which maintains load-balance by employing pairwise exchanges of nodes. Unfortunately it has $O(n^2 \log n)$ complexity but a linear-time variant which delivers similar results has been proposed by Fiduccia & Mattheyses (FM), [6]. The FM algorithm achieves this reduction partially by calculating swaps one node at a time rather than in pairs. Both algorithms have the possibility of escaping local minima traps by continuing to compute swaps even if the swap gain is negative in the hope that some later total gain may be positive. Here we introduce an algorithm largely inspired by the KL/FM algorithms but with several modifications to better suit our purposes.

3.4.1 Motivation

In common with Hendrickson & Leland, [9], we remark that the gains of each node do not have to be recalculated at each pass of the algorithm – having calculated them initially, we need only recalculate gains for migrating nodes and their immediate neighbours. However we extend the simplification by noting that, since edge weights are always positive integers, any node internal to a subdomain (i.e. $\in S_p - B_p$) will have negative gain. Although it is easy to devise arbitrary graphs in which an internal node may have maximum gain it seems to very occur very rarely for our sort of applications (for example, for a cell-centred finite volume triangular discretisation with unit edge weights the *maximum* gain for an internal node is 1 as is the *minimum* gain for a border node). Thus we need only calculate initial gains for border nodes and thereafter for nodes coming in to the border or potential border. Of course, the full algorithm continues to evaluate swaps even when the step gain is negative and if it explores the entire graph in this way it must eventually calculate the gain of every node. However, if as suggested by several authors, [7, 9], we terminate this process at some early stage, it is very likely that much of the graph will remain unexplored. This can make the whole algorithm cost sublinear and potentially insignificant for very coarse granularities.

A second simplification arises in extending the algorithm to arbitrary P. Previously authors, [9, 19], have implemented this by employing pairwise exchanges (e.g. for 4 processors, 1 pairs with 2 & 3 pairs with 4; then 1 with 3 & 2 with 4 and finally 1 with 4 & 2 with 3). However we note that it is unlikely that an overall gain will accrue by transferring a node to a subdomain which it is not adjacent to in the processor graph. Thus we only consider exchanges between neighbouring subdomains.

For either algorithm, the swapping of nodes between two sub-domains is an inherently non-parallel operation and hence there are some difficulties in arriving at efficient parallel versions, [16]. In a naive parallel implementation one could imagine a pair of processors alternately sending each other one node at a time – effective but horrendously inefficient. A second difficulty arises from the fact that the algorithms treat bisections and so processors must exchange in pairs. If, however, a processor is only going to exchange nodes with its neighbours, it might seem that we have to derive a schedule based on the processor graph – potentially an expensive operation. In fact, we overcome both of these problems by replicating the transfer decision process on both sides of each sub-domain interface. Thus a processor p examines both its border and halo nodes to decide *not only* which of its nodes should migrate to neighbour q, *but also* which nodes should transfer from q to p. It does not use the information about transfers of nodes that it does not own, but the duplication of this transfer scheme allows any pair of processors to arrive *independently* at the same optimisation. Although this may be computationally inefficient (because the workload is doubled) it is almost certainly much more efficient than a 'hand-shaking' communication for each pair of nodes. It also avoids the need for pairwise exchanges between neighbouring processors.

3.4.2 Implementation

At each iteration a processor, p, calculates the preference and gain of its own border nodes (see 3.1) and a halo update is carried out to flush these values around the network. Next the processor examines the interfaces with its neighbours and for each p-q interface creates a list of border nodes n which have preference $f(n) = q$ and a list of halo nodes m which have preference $f(m) = p$. It then chooses a starting list – either that containing the node with the highest gain or, if both are equal, the longest list or failing that a random choice. Alternating between the lists, each list is sorted and the node with highest gain is marked for migration. It is then removed from the list and its neighbours in either list have their gains adjusted as if it had migrated. The process continues while the sum of the gains of the two nodes selected that step is > 0 (note that zero gain swaps can lead to cyclic behaviour) and the last two nodes selected are re-marked as not migrating. Having carried out this selection process for all of its neighbours, a processor migrates all of its border nodes that have been selected, receives migrating nodes from other processors and re-evaluates its border.

