Parallel Mesh Partitioning on Distributed Memory Systems *

C. Walshaw and M. Cross
Computing & Mathematical Sciences, University of Greenwich, Park Row, Greenwich, London, SE10 9LS, UK.
C.Walshaw@gre.ac.uk; http://www.gre.ac.uk/~c.walshaw

Abstract

We discuss the problem of deriving parallel mesh partitioning algorithms for mapping unstructured meshes to parallel computers. In itself this raises a paradox – we seek to find a high quality partition of the mesh, but to compute it in parallel we require a partition of the mesh. In fact, we overcome this difficulty by deriving an optimisation strategy which can find a high quality partition even if the quality of the initial partition is very poor and then use a crude distribution scheme for the initial partition. The basis of this strategy is to use a multilevel approach combined with local refinement algorithms. Three such refinement algorithms are outlined and some example results presented which show that they can produce very high global quality partitions, very rapidly. The results are also compared with a similar multilevel serial partitioner and shown to be almost identical in quality. Finally we consider the impact of the initial partition on the results and demonstrate that the final partition quality is, modulo a certain amount of noise, independent of the initial partition.

Key words. graph partitioning, mesh partitioning, load-balancing, multilevel algorithms.

1 Introduction

The need for mesh partitioning arises naturally in many computational fluid dynamics (CFD) and computational mechanics (CM) applications. Meshes composed of elements such as triangles or tetrahedra are often more suited than regularly structured grids for representing completely general geometries and variable mesh densities. Meanwhile, the modelling of complex behaviour patterns often renders problems

*Invited lecture. In: Proc. Parallel & Distributed Computing for Computational Mechanics, Weimar, Germany, 1999

putational demands, or both. Such computational modelling challenges increasingly require, therefore, the solution to be calculated as efficiently as possible in parallel on an unstructured mesh.

1.1 Problem classification

Distributing an unstructured mesh across a parallel computer so that the computational load is evenly balanced and the parallel overhead minimised is known as *mesh partitioning*. It is well known that this problem is NP-complete, e.g. [9], and in recent years much attention has been focused on developing suitable heuristics, many based on a graph corresponding to the communication requirements of the mesh. Typically, such mesh partitioning problems arise in three different ways and can be characterised as the:–

(i) **static partitioning problem** (the classical problem) which arises in trying to distribute an existing mesh amongst a set of processors;

(ii) **static load-balancing problem** which arises from a mesh that has been generated in parallel;

(iii) **dynamic load-balancing/partitioning problem** which arises from either adaptively refined meshes, or from meshes in which the computational workload for each mesh entity can vary with time or even from machines on which (due to external user load) the computational resources may vary.

In the second two cases, (ii) & (iii), the initial data is a distributed mesh which may be neither load-balanced nor optimally partitioned. One way of dealing with this is to ship the mesh back to some host processor, run a serial partitioning algorithm on it and redistribute. However, this is unattractive for many reasons. Firstly, an $O(N)$ overhead for the mesh partitioning is simply not scalable if the solver is running at $O(N/P)$. Indeed the mesh may not even fit into the memory of the host machine and thus incur enormous delays through memory paging. In addition, a partition of the mesh (which may even be optimal) already exists, so it makes sense to reuse this as a starting point for repartitioning, [23, 31]. 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. Thus, because the mesh is already distributed, it is a natural strategy to repartition it *in situ*.

On the other hand, it could be argued that case (i) can be best handled by a serial partitioning algorithm (of which many exist). However, once again an $O(N)$ start-up cost for the mesh partitioning may not be acceptable and the same memory problems can arise. Also, assuming that a parallel machine is available to run the solver, it makes sense to use it for the initial partition as well.

With these issues in mind it seems to make sense to look for a partitioning framework with the requirements that (apart from being as fast as possible) it:–

(a) works in parallel;

(b) can optimise an existing partition;

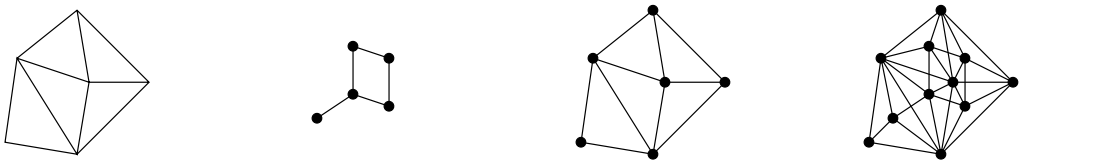
(c) can find a high quality partition *independent* of the existing partition;

(d) incorporates a load-balancing technique.

This of course raises a paradox for case (i): we seek to find a partition of a previously unpartitioned mesh in parallel; however, to do it in parallel we must first distribute the mesh sensibly amongst the processors and to distribute the mesh sensibly we must first find a reasonable partition. In fact it is requirement (c) that, if met, can answer this paradox. It is easy to distribute a mesh if there are no guarantees about the partition quality (e.g. by assigning mesh entities to processors on a cyclic basis). Thus if requirement (c) can be met, we can indeed solve problem (i) to high quality by initially using a crude distribution of the mesh and then optimising it in parallel.

In this paper we shall outline a framework for a mesh partitioning strategy which can satisfy all four requirements (a)-(d) and which thus aims to solve all three problems (i)-(iii) using the same algorithm. In particular this paper will focus on the solution of the static partitioning problem (i) and the requirements (a)-(c). The companion paper, entitled 'Dynamic Mesh Partitioning and Load-Balancing for Parallel Computational Mechanics Codes', [28], will focus on problem (iii) and requirement (d).

1.3 Notation and Definitions



(a) mesh (b) dual graph (c) nodal graph (d) combined graph

Figure 1: An example mesh and some possible graph representations.

In the context of mesh partitioning, it is normal to represent an unstructured mesh of elements, nodes, faces, etc. as a graph of vertices & edges which represent units of workload and data dependencies, respectively. In fact this is a useful abstraction to measure partition quality, even if, as in the case of geometric partitioners (§2), the

ure 1(a), the graph vertices can either represent the mesh elements (the dual graph; Figure 1(b)), mesh nodes (the nodal graph; Figure 1(c)), a combination of both (the full or combined graph; Figure 1(d)) or some special purpose representation to model more complicated interactions in the mesh. In each case the graph vertices represent units of workload that exist in the underlying solver and edges represent data dependencies (e.g. the temperature in a given element will depend on (at least) its immediate neighbouring elements).

