Parallel Optimisation Algorithms for Multilevel Mesh Partitioning

C. Walshaw and M. Cross
Centre for Numerical Modelling and Process Analysis,
University of Greenwich, London, SE18 6PF, UK.
email: C.Walshaw@gre.ac.uk

Mathematics Research Report 99/IM/44

Abstract

Three parallel optimisation algorithms for use in the context of multilevel graph partitioning for unstructured meshes are described. The first, interface optimisation reduces the problem to a set of independent problems in interface regions. Alternating optimisation is restriction of this technique in which mesh entities are only allowed to migrate between subdomains in one direction. The third treats the gain as a potential field and uses the concept of relative gain for selecting appropriate vertices to migrate. The results are compared and seen to produce very high global quality partitions, very rapidly. The results are also compared with another state-of-the-art partitioning tool and shown to be of higher quality although taking longer to compute.

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

1 Introduction

Many of todays computational modelling challenges are of a size and complexity that requires the solution to be calculated in parallel on an unstructured mesh. Distributing the mesh across a parallel computer so that the computational load is evenly balanced and the data locality maximised is known as mesh partitioning and is often achieved by partitioning a graph corresponding to the communication requirements of the mesh. A particularly popular and successful class of algorithms which address this partitioning problem are known as multilevel algorithms. They usually combine a graph contraction algorithm which creates a series of progressively smaller and coarser graphs together with a local optimisation method which, starting with the coarsest graph, refines the partition at each graph level. In this paper we present three parallel optimisation algorithms for refining a partition and if necessary balancing the load. We also present an enhancement of the technique which uses imbalance to achieve higher quality partitions.

In particular, the algorithms described in this paper are designed to address the three problems that arise in partitioning of unstructured finite element and finite volume meshes. Specifically 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 meshes in which the computational workload for each mesh entity can vary with time or even 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 graph which may be neither load-balanced nor optimally partitioned. One way of dealing with this is to ship the graph back to some host processor, run a serial static 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 graph 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 graph (which may even be optimal) already exists, so it makes sense to reuse this as a starting point for repartitioning, [29, 36]. 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 graph is already distributed, it is a natural strategy to repartition it *in situ*.

In this paper we concentrate on the on the static case, (i), where the mesh is initially read in from file in parallel giving a crude but fast initial distribution. However, the techniques described contain a load-balancing component and seem well able to handle the static and dynamic load-balancing problems, (ii) & (iii), and this has been demonstrated for one of the algorithms in the dynamic case, [3, 36].

1.1 Overview

This paper contains a synopsis of research at the University of Greenwich that has taken place over the past five years into parallel optimisation algorithms for multilevel mesh partitioning. Several new ideas, not previously published before, are presented whilst other techniques are summarised. Related work in the area is discussed in §1.3.

To introduce the subject, in Section 2 we describe the multilevel paradigm and give a summary of a new enhancement, the idea of a multilevel balancing schedule (previously used with a serial multilevel partitioner in [33]). In Section 3 we then describe three different parallel optimisation algorithms which both balance a partition of the graph to within some given tolerance and also refine the quality in terms of the weight of cut edges. Section 4 contains a description of the (complex) parallel implementation of the partitioner. In Section 5 we present results and comparisons of the three optimisation algorithms together with a hybrid scheme and a comparison with another state-of-the-art parallel partitioner. Finally in Section 6 we draw some conclusions and present some ideas for further investigation.

1.2 Notation and Definitions

Let $G = G(V, E)$ be an undirected graph of vertices V, with edges E which represent the data dependencies in the mesh. The graph vertices can either represent mesh nodes (the nodal graph), mesh elements (the dual graph), a combination of both (the full or combined graph) or some special purpose representation to model the data dependencies in the mesh. We assume that both vertices and edges are 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* on G which we shall refer to as $G_\pi = G_\pi(S, L)$; there is an edge or link (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 $v_2 \in S_q$ and 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 $|E_c| = |L|$). 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 is assigned a subdomain S_p.

The definition of the graph partitioning problem is to find a partition which evenly balances the load or 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* 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). 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 §3.2 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.

1.3 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, [31], 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 [4, 7, 8, 27], 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 the parallel graph partitioner ParMETIS of Karypis & Kumar, [23, 29]. This uses an alternating tolerance-based optimisation algorithm similar to the one described in §3.6 (although we have additionally enhanced the algorithm with the use of a multilevel balancing schedule, §2.5, and by incorporating flow 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 here, 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 (described in Section 4). The effects of these different strategies on the optimisation are discussed in §5.3.

2 The multilevel paradigm

In recent years it has been recognised that an effective way of both speeding up partition refinement and, perhaps more importantly giving it 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 and has been successfully developed as a strategy for overcoming the localised nature of the Kernighan-Lin (and other) algorithms. The multilevel idea was first proposed by Barnard & Simon, [2], as a method of speeding up spectral bisection and improved by Hendrickson & Leland, [15] who generalised it to encompass local refinement algorithms.

2.1 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, [15]. The idea is to find a maximal independent subset of graph edges and then collapse them. The set is independent because no two edges in the set are incident on the same vertex (so no two edges in the set are adjacent), and maximal because 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 either end of it are merged to form a new vertex $v \in V_{l+1}$ with weight $|v| = |u_1| + |u_2|$. 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 simple way to construct a maximal independent subset of edges in serial is to visit the vertices of the graph in a random order and pair up or match unmatched vertices with a unmatched neighbour. It has been shown, [21], that it can be beneficial to the optimisation to collapse the most heavily weighted edges and our matching algorithm uses this heuristic.

2.1.1 Local matching

To carry out the matching, each processor maintains a double linked list of vertices that it owns which has internal vertices first and boundary vertices at the end of the list. Within the two sections the internal & boundary vertices are randomly ordered. Starting at the head of the list the each vertex is matched with a random unmatched local neighbour and both vertices are removed from the list. If a vertex has no unmatched local neighbours it is matched with itself and removed. So far this procedure is entirely parallel (with no dangers of conflicts since only local edges between core vertices are selected) but usually leaves a few boundary vertices who have no unmatched local neighbours but possibly some unmatched non-local neighbours.