There are some important points to be noted about this method. Firstly, whenever a processor makes a decision about sorting (or randomly selecting the starting list) it is *essential* that its neighbour makes the same choice; otherwise they might reach different migration schemes and increase the cut-edge weight. Thus when sorting, if two nodes have the same gain, they are sorted on some other factor (and for this reason it is of no benefit to use the bucket sort of Fiduccia and Mattheyses). One possibility is the global index of each, to which each processor presumably has access. However this might highlight some feature of the mesh, so as an alternative, for each node, we seed a random number generator with its index and then generate a pseudo-random index. For a similar reason, when adjusting the gain of nodes in B_p on the p-q interface, we do not adjust those nodes which have a different preference r as this could have a knock-on effect on the migration scheme for p which would not be seen by processors q and r.

For adjusting the gain of nodes in the halo H_p, note that processor p needs to know about edges between nodes within the halo, something which is not necessary for phases 1 & 2 of the optimisation. In fact one solution is to augment the halo to a double layer of nodes around the the sub-domain and in this case each processor can calculate all the gains and preferences of its halo nodes without any need for a halo update. We shall study the relative costs and efficiency of either case when we implement the algorithm in parallel.

4 Experimental results

4.1 The code testbed

The software tool written at Greenwich to implement the new optimisation technique is known as JOSTLE. This is not an acronym; rather it reflects the way the subdomains jostle one another to reach a steady-state. It is modular in nature and consists of three parts; the initial partitioning module, the partition optimisation module and the graph coarsening module. These modules are designed for interchangeable use. Currently the code runs sequentially and in this case one might coarsen the graph, partition the reduced graph, optimise the partition, interpolate onto the full graph and reoptimise. With parallel code one might partition the full graph sequentially, distribute the graph, coarsen each subdomain, optimise the partition of the reduced graph, interpolate and reoptimise.

4.1.1 The initial partition

The aim of the initial partitioning code is to divide up the graph as rapidly as possible in order that it can be distributed and the partition optimised in parallel and the code therefore utilises a version of the Greedy algorithm [4]. This is clearly seen to be the fastest *graph-based* method as it only visits each graph edge once. The variant employed here differs from that proposed by Farhat only in that it works solely with a graph rather than the nodes and elements of a finite element mesh.

4.1.2 Clustering: reducing the problem size

For coarse granularity partitions it is inefficient to apply the optimisation techniques to every graph node as most will be internal to the subdomains. A simple technique to speed up the load-balancing process, therefore, is to group nodes together to form *clusters*, use the clusters to define a new graph, recursively iterate this procedure until the graph size falls below some threshold and then apply the partitioning algorithm to this reduced size graph. This is quite a common technique and has been used by several authors in various ways – for example, in a multilevel way analogous to multigrid techniques, [1], and in an adaptive way analogous to dynamic refinement techniques, [24].

The technique used here for graph reduction is another variant of the Greedy algorithm [4], although various algorithms have been successfully employed, [1, 9, 11]. It is used recursively to cluster nodes into small groups, each of which defines a node of the reduced graph. It is, of course, important that the groups of nodes are connected (in order to retain the features of the original graph) and so we relax the condition that each cluster should contain the same node weight in favour of guaranteeing that the nodes of each cluster form a connected set. The node and edge weights for the reduced graph derive simply from the sum of node weights over the cluster and sum of edge weights from the cluster to other clusters. The algorithm (which can be easily parallelised) is described more fully in [21]

5 The experiments

The following experiments were mostly carried out on a Silicon Graphics Indigo 2 with a 150 MHz CPU and 64 Mbytes of memory. The final set of results for the largest mesh came from a Sun SPARC station with a 50 MHz CPU and 224 Mbytes of memory; typically this processor is about 50% slower than the Silicon Graphics machine but the memory requirements forced its usage. The code has also been compiled and run on an IBM RS6000 with similar timing ratios. Further results can be found in [21, 22].

We include results for runs on 4 different graphs. The Barth5 mesh (which is available by anonymous ftp from `riacs.edu/pub/grids` and has previously been used for benchmarking partitioning algorithms, [1, 9]) is a moderate sized ($N = 15,606$, $E = 45,878$) finite-element mesh around a 4- element airfoil. Tri60K is a two-dimensional finite-volume mesh ($N = 60,005$, $E = 89,440$) arising from a casting simulation, [15]. Tet100K and Tet1M are large three-dimensional finite-volume meshes ($N = 103,081$, $E = 200,976$; $N = 1,119,663$, $E = 2,212,012$ respectively) in the shape of a Y standing on a baseplate.