Thus, let $G = G(V, E)$ be an undirected graph of vertices V and edges E. We assume that both vertices and edges can be weighted (with positive integer values) and that $|v|$ denotes the weight of a vertex v and similarly for edges and sets of vertices and edges. Given that the mesh needs to be distributed to P processors, define a partition π to be a mapping of V into P disjoint subdomains S_p such that $\bigcup_P S_p = V$. The partition π induces a *subdomain graph*, $G_\pi(S, L)$, on G with vertices S_p representing subdomains (the sets of vertices assigned to processor p) and edges or links $(S_p, S_q) \in L$ if there are vertices $v_1, v_2 \in V$ with $(v_1, v_2) \in E$ and $v_1 \in S_p$ and $v_2 \in S_q$. The weight of a subdomain is just the sum of the weights of the vertices in the subdomain, $|S_p| = \sum_{v \in S_p} |v|$. We denote the set of inter-subdomain or cut edges (i.e. edges cut by the partition) by E_c (note that the total weight of cut edges $|E_c| = |L|$ the total weight of edges in the subdomain graph). Vertices which have an edge in E_c (i.e. those which are adjacent to vertices in another subdomain) are referred to as *border* vertices. Finally, note that we use the words subdomain and processor more or less interchangeably: the mesh is partitioned into P subdomains; each subdomain S_p is assigned to a processor p and each processor p owns a subdomain S_p.

The definition of the graph partitioning problem is to find a partition which evenly balances the load (i.e. vertex weight) in each subdomain whilst minimising the communications cost. To evenly balance the load, the optimal subdomain weight is given by $\overline{S} := \lceil |V|/P \rceil$ (where the ceiling function $\lceil x \rceil$ returns the smallest integer greater than x) and the *imbalance*, θ, is then defined as the maximum subdomain weight divided by the optimal (since the computational speed of the underlying application is determined by the most heavily weighted processor). Note that $\theta \geq 1$ and perfect balance is given by $\theta = 1$. As is usual, throughout this paper the communications cost will be estimated by $|E_c|$, the weight of cut edges or cut-weight, although see §3.1 for further discussion on this point. A more precise definition of the graph partitioning problem is therefore to find π such that $|S_p| \leq \overline{S}$ and such that $|E_c|$ is minimised. Note that perfect balance is not always possible for graphs with non-unitary vertex weights and in fact it is quite usual to tolerate a certain amount of imbalance (e.g. an imbalance tolerance of 3% or $\theta = 1.03$) in order that higher quality partitions can be found.

Although the aim of this paper is to focus on graph-based parallel partitioning strategies, nonetheless it should be mentioned that a relatively simple solution to the problem is to utilise one of several geometric based sorting approaches. For example, if all mesh entities are sorted by x-coordinate (and very rapid parallel sorting algorithms exist) then the mesh can be easily partitioned by cutting it with planes orthogonal to the x-axis and intuitively such a partition should (at least for well shaped meshes) keep the volume of data to be communicated small. Such a strategy can also be combined with a recursive bisection approach (by bisecting the region and then recursively bisecting the resulting subregions) and this approach allows cutting planes orthogonal to any of the axes (known as Recursive Coordinate Bisection, [24]) or orthogonal to some feature of the region (such as Recursive Inertial Bisection where the cutting planes are determined by the principle moment of inertial of the original geometry, [8]). Indeed, in an algorithm known as Unbalanced Recursive Bisection, Jones & Plassmann, [16], even modified the technique to produce subdomains with good aspect ratio by not requiring that the bisection split each domain into equal weight portions but instead considering the resulting aspect ratios of the two subdomains. Once again, all that these strategies require is the multiple application of a parallel sorting algorithm rendering them very fast and simple to implement. However, the quality of partitions is generally much poorer than that produced by graph-based algorithms which consider the mesh data dependencies. The geometric approaches can produce subdomains which are long and thin or which are split into multiple disconnected components. Moreover, the quality tends to get worse the more complicated the geometry of the mesh becomes.

A more sophisticated geometric approach, developed by Miller et al., [21], uses circles or spheres rather than planes to cut the mesh. In addition mesh quality measures (i.e. for most meshes, entities which have a data dependency are likely to be geometrically close together) allow the authors to give theoretical bounds on the partition quality. It is more complicated to implement than the cutting plane approaches but has been parallelised in [15].

3 Parallel partition optimisation algorithms

In Section 1.2 we discussed the need for parallel algorithms which can optimise an existing partition and so in this section we discuss the problems which arise in trying to develop such parallel iterative partition optimisation algorithms. Initially we describe the concept of the gain & preference functions (§3.1) and then in §3.2 we discuss the outer iterative loop of the optimisation and the difficulties arising in a parallelising the inner vertex migration loop. Three possible solutions (detailed descriptions of which can be found in [29]) are suggested in §3.3.

Key concepts for partition optimisation algorithms are the ideas of *gain* and *preference*. Loosely, the gain, $\texttt{gain}(v, q)$, of a vertex v in subdomain S_p can be calculated for every other subdomain, S_q, $q \neq p$, and expresses some 'estimate' of how much the partition would be 'improved' were v to migrate to S_q. The preference, $\texttt{pref}(v)$, is then just the preferred subdomain for v to migrate to and thus the value of q which maximises the gain; i.e. $\texttt{pref}(v) = q$ where $\texttt{gain}(v, q)$ attains $\max_{r \in P} \texttt{gain}(v, r)$. Throughout the following vertices are only allowed to migrate to the subdomain to which their preference is set.

Note that the gain is usually directly related to some cost function which measures the quality of the partition and which we aim to minimise. Typically the cost function used is simply the total weight of cut edges or cut-weight, $|E_c|$, and then the gain expresses the change in $|E_c|$. More recently, there has been some debate about the most important quantity to minimise (e.g. [11]) and, for example, in [26], Vanderstraeten *et al.* demonstrate that it can be extremely effective to vary the cost function based on a knowledge of the solver, ideas which, in [30], we have used to extend multilevel techniques to optimise for subdomain shape or aspect ratio. Whichever cost function is chosen, however, the idea of gains is generic.