2.1.2 Parallel matching

The simplest solution would be to terminate the matching at this point. However, in the worst case scenario, if the initial partition is particularly bad and most vertices have no local neighbours (for example a random partition), little or no matching may have taken place. We therefore continue the matching with an parallel iterative procedure which finishes only when there are no vertices unmatched. Within each iterative step, each processor first carries out a halo update to notify its neighbours which vertices are unmatched. It then visits its list of unmatched vertices, removing those with no unmatched neighbours, or selecting a neighbouring external vertex to match with. It next carries out another halo update is to notify neighbours of this selection and finally the unmatched vertices are visited again, matched whenever selection is mutual (i.e. when processor p has matched vertex $u \in B_p$ with $v \in B_q$, and processor q has matched vertex $v \in B_q$ with $u \in B_p$) and one of each pair of matching vertices is randomly selected to migrate to the processor owning the other. Finally, after all vertices have been matched, any that have been selected for migration are transferred.

Note that for algorithm to avoid selection cycles (i.e. u selects v, v selects w and w selects u) and thus locking, the selection of a match is made in a pseudo-random way. For each edge (u, v) a random number generator is seeded with the sum of the global indices of u and v and random number generated. The edge with the highest number is then selected. The reasoning behind this is explained in [20].

2.1.3 Construction of the coarsened graph

The construction of the coarsened graph is essentially completely parallel as, after migration, all vertices are now matched with a local neighbour. A global maximum operation is used to find m, the maximum number of clusters on any one processor. Each processor p then numbers the new vertices in the reduced graph that it is creating from $p \times m$ upwards and this ensures that the new global indices are unique.

2.2 The initial partition

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, [13], 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. Unlike Gupta, however, we do not carry out repeated interpolation/contraction cycles of the coarsest graphs to find a well balanced initial partition but instead, since our optimisation algorithms incorporate balancing, we commence on the interpolation/optimisation sequence immediately.

2.3 The global graph

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 [33], although essentially the techniques are very similar to those discussed here and in particular the serial optimisation algorithm which incorporates flow-based load-balancing, Kernighan-Lin style hill-climbing, Fiduccia-Mattheyses style bucket sorting and imbalance tolerance is similar to that described here as the interface optimisation algorithm, §3.5. 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 by the user) for the results in this paper is 20 vertices per processor.

2.4 Parallel partition interpolation

Having optimised a graph G_l, the partition must be interpolated onto its parent G_{l-1}. The interpolation itself is a trivial matter; if a vertex $v \in G_l$ is in subdomain S_p then the matched pair of vertices that it represents, v_1 & v_2, will be in S_p. However, during optimisation, we only migrate the vertices of G_l (and not the entire parent hierarchy) and so if v has migrated from its original processor then v_1 & v_2 will not be stored locally and may not even be stored on a neighbouring processor. Thus for each $v \in S_p$ which was not created locally, processor p must inform the originating processor (which can be easily determined by the global index of v – see above) where it now resides and the parent vertices must be migrated to S_p. Each processor therefore sends messages to the originators of all the vertices it owns, receives the corresponding messages about vertices it created and which have migrated away (by continually polling – waiting on incoming messages – until it knows the destination of all the vertices it created) and then migrates parent vertices to their child's new subdomain.

2.5 Multilevel balancing schedule

It has been noted previously (e.g. [22, 35]) that allowing a small amount of imbalance often leads to a higher partition quality. We also observe that one of the most attractive features of the multilevel paradigm is the way in which the partition quality (usually the number of cut edges) is refined gradually as the multilevel optimisation proceeds; i.e. after each refinement the partition quality of a given graph G_l is usually better than that of G_{l+1} (because there are more degrees of freedom). We combine both observations (imbalance can lead to higher partition quality and gradual refinement of quality being an attractive feature) by allowing a variable amount of *imbalance* which is reduced gradually as the multilevel optimisation proceeds. The idea is that by allowing a large imbalance in the coarsest graphs a better partition may be found than if balance was rigidly enforced, but that this imbalance will not cause degradation in the final partition of the finest graph if removed gradually throughout the multilevel procedure. Note particularly the second statement – if the finest graph starts the refinement with a high quality but poorly balanced partition, then much of the quality may be destroyed by balancing.

In order to talk about improving the balance gradually from one graph level to another, for each graph, G_l, let T_l be the target subdomain weight. If every subdomain, S_p, is not heavier than this target (i.e. $\max|S_p| \leq T_l$) then we say that the graph is sufficiently balanced and the optimisation can concentrate on refinement alone (so long as the balance is not destroyed). However, if $\max|S_p| > T_l$, then the optimisation must concentrate on balancing (with some regard to refinement). Clearly this series $\{T_l\}$ is an arbitrary heuristic, but it must be determined with two caveats:

- if it ascends too rapidly, the balance inherited by G_l from G_{l+1} may cause the partition quality to be lost in trying to attain T_l.
- if it ascends too slowly, the benefits for the partition quality of having a high imbalance tolerance may never be seen.

In [33] we derive (and report results from) different balancing schedules but here use the most successful formula from [33] and set $T_l = \theta_l \overline{S}$, where $\overline{S} = \lceil |V|/P \rceil$ is just the optimal subdomain weight (see §1.2) and

$$\theta_l = \left[1 + 2 \left(\frac{P}{N_{l-1}} \right)^{\frac{1}{2}} \right]$$

where N_{l-1} is the number of vertices in G_{l-1}, the parent graph of G_l. In other words a graph G_l is considered balanced if the imbalance is less than $\theta_l = 1 + 2(\frac{P}{N_{l-1}})^{\frac{1}{2}}$ for $l > 0$. For the final (and original) graph, G_0, which has no parent, we can either set $\theta_0 = 1$ to aim for perfect balancing or, as is often the case, e.g. [22], allow a slight imbalance. For the results in this paper we have chosen to set $\theta_0 = 1.05$ and then we set $\theta_l = \max(\theta_0, 1 + 2(\frac{P}{N_{l-1}})^{\frac{1}{2}})$ for $l > 0$.