5.1 Metrics

We use four metrics to measure the performance of the algorithms. Three of these are concerned with the partition quality – the total weight of cut edges, $|E_c|$, the maximum percentage load-imbalance, δ (defined below), and the average processor degree, $D_a := \{\sum_{p\in P} \deg(p)\}/P$. The fourth is $t(s)$ the execution time in seconds of each algorithm.

Unfortunately, there are no *ideal* metrics for assessing partition quality as the parallel efficiency of the problem from which the mesh arises will depend on many things – typically the machine (size, architecture, latency, bandwidth and flop rate), the solution algorithm (explicit, implicit with direct linear solution, implicit with iterative linear solution) and the problem itself (size, no. of iterations) all play a part. However, $|E_c|$ gives a rough indication of the volume of communication traffic and the average degree of the processor graph, D_a, gives an indication of the number of messages each processor must send. From a load-balancing point of view the worst percentage imbalance is derived from $\delta := 100 \times (\max_{p\in P} |S_p| - W)/W\%$, since the speed of the calculation is determined by the processor with the largest workload, not by underloaded processors.

5.2 Example runs

Phase	iterations	$\lvert E_c \rvert$	$t(s)$	$\delta\%$	D_a
initial partition	–	3834	0.76	0.00	5.88
energy minimisation	41	2384	12.32	6.08	4.81
load balancing	26	2927	0.48	0.64	4.81
local refinement	4	2556	0.10	0.64	4.84

Table 2: Results for the Tri60K mesh: $P = 64$, $N = 60,005$, $E = 89,440$

To give a flavour of the sort of results coming from the JOSTLE code we show two typical runs. Table 2 gives, for the Tri60K mesh and $P = 64$, a breakdown of the results obtained after the initial partition and after each of the three optimisation phases. Because both the load-balancing and local refinement phases deal only with border nodes, most of the time is spent in the energy minimisation part of the optimisation where all of the graph edges are visited at each iteration (except where a subdomain hasn't changed and values from the previous iteration are used). Here we see that the subdomain heuristic does a very good job of minimising $|E_c|$ although the load-balance is not ideal and future work will look at trying to improve the balancing part of this phase.

phase	graph	iterations	$t(s)$	$\lvert E_c \rvert$	$\delta\%$	D_a
graph coarsening	$G(60005, 89440)$	2	1.13	–	–	–
initial partition	$g(4064, 11147)$	–	0.03	4628	0.96	5.50
optimisation	$g(4064, 11147)$	48	2.58	3305	3.62	4.59
optimisation	$G(60005, 89440)$	14	3.10	2432	0.32	4.62

Table 3: Results for the Tri60K mesh: $P = 64$, $N = 60,005$, $E = 89,440$

Table 3 shows (in less detail) the same run using graph coarsening. Having coarsened the graph G to create a reduced graph $g(N = 4,064; E = 11,147)$, a partition is generated and optimised. The partition is then interpolated onto the full graph and reoptimised. As can be seen, most of the iterations take place on g and as a result, the runtime is substantially reduced. Perfect load-balance is not attained for g, however, because the clusters do not have uniform node weights and the $|E_c|$ figure is not greatly optimised because the clusters have fairly irregular shapes. For this reason it is well worth doing some further optimisations on the full graph G and we see that the load is nearly balanced and $|E_c|$ cut by 30% in just 14 iterations.

Comparing the two examples, graph coarsening reduces the overall run time by 50% and in addition, for this example the $|E_c|$ figure is actually lower with coarsening. This is somewhat surprising and it is the authors' experience that the reverse is typical, with the solution quality reflecting the amount of work that is put in to achieve it.

5.3 Comparative results

In order to demonstrate the quality of the partitions we have compared the method with two of the most popular partitioning algorithms, Greedy and Multilevel Recursive Spectral Bisection (MRSB). The Greedy algorithm, [4], is actually performed as part of the JOSTLE code and is fast but not particularly good at minimising cut edges. MRSB, on the other hand, is a highly sophisticated method, good at minimising $|E_c|$ but suffering from relatively high runtimes, [1]. The MRSB code was made available to us by one of its authors, Horst Simon, and run unchanged with contraction thresholds of 100 and 500.