For the purposes of this paper we shall assume that $\texttt{gain}(v, q)$ just expresses the reduction in the cut-weight, $|E_c|$. Note that there can never be a reduction in the cut-weight if a vertex v is transferred to a subdomain S_q to which it is not adjacent (since there will be no cut edges between v and S_q). For this reason, we only calculate gains for each border vertex to their adjacent subdomains and this in turn restricts the preference to such subdomains. Indeed, in a high quality partition, most border vertices will only be adjacent to one other subdomain, S_q, and then the preference is simply q. As a consequence processors only migrate vertices to neighbouring subdomains along edges of the subdomain graph (see Section 1.3).

3.2 Parallelising a serial iterative optimisation algorithm

Consider the (partitioned) graph depicted in Figure 2. We can determine the gain and preference for each border vertex as shown; for example as $2p$ for vertex v meaning that it has a gain of 2 and a preference to migrate to S_p (or in other words, migrating vertex v from subdomain S_r to subdomain S_p will reduce the cut-weight by 2). A typical serial Kernighan-Lin (KL) type algorithm for optimising this partition (such as described in [27]) would consist of inner and outer iterative loops. The inner loop picks vertices (usually those with the highest gain) and migrates them from one subdomain to another. It will not usually visit any vertex more than once during the course of an inner loop in order to prevent cyclic behaviour and terminates when all vertices have been visited or when there is little prospect of further improvement with the unvisited vertices. The outer loop is simply repeated applications of the inner loop and terminates when no migration takes place within an inner loop.

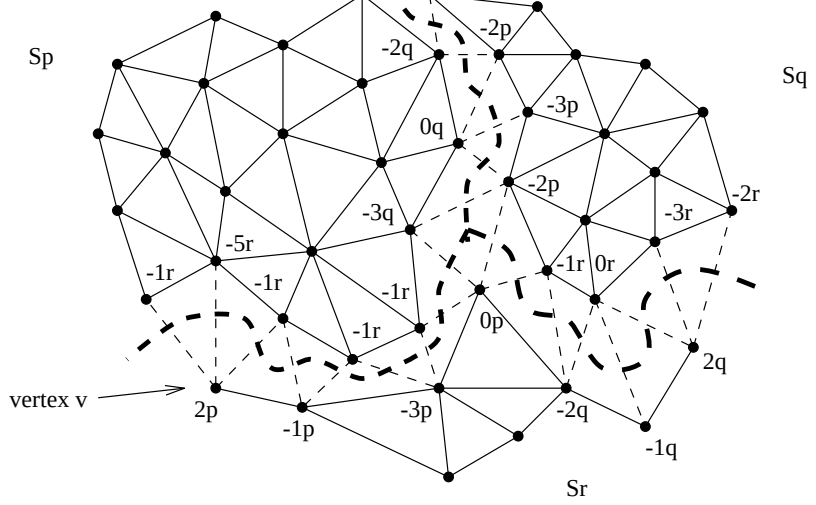


Figure 2: An example graph with subdomains S_p, S_q & S_r.

```
while (optimising) {
    optimising = 0;
    calculate gain & preference of own border vertices;
    halo update of gains & preferences;
    determine which vertices to migrate (inner loop);
    if (migration required) {
        optimising = 1;
        bulk migration of vertices;
    }
    global update (optimising);
}
```

Figure 3: The outer iterative loop.

The main problems in parallelising this procedure lie within the inner loop. Firstly, if the graph is distributed, migrating one vertex at a time involves far too much communication overhead (with most of the processors lying idle most of the time) and for this reason we employ a bulk migration scheme where each processor finds as many border vertices as possible to migrate and moves them once per iteration of the outer loop. The outer loop (executed concurrently on each processor) is shown in Figure 3. Note that it contains three communication steps, a halo update of the border vertices gain and preference values, the migration of vertices to their neighbours and the global update of the `optimising` flag (see Section 5).

The second and more difficult problem in parallelising the serial algorithm lies in determining which vertices to migrate. In fact, the swapping of vertices between two subdomains is an inherently non-parallel operation and hence there are some difficulties in arriving at efficient parallel versions, [22]. Since all the processors are acting in parallel on the vertices that they own, simply moving vertices with the

Figure 4: An example collision when vertices with postive gains migrated simultaneously result in an *increase* in cost.

highest gain is not a satisfactory solution as it means that adjacent vertices may be swapped simultaneously (a non-optimal event often known as a *collision*) and this may lead to an *increase* in the cost, particularly in graphs with weighted edges. For example, given the situation in Figure 4 with edges weighted as shown, processor p may wish to migrate vertex v_2 to S_q (on the basis that it has a gain of 1) while at the same time processor q wishes to migrate vertex v_3 to S_p for the same reason. Whilst the migration of either of these vertices individually will result in a reduction in the cut-weight of 1, the migration of both at the same time will actually result in an increase in cut-weight from 2 to 4.