3 Three balancing and refinement optimisation algorithms

In this section we describe and compare three parallel iterative optimisation algorithms all of which combine load-balancing and partition quality refinement. Initially we describe the concepts of load-balancing, gain & preference functions and bucket sorting (Sections 3.1, 3.2 and 3.3, respectively) and then in §3.4 we describe the outer iterative loop of the optimisation common to all three algorithms. The three algorithms are motivated in §3.4 and then described in detail in Sections 3.5, 3.6 and 3.7.

3.1 Load-balancing: calculating the flow

Given a graph partitioned into unequal sized subdomains, we need some mechanism for distributing the load equally. To do this we solve the load-balancing problem on the subdomain graph, G_π, (see §1.2)

in order to determine a *balancing flow*, a flow along the edges of G_π which balances the weight of the subdomains. By keeping the flow localised in this way, vertices are not migrated between non adjacent subdomains and hence (hopefully) the partition quality is not degraded (since a vertex migrating to a subdomain to which it is not adjacent is almost certain to have a negative gain).

This 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 introduced in [30] and some common strategies described. Much work has been carried out on parallel or distributed algorithms and, in particular, on diffusive algorithms, e.g. [5, 11], but here we use an elegant technique developed by Hu & Blake, [17, 19], which converges faster than diffusive methods. This method was derived to minimise the Euclidean norm of the transferred weight although it has recently been shown that this is true for all diffusion methods, [6, 18]. The algorithm simply involves solving the system $L\mathbf{x} = \mathbf{b}$ where L is the Laplacian of the subdomain graph:

$$L_{pq} = \begin{cases} \text{degree}(S_p) & \text{if } p = q \\ -1 & \text{if } p \neq q \text{ and } S_p \text{ is adjacent to } S_q \\ 0 & \text{otherwise} \end{cases}$$

and where $b_p = |S_p| - \overline{S}$ (the weight of S_p less the optimal subdomain weight). The weight to be transferred across edge (S_p, S_q) is then given by $x_p - x_q$. Note that this method is closely related to diffusive algorithms except that the diffusion coefficients are not fixed but are determined at each iteration by a conjugate gradient search. The algorithm is employed as suggested in [17], solving iteratively with a conjugate gradient solver. However, whilst this is an algorithm which is easily parallelised, we have found it more cost effective, for the numbers of processors which we used for testing (up to 128), to broadcast a copy of the subdomain graph around the parallel machine and duplicate the (serial) solution of the problem on every processor. Clearly for large numbers of processors this will not scale and so we have also implemented a fully parallel version (although it is not used for the tests here).

This algorithm (or, in principle, any other distributed load-balancing algorithm) generates a balancing flow across edges of the subdomain graph, i.e. F_{pq} along the edge (S_p, S_q), which is stored in memory. However, the optimisation algorithms which actually decide which vertices to move may not be able to satisfy the required flow instantly (because they are limited in the amount of weight they can transfer in one iteration) and thus decrement the values for F_{pq} by any weight that is actually transferred. Indeed for various reasons, the optimisation may exceed the required flow in which case the appropriate flow in the opposite direction is recorded (e.g. if $F_{pq} = 10$ but processor p actually transfers a weight of 15 then F_{pq} is set to 0 and F_{qp} set to 5). In this way a legitimate balancing flow is always maintained even if it takes many iterations to realise it. Note that in the following we require that flow is positive ($F_{pq} \geq 0$ and $F_{qp} \geq 0$) and unidirectional – i.e. either $F_{pq} = 0$ or $F_{qp} = 0$ (or both). If either of these requirements are false then the flow can be adjusted to meet them by setting $F_{pq} = F_{pq} - \min(F_{pq}, F_{qp})$ and $F_{qp} = F_{qp} - \min(F_{pq}, F_{qp})$.

Occasionally whilst optimisation is taking place vertex migration can cause the subdomain graph to change (e.g. two non-adjacent subdomains may become adjacent). If an edge disappears over which flow is scheduled to move the subdomain graph must be rebalanced although we speed this process up by adding the extraneous flow back into its source subdomain and rebalancing the graph from that point. The number of possible rebalances on any graph is restricted to avoid cyclic behaviour.

3.2 The gain and preference functions

A key concept in all three 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 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 and in [32], 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 [34], 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 gain 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 say, and for such vertices the preference is simply q. As a consequence processors only migrate vertices to neighbouring subdomains along edges of the subdomain graph.

3.3 Bucket sorting

The bucket sort is an essential tool for the efficient and rapid sorting of vertices by their gain. The concept was first suggested by Fiduccia & Mattheyses in [10] and the idea is that all vertices of a given gain g are placed together in a 'bucket' which is ranked g. Finding a vertex with maximum gain then simply consists of finding the (non-empty) bucket with the highest rank and picking a vertex from it. If the vertex is subsequently migrated from one subdomain to another then its gain and the gains of all its neighbours have to be adjusted and resorted by gain. Using a bucket sort for this operation simply requires recalculating the gains of the vertex and its neighbours and transferring them to the appropriate buckets, an essentially localised operation. If a bucket sort were not used and, say, the vertices were simply stored in a list in gain order, then the entire list would require resorting (or at least merge-sorting with the sorted list of adjusted vertices), an essentially $O(N)$ operation for every migration.