The JOSTLE code can be run in three different configurations, each easily specified to the code in a defaults file. Two of these use the graph coarsening, §4.1.2; 'JOSTLE/fast reduction' uses the full algorithm as described on the reduced graph and then only uses the load-balancing and local-refinement on the full graph; 'JOSTLE/reduction' additionally uses the subdomain heuristic, §3.2, although with a cut off after 5 iterations. The other configuration, 'JOSTLE', does not use any graph coarsening.

Method	$\lvert E_c \rvert$	$t(s)$	$\delta\%$	D_a
GREEDY	4046	0.29	0.00	5.34
JOSTLE/fast rdtn	2966	1.86	0.82	4.59
JOSTLE/reduction	2949	2.88	1.23	4.59
JOSTLE	2972	5.11	1.23	4.62
MRSB/100	3122	11.59	0.00	4.62
MRSB/500	3097	17.90	0.00	4.47

Table 4: Results for the Barth5 mesh: $P = 64$, $N = 15,606$, $E = 45,878$

Table 4 shows results for the Barth5 mesh and $P = 64$. As is reflected in this table, throughout all experiments carried out JOSTLE appears to achieve approximately the same partition quality as MRSB though much more rapidly. The $\lvert E_c \rvert$ figures are slightly lower than for MRSB and the imbalances are generally small (smaller than one would expect for simulated annealing, say); for example, the 1.23% imbalance arises from the most heavily loaded processor having 247 nodes as opposed to an optimum of 244.

	GREEDY		JOSTLE		MRSB	
P	$\lvert E_c \rvert$	$t(s)$	$\lvert E_c \rvert$	$t(s)$	$\lvert E_c \rvert$	$t(s)$
16	10939	3.04	6212	14.70	6213	38.11
32	15549	3.17	9038	15.72	9459	61.17
64	20017	1.87	12426	16.56	13035	107.16
128	26274	2.07	15784	23.78	17751	148.30
256	33556	2.14	22250	24.51	22745	172.47

Table 5: Results for Tet100K mesh: $N = 103,081$, $E = 200,976$

Table 5 gives results for the Tet100K mesh and varying P. Here the JOSTLE code has been run in the 'JOSTLE/reduction' configuration and MRSB has been run with a contraction threshold of 100. Again we see approximately equivalent partition quality between JOSTLE and MRSB although with much faster run times. More importantly, the timings for JOSTLE increase more slowly with P than those for MRSB, suggesting that as machine sizes increase it should be even more efficient.

P	$\lvert E_c \rvert$	$t(s)$	$\delta\%$	D_a
10	49549	133.37	0.00	4.60
20	64088	133.67	0.00	6.30
50	87060	132.32	0.00	9.68
100	116966	148.85	0.01	10.98
200	129963	167.92	0.02	12.38

Table 6: Results for Tet1M mesh: method is 'JOSTLE/fast rdtn', $N = 1,119,663$, $E = 2,212,012$

In the final set of results, table 6, we offer an example of how fast the code can be on a workstation even for a huge (million node) mesh. Indeed these runs were carried out on a Sun SPARC station which is about 50% slower than the Silicon Graphics Indigo 2 used for the other experiments. We were unable to run MRSB on this problem because we could not compile the code with enough memory. However, these runtimes compare very favourably with a similar sized problem (1,010,174 tetrahedra) partitioned using a parallelised RSB code which took 178 seconds to run on 256 nodes of a Thinking Machines CM5 [10].

6 Conclusion

We have outlined a new method for optimising graphs partitions with a specific focus on its application to the mapping of unstructured meshes onto parallel computers. In this context the graph-partitioning task can be very efficiently addressed through a two-stage procedure – one to yield a legal initial partition and the second to improve its quality with respect to interprocessor communication and load-balance. The method is further enhanced through the use of a clustering technique. The resulting software tool, JOSTLE, has been designed for implementation in parallel for both static and dynamically refined meshes. For the experiments reported in this paper on static meshes of up to one million nodes, the JOSTLE procedures are an order of magnitude faster than existing techniques, such as Multilevel Recursive Spectral Bisection, with equivalent quality.

Future work on this technique will include further study into distributed load-balancing algorithms, a parallel implementation and work on the mapping of meshes to machine topologies.

Acknowledgements

We would like to thank Horst Simon for the copy of his Multilevel Recursive Spectral Bisection code. We would also like to acknowledge Peter Lawrence and Kevin McManus for supplying the Tri60K, Tet100K and Tet1M meshes.

References

[1] S. T. Barnard and H. D. Simon. A Fast Multilevel Implementation of Recursive Spectral Bisection for Partitioning Unstructured Problems. *Concurrency: Practice & Experience*, 6(2):101–117, 1994.