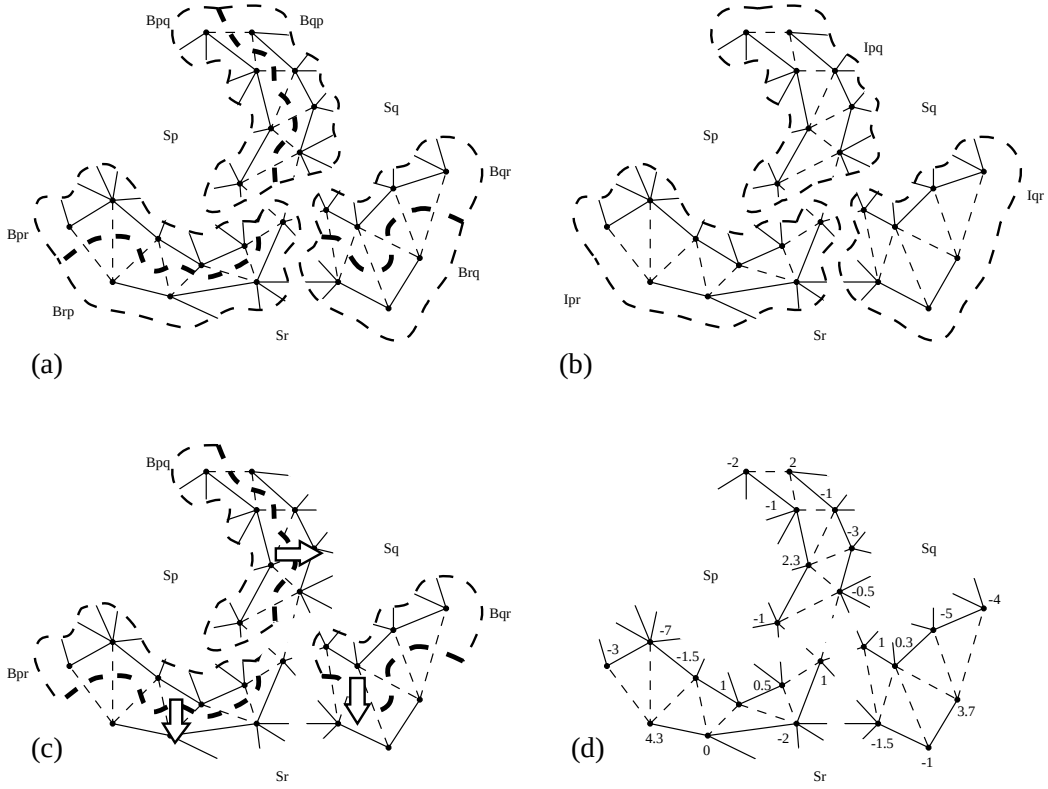


Figure 5: An example graph showing (a) subdomain faces; (b) interface regions for independent optimisation; (c) one of each pair of faces selected of alternating optimisation; and (d) relative gains as a 'potential field'.

since every border vertex has a subdomain and a preference we can isolate border regions and define subsets $B_{pq} = \{v \in B_p : \texttt{pref}(v) = q\}$, or in other words, B_{pq} is the set of vertices in the border B_p of subdomain S_p with a preference q. We will refer to these sets as *subdomain faces*. Figure 5(a) shows the six subdomain faces for the example graph in Figure 2. Each pair of subdomain faces, $B_{pq} \bigcup B_{qp}$ then forms an *interface* region I_{pq}. Note that since the preference of every border vertex is fixed throughout each outer iteration (because it is only determined once during the iteration) then these interfaces cannot change during that iteration. This allows us to isolate regions of the graph which in turn helps to avoid collisions.

3.3 Outline algorithms

Full descriptions of three different algorithms for choosing, in parallel, vertices to migrate whilst attempting to avoid collisions are given in [29]. Here, to motivate them quickly without the somewhat intricate details, the algorithms can be summarised as:–

- **Interface Optimisation**. A serial optimisation algorithm is executed independently in each of the interface regions I_{pq} by either one of the processors p or q. Figure 5(b) shows the three interface regions for the example graph in Figure 2.

- **Alternating Optimisation**. One of each pair of subdomain faces is selected and the owning processor chooses vertices from that face for migration (to its opposite face). A certain amount of imbalance tolerance (see §1.3) is crucial for this algorithm to work because the active processor is not allowed to create serious imbalance and so if the tolerance is zero no vertices can be migrated. In the following iteration of the outer loop the alternate face is selected. Figure 5(c) shows an example of the three selected regions in a given iteration of the outer loop for the graph in Figure 2.

- **Relative Gain Optimisation**. If we think of the gain as a force or potential we can imagine a relative gain for every border vertex according to the neighbouring vertices in the opposite face. Intuitively, if the gain of the opposite vertices is high they are likely to migrate and so v should not migrate; if the opposing gain is low then there is little danger of a collision if v migrates and so the relative gain attempts to express this migration potential. Figure 5(d) shows the relative gains of the border vertices for the graph in Figure 2. Each processor then picks an 'appropriate' weight of vertices to migrate, highest relative gain first. The fact that the gains of all vertices in the opposite face are taken into account (in the relative gain calculation) helps to avoid most collisions.

In [29] these three approaches are compared (within the context of a multilevel partitioner – see below §4); a summary of the results can be found in §6.1.

Returning to the requirements in §1.2, so far we have discussed in Section 3 algorithms which (a) work in parallel and (b) can optimise an existing partition. In fact, they also incorporate a load-balancing component (although not described here – see [29]) and hence, in addition, satisfy requirement (d). However, the serial Kernighan-Lin algorithm on which they are based is an inherently localised optimisation algorithm. In other words, given a poor quality initial partition it can make local improvements, but is unlikely to achieve a high global quality solution (because its limited hill-climbing abilities will not always allow it to escape from local minima traps in the solution space).

In recent years, therefore, it has been recognised that an effective way of both speeding up partition optimisation algorithms and, perhaps more importantly giving them a global perspective is to use multilevel techniques. The idea is to group vertices together to form *clusters*, use the clusters to define a new graph and recursively iterate this procedure to create a series of increasingly coarse graph until the size of the coarsest graph falls below some threshold. A fast and possibly crude initial partition of the coarsest graph is calculated and then successively interpolated onto and optimised on each of the graphs in reverse order. This sequence of contraction followed by repeated interpolation/optimisation is known as the *multilevel paradigm*. The multilevel idea was first proposed by Barnard & Simon, [2], as a method of speeding up spectral bisection and improved by both Hendrickson & Leland, [14] and Bui & Jones, [4], who generalised it to encompass local refinement algorithms.

4.1 Graph contraction

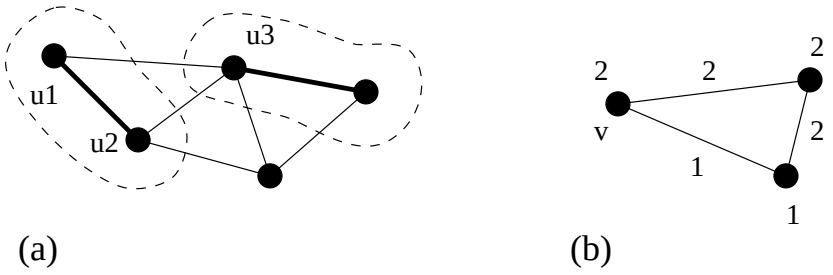


Figure 6: An example of graph contraction.