In our implementation each bucket is (as usual) represented by a double linked list of vertices (since vertices must be extracted from the list without having to search through it). However, we additionally prefer to sort the vertices by gain and then by weight. The reasoning behind this is simple: if, for example, a transfer of weight 3 between two subdomains is allowed then it is preferable to pick 3 vertices each of gain 1 and weight 1 rather than 1 vertex of gain 1 and weight 3. Conversely if a transfer of weight 2 is required then it is better to move 1 vertex of weight 2 and gain -1 rather than 2 vertices of weight 1 and gain -1. Thus we order the vertices primarily by gain and then by weight, lightest first for positive gains and heaviest first for negative gains. Rather than sorting the contents of each bucket we simply provide a different bucket for each gain/weight combination and so, if W represents the weight of the largest vertex in a given graph $\texttt{gain}(v)$ the gain of a vertex v, we rank v with the formula:

$$\text{rank}(v) = \begin{cases} \texttt{gain}(v) \times W + W - |v| & \text{if } \texttt{gain}(v) > 0 \\ \texttt{gain}(v) \times W + |v| - 1 & \text{otherwise} \end{cases}$$

which provides the desired ordering. The ranking is unique for each combination of $\texttt{gain}(v)$ and $|v|$ because $1 \leq |v| \leq W$ for all vertices v (it is assumed that $|v| > 0$) and so

$$\texttt{gain}(v) \times W \leq \text{rank}(v) < [\texttt{gain}(v) + 1] \times W$$

since

$$\text{rank}(v) \leq \texttt{gain}(v) \times W + W - 1 < \texttt{gain}(v) \times W + W.$$

Note that in the very coarse graphs at the top of the multilevel process, it is possible or even common to produce graphs with a wide range of vertex weights and potential gains. For this reason, rather than maintaining a sparse but potentially huge array of pointers to buckets, we store the non-empty buckets in a binary tree adding and deleting buckets as required. This tree structure may still be large but cannot exceed

the number of border vertices in the graph in size. In the sections below the term bucket tree will be used to refer to the binary tree of buckets.

3.4 Parallelising a serial iterative optimisation algorithm

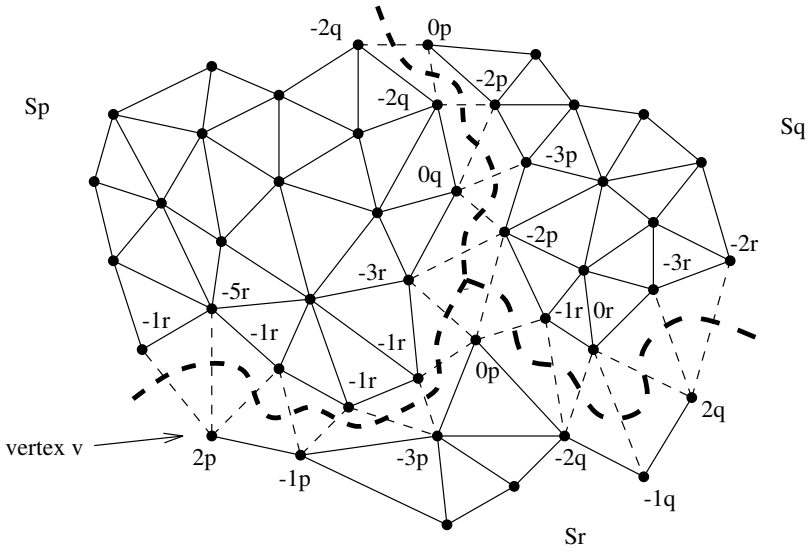

Figure 1: An example graph with subdomains S_p, S_q & S_r.