[2] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Prentice Hall, Englewood Cliffs, NJ, 1989.

[3] G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *J. Par. Dist. Comput.*, 7:279–301, 1989.

[4] C. Farhat. A Simple and Efficient Automatic FEM Domain Decomposer. *Comp. & Struct.*, 28:579–602, 1988.

[5] C. Farhat and H. D. Simon. TOP/DOMDEC – a Software Tool for Mesh Partitioning and Parallel Processing. Tech. Rep. RNR-93-011, NASA Ames, Moffat Field, CA, 1993.

[6] C. M. Fiduccia and R. M. Mattheyses. A Linear Time Heuristic for Improving Network Partitions. In *Proc. 19th IEEE Design Automation Conf.*, pages 175–181, IEEE, 1982.

[7] J. R. Gilbert and E. Zmijewski. A Parallel Graph Partitioning Algorithm for a Message-Passing Multiprocessor. *Int. J. Parallel Programming*, 16(6):427–449, 1987.

[8] B. Hendrickson and R. Leland. Multidimensional Spectral Load Balancing. Tech. Rep. SAND 93-0074, Sandia National Labs, Albuquerque, NM, 1992.

[9] B. Hendrickson and R. Leland. A Multilevel Algorithm for Partitioning Graphs. Tech. Rep. SAND 93-1301, Sandia National Labs, Albuquerque, NM, 1993.

[10] Z. Johan, K. K. Mathur, S. Lennart Johnsson, and T. J. R. Hughes. Scalability of Finite Element Applications on Distributed Memory Parallel Computers. TR-16-94, Harvard University, Cambridge, MA, 1994.

[11] B. W. Jones. *Mapping Unstructured Mesh Codes onto Local Memory Parallel Architectures*. PhD thesis, School of Maths., University of Greenwich, London SE18 6PF, UK, 1994.

[12] B. W. Kernighan and S. Lin. An Efficient Heuristic for Partitioning Graphs. *Bell Systems Tech. J.*, 49:291–308, 1970.

[13] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[14] R. Lohner, R. Ramamurti, and D. Martin. A Parallelizable Load Balancing Algorithm. AIAA-93-0061, 1993.

[15] K. McManus, M. Cross, and S. Johnson. Integrated Flow and Stress using an Unstructured Mesh on Distributed Memory Parallel Systems. In *Parallel CFD'94*. Elsevier, 1995.

[16] J. Savage and M. Wloka. Parallelism in Graph Partitioning. *J. Par. Dist. Comput.*, 13:257–272, 1991.

[17] N. G. Shivaratri, P. Krueger, and M. Singhal. Load distributing for locally distributed systems. *IEEE Comput.*, 25(12):33–44, 1992.

[18] J. Song. A partially asynchronous and iterative algorithm for distributed load balancing. *Parallel Comput.*, 20:853–868, 1994.

[19] P. R. Suaris and G. Kedem. An Algorithm for Quadrisection and Its Application to Standard Cell Placement. *IEEE Trans. Circuits and Systems*, 35(3):294–303, 1988.

[20] D. Vanderstraeten and R. Keunings. Optimized Partitioning of Unstructured Finite Element Meshes. (accepted by *Int. J. Num. Meth. Engng.*), 1993.

[21] C. Walshaw, M. Cross, S. Johnson, and M. Everett. A Parallelisable Algorithm for Partitioning Unstructured Meshes. In *Proc. Irregular '94: Parallel Algorithms for Irregularly Structured Problems* (in press), 1994.

[22] C. Walshaw, M. Cross, S. Johnson, and M. Everett. JOSTLE: Partitioning of Unstructured Meshes for Massively Parallel Machines. (submitted for the Proceedings, Parallel CFD'94, Kyoto), 1994.

[23] C. Walshaw, M. Cross, S. Johnson, and M. Everett. Mapping Unstructured Meshes to Parallel Machine Topologies. (in preparation), 1994.

[24] C. H. Walshaw and M. Berzins. Dynamic Load-Balancing For PDE Solvers On Adaptive Unstructured Meshes. (accepted by *Concurrency: Practice & Experience*), 1993.

[25] R. D. Williams. Performance of dynamic load balancing algorithms for unstructured mesh calculations. *Concurrency: Practice & Experience*, 3:457–481, 1991.