To create a coarser graph $G_{l+1}(V_{l+1}, E_{l+1})$ from $G_l(V_l, E_l)$ we use a variant of the edge contraction algorithm proposed by Hendrickson & Leland, [13]. The idea is to find a maximal independent subset of graph edges (or *matching* of graph vertices) and then collapse them. The set is independent if no two edges in the set are incident on the same vertex (so no two edges in the set are adjacent), and maximal if no more edges can be added to the set without breaking the independence criterion. Having found such a set, each selected edge is collapsed and the vertices, $u_1, u_2 \in V_l$ say, at

Figure 6(a) shows an example of a matching with the thicker lines showing those edges that have been selected for collapse and the dashed lines show the resulting pairs of vertices which will be merged to form a new vertex in the coarser graph. Figure 6(b) show the resulting coarsened graph with edge and vertex weights marked (assuming unit weights in the original graph). Edges which have not been collapsed are inherited by the child graph, G_{l+1}, and, where they become duplicated, are merged with their weight summed. This occurs if, for example, the edges (u_1, u_3) and (u_2, u_3) exist when edge (u_1, u_2) is collapsed. Because of the inheritance properties of this algorithm, it is easy to see that the total graph weight remains the same, $|V_{l+1}| = |V_l|$, and the total edge weight is reduced by an amount equal to the weight of the collapsed edges. A full description of the parallel implementation of the matching techniques and the construction of the coarsened graph can be found in [31].

4.2 The initial partition & the global graph

The normal practice of the serial multilevel strategy is to construct the series of graphs until the number of vertices in the coarsest graph is smaller than some threshold and then carry out an initial partition. In parallel, the graph is already distributed and so an initial partition already exists. Here, following the idea of Gupta, [10], we continue coarsening until the number of vertices in the coarsest graph is the same as the number of subdomains, P, and this gives us automatically an initial partition with one vertex per subdomain. However, although contraction down to a single vertex per subdomain is rapid in serial (since at the coarsest levels the graphs become very small indeed), in parallel it can be relatively inefficient since each contraction involves several communication phases. For this reason, once the size of the graph falls below a given threshold, each processor broadcasts its portion so that every processor has a copy of the entire graph (which we refer to as the *global graph*). The contraction and interpolation/optimisation process can then continue entirely in serial with every processor duplicating the work. The serial algorithms used are described in full in [27], although essentially the techniques are very similar to those discussed here. The optimum threshold at which to construct the global graph is of course machine dependent (based on the ratio of the cost of communication and computation) but the default setting (which can be reset at run-time) for the results in this paper is 20 vertices per processor.

5 Parallel implementation

The software tool written at Greenwich to implement the optimisation techniques outlined here is known as JOSTLE and is freely available for academic and research purposes under a licensing agreement[1]. It is written in C for distributed memory par-

[1] available from `http://www.gre.ac.uk/jostle`

library MPI (although of course it will also run efficiently on shared memory architectures where MPI is installed). We work in the owner-computes single-program multiple-data paradigm so that the vertices in each subdomain, S_p, are assigned to processor p, which also holds a one deep halo or read-only copy of vertices adjacent to S_p. We classify parallel operations as local, neighbourhood or global; local operations take place entirely on processor, neighbourhood operations involve communication with processors neighbouring in the subdomain graph and global operations involve all of the processor communicating together. Here we employ three communication operations: global reduction, halo updates and vertex migration. Reduction is a global operation on scalars (or short vectors) such as finding a maximum across all the processors. Halo updating is a neighbourhood operation and involves each processor, p, informing its neighbours of certain values assigned to its border vertices, B_p.

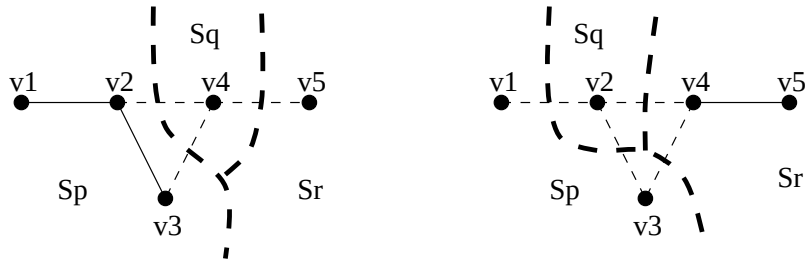


Figure 7: An example of vertex migration.

Vertex migration, the transfer of vertices from one distributed memory processor to another, is also a neighbourhood operation but is a non-trivial task to implement efficiently, [12], particularly matching up the pointer lists of vertices & edges and ensuring consistency of halo information across the parallel system. Space limitations prevent a full description but we give an overview of the technique which has been implemented as a procedure with two communication subphases. In the first, vertices marked for migration are packed into buffers together with any additional vertices and edges required to complete the halo on the destination processor and not already known to exist there. Consider for example the graph shown in Figure 7. If vertex v_2 migrates from S_p to S_q then a copy of vertex v_1 must also be sent to complete the halo on q, but a copy of v_3 need not be sent because it already exists in the halo of S_q. Vertices which are migrating also initiate messages to neighbouring processors which are not their destination; e.g. if v_4 migrates to S_r then processor p must be informed of this move. Having sent all such messages each processor unpacks the corresponding received messages and creates the new vertices and edges as instructed. The second communication phase is required (unfortunately) to fully update the ownership of halo vertices. In the example, when v_4 migrates to S_r it takes a copy of v_2 with it as a halo vertex. However, in the meantime v_2 has migrated to S_q and so q must send another message to r to update this information. Finally halo vertices which are no longer required are deleted; e.g. prior to the migration v_5 is a halo vertex of S_q, afterwards it is not required.

and vertex migration require some scheme for each processor to find local copy of a vertex. A global array of pointers (of length V) would render the memory unscalable, so each processor maintains an array, of scalable size $O(N/P)$, of binary tree structures which can be rapidly searched. Thus, given a global index, a processor can locate the appropriate binary tree in the array (using a modulus function on the global index) and search it for a pointer to the vertex. The use of such binary trees means that pointers to vertices can be easily added or deleted as migration occurs; the use of an array of them means that none of the trees should become too deep.

5.1 Related work

Whilst there has been a considerable amount of research into mesh partitioning recently, little of it seems to be specifically on the parallel solution of the graph partitioning problem. Nonetheless, a number of parallel methods do exist. The multilevel recursive spectral bisection algorithm, [2], has been parallelised [1]; this greatly improves the performance but the algorithm is still relatively slow (because of the need to find eigenvectors of a graph and the resulting requirement for expensive floating point linear algebra). A similar problem arises for HARP, [25], a parallel spectral inertia bisection algorithm, although once the eigenvectors are calculated initially (and possibly off-line) the algorithm can be repeatedly used for dynamically load-balancing graphs where the graph weights change (providing the edge topology remains fixed). A number of parallel single-level algorithms have also been developed, such as [3, 5, 6, 20], however without the global view provided by the multilevel techniques it is unclear whether such methods can achieve the highest quality partitions and they are often more suited to incremental dynamic partitioning and load-balancing where the existing partition may already be of high quality.