Consider the graph depicted in Figure 1. The subdomains S_p, S_q & S_r contain 21, 15 & 9 vertices respectively and so we can calculate a balancing flow to be 4 from S_p to S_q ($F_{pq} = 4$), 2 from S_p to S_r and 2 from S_q to S_r (note that this is not a unique solution). We can also 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_q 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 [33]) would consist of inner and outer iterative loops. The inner loop picks vertices (usually those in the bucket tree 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 the bucket tree is empty or when there is little prospect of further improvement with the vertices left in the bucket tree. The outer loop is simply repeated applications of the inner loop and terminates when no migration takes place within an inner loop.

```
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 2: The outer iterative loop.

The main problems in parallelising this procedure lie within the inner loop. Firstly, if the graph is dis-

tributed, 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 2. 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 4).

Figure 3: An example collision when vertices with postive gains migrated simultaneously result in an *increase* in cost.

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, [28]. Since all the processors are acting in parallel on the vertices that they own, simply moving vertices with the highest gain is not a satisfactory solution as it means that adjacent vertices may be swapped simultaneously (an 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 3 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. We refer to this sort of non-optimal behaviour as a collision.

An important part of our strategy for tackling this problem of collisions is to note that 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 4(a) show the six subdomain faces for the example graph in Figure 1. 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.

In the following three sections we describe in detail three different algorithms addressing this fundamental problem of collisions but to motivate them quickly 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 4(b) shows the three interface regions for the example graph in Figure 1.
- **Alternating Optimisation**. One of each pair of subdomain faces is selected and a tolerance-based algorithm chooses vertices from that face for migration (to its opposite face). A certain amount of imbalance tolerance is crucial for this algorithm to work. In the following iteration of the outer loop the alternate face is selected. Figure 4(c) shows an example of the three selected regions in a given iteration of the outer loop for the graph in Figure 1.
- **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 and an 'appropriate' proportion of vertices are moved highest relative gain first. Figure 4(d) shows the relative gains of the border vertices for the graph in Figure 1.

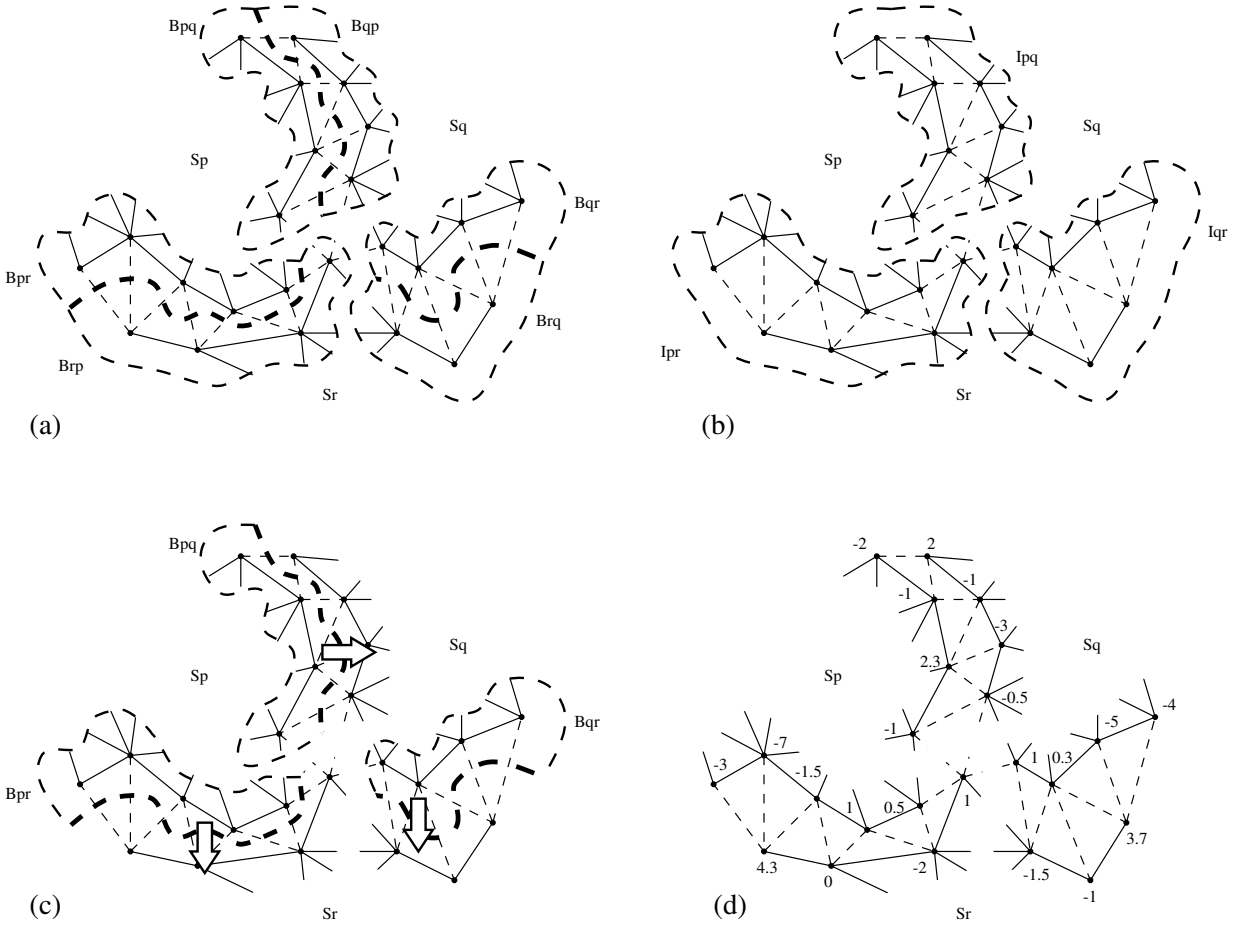

Figure 4: An example graph showing (a) subdomain faces; (b) interface regions for independent optimisation; (c) one of each pair of faces selected of alternating optimisation; and (d) relative gains as a 'potential field'.

3.4.1 Global convergence

Note that although all three algorithms tend to converge robustly (especially the interface algorithm) with the global cost, $|E_c|$, decreasing monotonically, unlike the serial algorithm, [33], this cannot be guaranteed. In order to prevent cyclic 'thrashing' therefore, the outer loop is terminated either after no migration has taken place during that iteration or after a couple of iterations if the cost has not decreased.

3.5 Interface optimisation

The interface optimisation technique works by treating each interface as an independent problem and executing a serial optimisation algorithm there. Thus for I_{pq}, the interface between S_p and S_q, one of the processors, p say, examines both its border and halo vertices to decide *not only* which of its vertices should migrate to neighbour q, *but also* which vertices should transfer from q to p. The distribution of interfaces amongst processors is carried out with a crude scheduling algorithm – simply that processor p handles I_{pq} if either $p < q$ and q is odd or if $p > q$ and q is even. All the interfaces are optimised simultaneously in parallel (although each processor may have to optimise a list of several) and then the processors pass lists to neighbours of non-local vertices which must be transferred (i.e. if processor p has optimised the interface

I_{pq} it must then tell processor q which vertices need to be transferred from S_q to S_p). Finally a bulk migration step occurs (as in Figure 2) and all vertices marked for migration are actually transferred between the processors.

For each interface the serial optimisation algorithm is very similar to that described in [33] and is initialised by inserting all the vertices in the interface into a bucket tree. The algorithm then proceeds by examining vertices highest gain first (by always picking vertices from the highest ranked bucket), testing whether the vertex is acceptable for migration (see below §3.5.1) and then transferring it out of the bucket tree. It terminates when the tree is empty although it may terminate early if the partition cost (i.e. the cut-weight) rises too far above the cost of the best partition found so far. This type of early termination is typical of KL type algorithms, [12, 16]; without it, the entire interface subgraph may be searched with diminishing prospect of finding a better solution along the search path.

3.5.1 Migration acceptance

Let T refer to the target weight for the graph (see §2.5). If the required flow from subdomain S_p to subdomain S_q is F_{pq}, a vertex v with weight $|v|$ (> 0) is accepted for migration from S_p to S_q (with weights $|S_p|$ and $|S_q|$) if

$$\begin{array}{ll} & \text{(a) } 2F_{pq} > |v| \\ \text{or} & \text{(b) } |S_q| + |v| \leq T \end{array} \tag{1}$$

These criteria are taken from our serial optimisation algorithm, [33], and reflect the aim of trying to balance the graph down to the target weight, T, and then keeping it there. Migration is thus accepted if (1a) it reduces the required flow or (1b) does not drive the imbalance above the target weight (although unlike the serial algorithm this cannot be guaranteed – see below §3.5.3).

When a vertex is accepted for migration, the gains of its neighbours, together with the flow and subdomain weight are modified as if the vertex had actually migrated (although global consistency is not maintained – see §3.5.3). For example, if $|S_p| = 23$, $|S_q| = 16$ & $F_{pq} = 3$, then if vertex v_1 with weight $|v_1| = 2$ is accepted for migration from S_p to S_q these values would be modified to $|S_p| = 21$, $|S_q| = 18$ & $F_{pq} = 1$. Further acceptance for migration from S_p to S_q of another vertex v_2 with weight $|v_2| = 2$ would then modify them to $|S_p| = 19$, $|S_q| = 20$ & $F_{pq} = -1$ or alternatively $F_{qp} = 1$.

3.5.2 Migration confirmation and hill-climbing

The algorithm uses a KL type hill-climbing strategy although it has only a limited effect because the interface regions are generally long and thin. As can be seen from (1) migrations can be *accepted* even if they increase the partition cost (i.e. have negative gain). As the interface optimiser runs, a record of the optimal partition of the interface achieved so far is maintained together with a list of vertices which have been accepted for migration since that value was attained. If subsequent migration finds a 'better' partition then the migration is *confirmed* and the list is reset. Once the interface optimisation has terminated, only those vertices whose migration has been confirmed are actually marked for migration in the bulk migration step.

To define a 'better' partition, let $\overline{\pi}$ represent the optimal partition of the interface found so far and π^i the subsequent partition after some iterations of the interface optimiser. Each partition has a cost associated with it, $C(\pi)$, (in this case just the total weight of cut edges across the interface), a flow $F(\pi) = \max(F_{pq}, F_{qp})$ and an imbalance which depends on $W(\pi)$, the weight of the largest subdomain involved in the interface, $W(\pi) = \max(|S_p|, |S_q|)$. Again let T represent the target weight for the graph (see §2.5). Denoting $C(\pi^i)$, $F(\pi^i)$ & $W(\pi^i)$ by C^i, F^i & W^i respectively (and similarly for $\overline{\pi}$) then π^i is confirmed as a new optimal

partition if:

$$\begin{array}{lll} & \text{(a) } W^i \leq T & \text{and } C^i < \overline{C} \\ \text{or} & \text{(b) } W^i \leq T & \text{and } C^i = \overline{C} \quad \text{and } W^i < \overline{W} \\ \text{or} & \text{(c) } T < W^i & \text{and } F^i < \overline{F} \end{array} \tag{2}$$

Condition (2c) simply states that, while the graph is unbalanced (i.e. $W^i > T$), any partition which reduces the flow is confirmed. Conditions (2a) & (2b) are more typical of KL type algorithms and confirm any partition which either improves on the optimal cost (2a) or improves on the optimal balance without raising the cost (2b).

3.5.3 Implementation issues

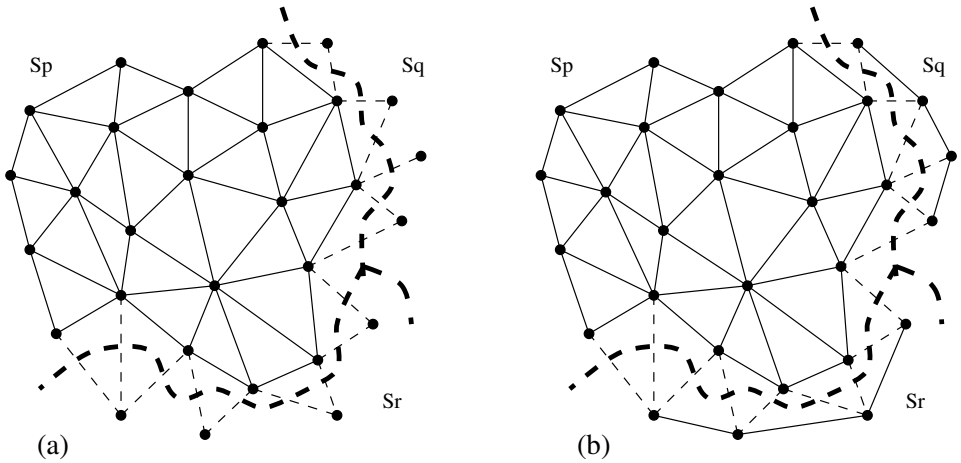

Figure 5: The normal halo (a) for subdomain S_p and (b) extended for the interface algorithm.

A number of issues arise in the parallel implementation of this algorithm:

- **Extension of halos.** In order to properly maintain the gains of halo vertices during the execution of the interface optimiser, a processor p needs to know about edges between halo vertices (i.e. if a vertex is accepted for migration then its neighbours' gains must be updated). These edges are not needed for the other two approaches (alternating and relative gain), nor for the parallel graph coarsening and this imposes an additional communication cost on the method. Figure 5(a) shows the halo normally available and 5(b) the extended halo required for this algorithm.
- **Additional communications.** The need for a processor p handling the optimisation of the interface I_{pq} to inform processor q of the results of that optimisation (prior to the bulk migration step in Figure 2) adds an additional communication step. On the other hand, during the halo update, processor p does not need to update the halo B_{pq} on processor q as processor q will not be requiring that information.
- **Data consistency.** For the interface I_{pq}, the subdomain weights, $|S_p|$ and $|S_q|$ used in (1) and in determining $\overline{W}$ and W^i for (2), are those set at the beginning of an outer iteration modified by any vertices accepted for migration. Unfortunately, there are multiple independent interface problems being solved simultaneously and so these values are not globally consistent. For this reason, unlike the serial version, the algorithm cannot guarantee that global imbalance will remain within the imbalance tolerance once the balance target has been attained. A possible modification would be to introduce a local tolerance T into (1) & (2), for example by distributing the tolerance around a subdomain's faces, but this has not been tested and in practice the algorithm almost always achieve the desired target balance.

3.6 Alternating optimisation

The basis of the alternating optimisation approach, independently suggested by Karypis & Kumar, [23], and Walshaw *et al.*, [36], can be seen as a restriction of the interface optimisation algorithm above. The strategy is to only allow vertices to migrate in one direction across any interface during a given iteration of the outer loop – see Figure 4(c). Thus for an interface, I_{pq}, if the chosen direction is from S_p to S_q then vertices are only allowed to migrate from the face B_{pq} of subdomain S_p to subdomain S_q; in the following outer iteration the direction is reversed. The decision of which face to choose is carried out with a crude scheduling algorithm similar to that given above.