Most closely related to the work presented here is ParMETIS, the parallel graph partitioner of Karypis & Kumar, [17, 23]. This uses an alternating tolerance-based optimisation algorithm similar to the one suggested in §3.3 (although we have additionally enhanced the algorithm by incorporating load-balancing directly into the optimisation process). Perhaps the major difference in strategy though is the approach to vertex migration. ParMETIS uses virtual migration and so the graph distribution is fixed throughout the optimisation and vertices which migrate from one subdomain to another simply have their subdomain field changed and thus a processor may own subsets of several (or even all) subdomains. In the algorithms described above, however, each subdomain is mapped to a single processor and vertices which migrate from one subdomain to another are actually copied and recreated on the destination processor.

The algorithms have been tested on a Cray T3E-900/512 at the University of Stuttgart. For each test the mesh is read in parallel and distributed contiguously to the processors (i.e. processor 0 is given the first $|V|/P$ vertices, processor 1 the next $|V|/P$, etc.). This means the initial partition can be of extremely poor quality (although see §6.3 for results on the impact of the initial distribution). The algorithm is allowed a 5% final imbalance tolerance (set at run-time by the user).

mesh	$\|V\|$	$\|E\|$	mesh type
4elt	15606	45878	2D nodal graph
t60k-n	30570	90575	2D nodal graph
t60k-d	60005	89440	2D dual graph
dime20	224843	336024	2D dual graph
t60k-f	90575	360030	2D full graph
fe-rotor	99617	662431	3D nodal graph
598a	110971	741934	3D nodal graph
mesh100	103081	200976	3D dual graph
cyl3	232362	457853	3D dual graph
fe-ocean	143437	409593	3D semi-structured graph

Table 1: Test meshes.

The test meshes have been chosen to be a representative sample of medium to large scale real-life problems and include both 2D and 3D examples of nodal graphs (where the mesh nodes are partitioned) and dual graphs (where the mesh elements are partitioned). Table 1 gives a list of the meshes and their sizes; since none of the graphs are weighted the number of vertices in V is the same as the total vertex weight $|V|$ and similarly for the edges E. Note that t60k-f is a combination of the t60k nodal graph and t60k dual graph, with the addition of edges between vertices from t60k-d which represent mesh elements and the vertices from t60k-n which represent their nodes.

6.1 Parallel results

The results of the parallel multilevel partitioning using the interface optimisation algorithm outlined in §3.3 are shown in Table 2 for four values of P (the number of processors/subdomains). The table shows the total weight of cut edges or cut-weight, C, and the parallel run-time in seconds, $t(s)$. In themselves the cut-weight results are not very illuminating but the following sections demonstrate their quality relative to a serial multilevel partitioner and the impact that the initial distribution has on the final partition. We do not show the final imbalance in the partition, but on average it was 4.7% and never exceeded the allowed imbalance of 5%.

Achieving high parallel performance for parallel partitioning codes such as JOSTLE is not as easy as, say, a typical CFD or CM code. For a start the algorithms use only

mesh	C	$t(s)$	C	$t(s)$	C	$t(s)$	C	$t(s)$
4elt	1070	0.49	1676	0.67	2728	0.84	4324	1.13
t60k-n	1753	0.87	2930	0.82	4378	0.79	6592	1.34
t60k-d	925	0.54	1573	0.52	2381	0.70	3525	1.31
dime20	1305	1.49	2256	1.17	3632	1.26	5374	1.97
t60k-f	5190	3.46	7931	3.33	12118	2.87	18200	3.20
fe-rotor	22789	8.36	36345	7.20	50580	6.58	70933	8.04
598a	27009	17.17	42172	12.63	59866	10.38	82292	10.54
mesh100	4662	2.85	6795	2.41	9993	2.61	13929	3.70
cyl3	9976	12.32	14639	7.98	20211	6.34	27628	6.77
fe-ocean	8546	6.52	14192	4.62	21845	3.60	31420	4.29

Table 2: The results of the parallel interface algorithm showing the cut-weight C and parallel run-time in seconds $t(s)$.

integer operations and so there are no MFlops[2] to 'hide behind'. In addition, most of the work is carried out on the subdomain boundaries and so very little of the actual graph is used. Also the partitioner itself may not necessarily be well load-balanced and the communications cost may dominate on the coarsest reduced graphs since at this stage there are very few vertices per processor. On the other hand, as was explained in §1.1, partitioning on the host may be impossible or at least much more expensive and if the cost of partitioning is regarded (as it should be) as a parallel overhead, it is usually extremely inexpensive relative to the overall solution time of the problem. With that in mind, the figures show good timings for this sort of code and more importantly, very low overheads (always less than 20 seconds) for the parallel partitioning. Indeed some of them could be substantially better if the mesh numbering bore some relation to locality (see below §6.3). The timings generally decrease as P increases although this is not so true on the smaller meshes and not for $P = 128$. We believe that this is because there is so little computational work that these figures just show parallel communication overhead.

We do not show here the results for the alternating and relative gain optimisation algorithms, but a discussion of their relative merits can be found in [29]. To summarise those results, the interface optimisation algorithm (see §3.3) generally produces very high quality partitions, very rapidly and provides the best results in terms of cut-weight. However, it does not completely remove imbalance in the final partition and a hybrid algorithm, using relative gain (see §3.3) with a final clean-up step of interface optimisation, produces very similar results (about 2.5% worse) equally rapidly *and* removes most of the imbalance. This suggests that the hybrid approach is an effective solution to the parallel partition optimisation problem and this is especially true in the light of recent work which suggests that the scalability of a domain decomposition based solver can be seriously affected by even small imbalances in processor loading, [18, 19]. The results are also compared with another state-of-the-art partitioning tool, ParMETIS, [17], and shown to be of higher quality (about 11% better) although taking longer to compute.