Our implementation of the algorithm is almost identical to the interface algorithm described above with a few simplifications. Firstly, each processor p can deal with all of its active faces (those faces from which migration is allowed) in one go and thus inserts all the border vertices from the active faces into a single bucket tree. The processor then visits the vertices in order of highest gain (by always picking vertices out of the highest ranked bucket) and a vertex v with preference $\texttt{pref}(v) = q$ is then marked for migration if

$$\begin{array}{ll} & \text{(a) } 2F_{pq} > |v| \\ \text{or} & \text{(b) } |S_q| + |v| \leq T \end{array} \tag{3}$$

where T is the target weight for the graph (see §2.5) and F_{pq} is the required flow from subdomain S_p to subdomain S_q (exactly as in (1) – see §3.5.1). All visited vertices are removed from the bucket tree and, if accepted for migration, the flow F_{pq} and subdomain weights are adjusted and their unvisited neighbours have their gains updated as if the migration had taken place and are repositioned in the bucket tree. This retention of the bucket sort is an important feature of the algorithm since once a vertex is accepted for migration, its neighbours are likely to have higher gains.

This inner loop on each processor terminates when either the bucket tree is empty or when:

$$\begin{array}{ll} & \text{(a) all vertices with non-negative gain have been visited} \\ \text{and} & \text{(b) all outgoing flow requirements have been satisfied} \end{array} \tag{4}$$

3.6.1 Comparison with other methods

The alternating algorithm is simpler to implement than the full interface algorithm because it does not require the extension of halos nor the additional communication step (see §3.5.3). The problem of global consistency of data still arises as before though.

Our version of the alternating algorithm differs from that of Karypis & Kumar, [23], in three respects:–

- We incorporate balancing flow directly into the algorithm.
- More importantly, a certain amount of tolerated imbalance is crucial for the algorithm to operate and so the use of the multilevel balancing schedule (see §2.5) considerably enhances the algorithm (see also the results in §5.3). Thus, if no imbalance is allowed then T, the target weight, is just the optimal subdomain weight $\overline{S} = \lceil |V|/P \rceil$. Once the graph is balanced then $F_{pq} = 0$ everywhere (i.e. no balancing flow is required) and at least one and possibly all subdomains will realise the optimal weight, $|S_p| = \overline{S} = T$. For all such subdomains, neither of the migration acceptance conditions (3a) & (3b) can ever be satisfied and hence the optimisation will be severely, if not totally, limited.
- As with the other two algorithms presented here, vertex migration is actually realised, rather than being virtual (see §1.3) and we believe this enhances the performance of the algorithm.

3.7 Relative gain optimisation

The third algorithm is a somewhat different approach which has already been described in [36] but which we summarise here. Rather than using an algorithm running on the entire interface or on alternating faces, the concept is to think of gain as a force or potential field. From this we can calculate the relative gain on each border vertex and use this as a mechanism to avoid collisions. Figure 4(d) illustrates this; the relative gain of each vertex is shown and it can be seen that on each interface, the vertices with the largest relative gain which are the vertices most likely to migrate do not lie directly opposite each other. The algorithm is not as predictable as interface optimisation or the alternating scheme mentioned above, which can both predict exactly the improvement in cost for a bi-partition and fairly accurately for a multiway partition. However, although the relative gain gives no more than an indication of which vertices to move, in practice it works very effectively and collisions are rare.

As before the method uses the outer iterative loop shown in Figure 2. For each outer iteration, the optimisation algorithm is run concurrently by every processor p which estimates the load it wishes to migrate from every face B_{pq}, visits all its own border vertices calculating their relative gain and finally marks an appropriate weight of vertices for migration prioritised by that relative gain. The outer loop then continues with the parallel bulk migration.

3.7.1 Load to be transferred

The vertex weight to be transferred by any processor p from each of its faces B_{pq} to neighbour S_q is given by a simple formula based on both the flow and the total weight of vertices with positive gain. Firstly let g_{pq} be the total weight of vertices in B_{pq} with gain > 0 (and similarly for B_{qp} and g_{qp}). Then if $d = \max(g_{pq} - f_{pq} + g_{qp} - f_{qp}, 0)$, the load to be migrated from S_p to each neighbour S_q, is set to $a_{pq} = f_{pq} + d/2$.

To motivate this formula a little consider the following. First of all, the amount of load to be migrated, a_{pq}, is decided by satisfying any required flow, f_{pq}, and we assume that this takes place by migrating vertices with the highest positive gain. Thus, after the flow has been satisfied the amount of vertices with positive gain is approximately given by $G_{pq} = g_{pq} - f_{pq} + g_{qp} - f_{qp}$. It could be argued that this will be an underestimate if $f_{pq} > g_{pq}$, but in this case the scheme is cautious rather than reckless. At this point we wish, in a similar manner to the KL algorithm to swap vertices so that none with a positive gain remain. After some experimentation we have found that simply moving $G_{pq}/2$ from S_p to S_q and vice-versa, ensures fast and effective optimisation provided the vertices are chosen carefully.

3.7.2 Relative gain

The relative gain is determined as follows; for a vertex v in the face B_{pq}, let $\Gamma_q(v)$ be the set of vertices in B_{qp} adjacent to v, i.e. $\Gamma_q(v) = \{u \in B_{qp} : u \leftrightarrow v\}$. The relative gain of a vertex v is then defined as