[2]Mega-Flops or millions of floating point operations are a common measure of performance

mesh	$P = 16$		$P = 32$		$P = 64$		$P = 128$	
	C_S	C_S/C_P	C_S	C_S/C_P	C_S	C_S/C_P	C_S	C_S/C_P
4elt	993	0.93	1675	1.00	2707	0.99	4281	0.99
t60k-n	1817	1.04	2820	0.96	4379	1.00	6416	0.97
t60k-d	952	1.03	1560	0.99	2412	1.01	3581	1.02
dime20	1257	0.96	2400	1.06	3665	1.01	5548	1.03
t60k-f	4731	0.91	7780	0.98	12020	0.99	17970	0.99
fe-rotor	22050	0.97	36050	0.99	52764	1.04	72779	1.03
598a	28198	1.04	42248	1.00	60775	1.02	83385	1.01
mesh100	4420	0.95	7016	1.03	10478	1.05	14529	1.04
cyl3	10543	1.06	14915	1.02	21014	1.04	28106	1.02
fe-ocean	8904	1.04	14370	1.01	22867	1.05	32241	1.03
Average		0.99		1.01		1.02		1.01

Table 3: A comparison of cut-weight results for serial (S) and parallel (P) partitioners.

Table 3 shows a comparison of the cut-weight C between the serial (S) version of JOSTLE (described in [27]) and the parallel (P) partitioning results shown in Table 2. For each value of P, the first column shows the value of C for serial partitioning, C_S, while the second column shows the ratio of C for serial partitioning over that for parallel partitioning, C_S/C_P. Thus the value 1.04 (t60k-n, $P = 16$) means that the serial partitioning resulted in a cut-weight 1.04 times as large (or 4% larger) than that of the parallel partitioning. As can be seen, the serial results are very similar to the parallel ones, with a maximum of 9% difference (t60k-f, $P = 16$). The average difference in the quality ranges between 1% better and 2% worse over the different values of P with an overall average of just 0.77% depreciation for the serial results. This demonstrates that the parallel partitioner produces results of more or less the same quality as the serial partitioner, an impressive result considering the fact that the serial version has access to all of the data whilst a processor running the parallel code can only access its local portion of the mesh.

6.3 The impact of the initial distribution

We have suggested the requirement in §1.2 that the partitioner should be able to find a high quality partition *independent* of the existing partition. It is of interest to ask, therefore, whether this is possible and indeed what impact the initial distribution has on the outcome of the final partition. In Table 4 we compare four different initial distribution schemes for the two example meshes chosen from the test set in Table 1. The cyclic distribution assigns vertex i to processor p if i modulo $P = p$, i.e. vertex numbers $0, P, 2P, \ldots$ are given to processor 0, vertices $1, P + 1, 2P + 1, \ldots$ to processor 1, etc. The random distribution assigns them randomly (using the standard C library random number generator `drand48` which has a uniform distribution over the unit interval). The block distribution is the one used for the results in Table 2 and assigns the first V/P vertices to processor 0, etc., while the greedy algorithm is a (serial)

mesh	distrib.	C_0	C	$t(s)$	C_0	C	$t(s)$	C_0	C	$t(s)$
t60k-n	cyclic	86929	1821	2.08	88210	2863	1.51	89586	4476	1.45
t60k-n	random	84843	1737	2.06	87722	2863	1.55	89103	4385	1.43
t60k-n	block	6639	1753	0.88	7998	2930	0.80	10536	4378	0.80
t60k-n	greedy	2248	1758	0.44	3511	2908	0.48	5336	4476	0.61
cyl3	cyclic	432639	10299	14.71	445449	14796	9.07	451608	20564	6.86
cyl3	random	429109	10195	14.87	443501	14508	9.33	450678	20606	6.83
cyl3	block	351188	9976	12.31	375349	14639	7.91	388139	20211	6.34
cyl3	greedy	20014	10398	3.49	27858	14984	2.93	37442	20911	3.47

Table 4: Results showing the effect of different initial distributions (with cut-weight C_0) on the final partition quality (cut-weight C) and the parallel partitioning time, t.

graph-based implementation of Farhat's algorithm, [7]. Note that the cyclic, random and block distributions are all parallel input algorithms in the sense that the mesh can be read in from file in parallel, while the greedy algorithm requires the execution of a separate serial partitioner. The results show for each value of P the cut-weight of the initial distribution, C_0, the cut-weight of the final partition, C and the partitioning time in seconds, $t(s)$.

The results clearly demonstrate two things. Firstly, modulo a certain amount of 'noise' (inevitable for discrete optimisation algorithms such as these) with a maximum variation of 4.8% in the final cut-weight, the quality of the final partition is independent of the quality of the initial distribution. Thus the partitioning techniques are clearly seen to provide global rather than just local optimisation. Secondly, however, the partitioning time *is* strongly dependent on the initial distribution, with the poorly distributed results taking much longer to partition.

Regarding the initial distribution schemes, note that the block distribution can lead to a wide variation in initial cut-weight dependent on whether the mesh has been numbered with some form of structure (i.e. as in t60k-n, vertices which are close in index have a good chance of being neighbours in the graph) or not (i.e. as in cyl3, where no such relation appears to exist). Finally note that the cyclic scheme almost always (and always in Table 4) produces an initial cut-weight worse than the random distribution for precisely the opposite reason; if such a relation exists in the numbering it is destroyed by placing contiguous vertices on different processors.

7 Summary

We have discuss the problem of deriving and implementing parallel mesh partitioning algorithms for mapping unstructured meshes to parallel computers. We have also addressed the paradox therein, that to find a high quality partition of the mesh in parallel we require a partition of the mesh, and overcome it by using an optimisation strategy which can find a high quality partition even if the quality of the initial distribution is very poor. The basis of this strategy is to use a multilevel approach

ment algorithms (§3.3). Some example results are presented in §6 which show that the strategy can produce very high global quality partitions, very rapidly. The results are also compared with a similar multilevel serial partitioner in §6.2 and shown to be almost identical in quality. Finally, in §6.3, we have considered the impact of the initial distribution and demonstrated the global quality of the results and that the final partition quality is, modulo a certain amount of noise, independent of the initial distribution. However, as might be expected, the initial distribution strongly affects the partitioning time.

Much work continues in the field of mesh partitioning, for example to optimise different cost functions and it is of interest to ask how generic are the techniques described here. In the near future we hope to provide further results using the algorithms to minimise alternative objective functions such as subdomain aspect ratio (a parallel formulation of the ideas in [30]) or machine mapping (rather than just cut-edge weight).

References

[1] S. T. Barnard. PMRSB: Parallel Multilevel Recursive Spectral Bisection. Cray Research Inc., 1996.

[2] 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.

[3] P. Buch, J. Sanghavi, and A. Sangiovanni-Vincentelli. A Parallel Graph Partitioner on a Distributed Memory Multiprocessor. In *Proc. 5th IEEE Symp. on Frontiers of Massively Parallel Computation*, pages 360–366. IEEE, 1995.

[4] T. N. Bui and C. Jones. A Heuristic for Reducing Fill-In in Sparse Matrix Factorization. In R. F. Sincovec *et al.*, editor, *Parallel Processing for Scientific Computing*, pages 445–452. SIAM, 1993.

[5] R. Diekmann, B. Meyer, and B. Monien. Parallel Decomposition of Unstructured FEM-Meshes. *Concurrency: Practice & Experience*, 10(1):53–72, 1998.

[6] P. Diniz, S. Plimpton, B. Hendrickson, and R. Leland. Parallel Algorithms for Dynamically Partitioning Unstructured Grids. In D. Bailey *et al.*, editor, *Parallel Processing for Scientific Computing*, pages 615–620. SIAM, 1995.

[7] C. Farhat. A Simple and Efficient Automatic FEM Domain Decomposer. *Comput. & Structures*, 28(5):579–602, 1988.

[8] C. Farhat and M. Lesoinne. Automatic Partitioning of Unstructured Meshes for the Parallel Solution of Problems in Computational Mechanics. *Internat. J. Numer. Methods Engng.*, 36:745–764, 1993.

problems. *Theoret. Comput. Sci.*, 1:237–267, 1976.

[10] A. Gupta. Fast and effective algorithms for graph partitioning and sparse matrix reordering. *IBM Journal of Research and Development*, 41(1/2):171–183, 1996.

[11] B. Hendrickson. Graph Partitioning and Parallel Solvers: Has the Emperor No Clothes? In A. Ferreira and J. Rolim, editors, *Proc. Irregular '98: Parallel Algorithms for Irregularly Structured Problems*, volume 1457 of *LNCS*, pages 218–225. Springer, 1998.

[12] B. Hendrickson and K. Devine. Dynamic Load Balancing in Computational Mechanics. (to appear in *Comput. Meth. Appl. Mech. Engrg.*).

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

[14] B. Hendrickson and R. Leland. A Multilevel Algorithm for Partitioning Graphs. In S. Karin, editor, *Proc. Supercomputing '95*, New York, NY 10036, 1995. ACM Press.

[15] Y. C. Hu, S.-H. Teng, and S. Lennart-Johnsson. A Data-Parallel Implementation of the Geometric Partitioning Algorithm. In M. Heath *et al.*, editor, *Parallel Processing for Scientific Computing*. SIAM, Philadelphia, 1997.

[16] M. T. Jones and P. E. Plassmann. Parallel Algorithms for the Adaptive Refinement and Partitioning of Unstructured Meshes. In *Proc. Scalable High Performance Comput. Conf. '94*, pages 478–485. IEEE, 1994.

[17] G. Karypis and V. Kumar. A Coarse-Grain Parallel Formulation of Multilevel k-way Graph Partitioning Algorithm. In M. Heath *et al.*, editor, *Parallel Processing for Scientific Computing*. SIAM, Philadelphia, 1997.

[18] D. E. Keyes. How Scalable is Domain Decomposition in Practice? (submitted to Proc. Int. Conf. Domain Decomposition Methods, Greenwich 1998).

[19] D. E. Keyes, D. K. Kaushik, and B. F. Smith. Prospects for CFD on Petaflops Systems. In M. Hafez and K. Oshima, editors, *CFD Review 1998*, pages 1079–1096, Singapore, 1998. World Scientific.

[20] R. Lohner, R. Ramamurti, and D. Martin. A Parallelizable Load Balancing Algorithm. AIAA-93-0061, American Institute of Aeronautics and Astronautics, Washington, DC, 1993.

[21] G. L. Miller, S.-H. Teng, W. Thurston, and S. A. Vavasis. Geometric Separators for Finite-Element Meshes. *SIAM J. Sci. Comput.*, 19(2):364–386, 1998.

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

Repartitioning of Adaptive Meshes. *J. Par. Dist. Comput.*, 47(2):109–124, 1997.

[24] H. D. Simon. Partitioning of Unstructured Problems for Parallel Processing. *Computing Systems Engrg.*, 2:135–148, 1991.

[25] H. D. Simon, A. Sohn, and R. Biswas. HARP: A Dynamic Spectral Partitioner. *J. Par. Dist. Comput.*, 50(1/2):83–103, 1998.

[26] D. Vanderstraeten and R. Keunings. Optimized Partitioning of Unstructured Computational Grids. *Internat. J. Numer. Methods Engng.*, 38:433–450, 1995.

[27] C. Walshaw and M. Cross. Mesh Partitioning: a Multilevel Balancing and Refinement Algorithm. Accepted by *SIAM J. Sci. Comput.* (originally published as Univ. Greenwich Tech. Rep. 98/IM/35), 1998.

[28] C. Walshaw and M. Cross. Dynamic Mesh Partitioning and Load-Balancing for Parallel Computational Mechanics Codes. In B. H. V. Topping, editor, *Parallel & Distributed Processing for Computational Mechanics*. Saxe-Coburg Publications, Edinburgh, 1999. (Proc. Euro-CM-Par, Weimar, Germany, 1999).

[29] C. Walshaw and M. Cross. Parallel Optimisation Algorithms for Multilevel Mesh Partitioning. To appear in *Parallel Comput.* (originally published as Univ. Greenwich Tech. Rep. 99/IM/44), 1999.

[30] C. Walshaw, M. Cross, R. Diekmann, and F. Schlimbach. Multilevel Mesh Partitioning for Optimising Domain Shape. To appear in *Int. J. High Performance Comput. Appl.* (originally published as Univ. Greenwich Tech. Rep. 98/IM/38), 1998.

[31] C. Walshaw, M. Cross, and M. Everett. Parallel Dynamic Graph Partitioning for Adaptive Unstructured Meshes. *J. Par. Dist. Comput.*, 47(2):102–108, 1997.