$$\texttt{gain}(v,q) - \frac{\sum_{\Gamma_q(v)} \texttt{gain}(u,p)}{O[\Gamma_q(v)]}$$

(where $O[\Gamma_q(v)]$ represents the number of vertices in $\Gamma_q(v)$). Put more simply, the relative gain of a vertex v is just the gain of v less the average gain of opposing vertices, and gives an indication of which are the best vertices to move in order to avoid collisions. For example, in Figure 1 vertex v has a gain of 2 and 3 opposing vertices in B_{pr} with total gain $(-1-5-1) = -7$ and so v has a relative gain of $2-(-7/3) = 4.33$. Thus to prioritise the migration, for each subdomain S_p, vertices in each border B_{pq} are sorted by relative gain, largest first, and a weight of a_{pq} is migrated to S_q according to this ordering. The sorting carried out need not be a full sort since it is only necessary to determine the level of relative gain below which no vertices will be moved and we have implemented a simple set-based sort on this basis.

4 Parallel implementation

The software tool written at Greenwich to implement the optimisation techniques 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 parallel computers with communications performed with the Message Passing Interface 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.

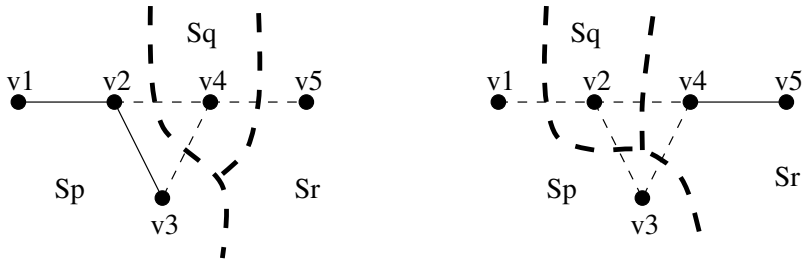

Figure 6: An example of vertex migration.

Vertex migration, the transfer of vertices from one distributed memory processor to another, is also a neighbourhood operation and is a non-trivial task to implement efficiently, [14], 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 6. 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.

Note that, since each vertex is identified with a unique global index, both halo updates 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 we use an array of binary tree structures which can be rapidly searched. Thus, given a global index, a processor can locate the appropriate binary tree (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

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

means that none of the trees should become too deep.

5 Results

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 §5.4 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); i.e. in the notation of §2.5, $\theta_0 = 1.05$

mesh	$\|V\|$	$\|E\|$	mesh type
4elt	15606	45878	2D nodal graph
t60k-nodal	30570	90575	2D nodal graph
t60k-dual	60005	89440	2D dual graph
dime20	224843	336024	2D dual graph
t60k-full	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-full is a combination of the t60k nodal graph and t60k dual graph, with the addition of edges between vertices from t60k-dual which represent mesh elements and the vertices from t60k-nodal which represent their nodes.

	$P = 16$		$P = 32$		$P = 64$		$P = 128$	
mesh	C_I	t_I	C_I	t_I	C_I	t_I	C_I	t_I
4elt	1070	0.49	1676	0.67	2728	0.84	4324	1.13
t60k-nodal	1753	0.87	2930	0.82	4378	0.79	6592	1.34
t60k-dual	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-full	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 interface algorithm showing the cut-weight C and parallel run-time in seconds t_s.

The results of the parallel multilevel partitioning using the interface optimisation algorithm from §3.5 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 (denoted C_I for the Interface algorithm), and the run-time in seconds, t_I (denoted similarly).

We do not show the final imbalance in the partition, but on average it was 1.047. Whilst it never exceeded the allowed imbalance of 1.05, this is relatively high and demonstrates how effectively the tolerance part of the algorithm uses any imbalance it is allowed (this was also noted in [33]).

In the following sections we compare the results with the two other optimisation algorithms and with a similar parallel multilevel mesh partitioner.

5.1 Comparison with alternating optimisation

	$P = 16$		$P = 32$		$P = 64$		$P = 128$	
mesh	C_A	C_A/C_I	C_A	C_A/C_I	C_A	C_A/C_I	C_A	C_A/C_I
4elt	1187	1.11	1794	1.07	2933	1.08	4542	1.05
t60k-nodal	1997	1.14	3161	1.08	4742	1.08	7036	1.07
t60k-dual	1021	1.10	1673	1.06	2534	1.06	3682	1.04
dime20	1441	1.10	2436	1.08	3781	1.04	5646	1.05
t60k-full	5846	1.13	8521	1.07	12989	1.07	19104	1.05
fe-rotor	24823	1.09	36571	1.01	52188	1.03	72802	1.03
598a	28101	1.04	43596	1.03	61381	1.03	83883	1.02
mesh100	4528	0.97	7274	1.07	10634	1.06	14633	1.05
cyl3	10622	1.06	15388	1.05	21322	1.05	28640	1.04
fe-ocean	10269	1.20	15792	1.11	25008	1.14	34119	1.09
Average		1.10		1.06		1.07		1.05

Table 3: A comparison of cut-weight results for alternating (A) & interface (I) optimisation.

Table 3 shows a comparison of the cut-weight C between alternating (A) & interface (I) optimisation. For each value of P, the first column shows the value of C for alternating optimisation, C_A, while the second column shows the ratio of C for alternating optimisation over that for interface optimisation, C_A/C_I. The value 1.11 (4elt, $P = 16$) means that alternating optimisation resulted in a cut-weight 1.11 times as large (or 11% larger) than that of interface optimisation. As can be seen, with one exception (mesh100, $P = 16$), the results for alternating optimisation are always worse and can be up to 20% larger (fe-ocean, $P = 16$). The average difference in the quality ranges between 10% and 5% over the different values of P with an overall average of 6.8% depreciation in quality. Although this does not demonstrate a dramatic improvement for the interface optimisation, if, as is commonly assumed, the parallel communications overhead in the underlying solver is related to the cut-weight, this could have a significant effect on the total parallel runtime (particularly for a static partition employed over many iterations).

The average final imbalance was 1.011 (with a maximum of 1.032). Once again this does not exceed the imbalance tolerance of 1.05 and is considerably better than the interface optimisation algorithm.

	$P = 16$		$P = 32$		$P = 64$		$P = 128$	
mesh	t_A	t_A/t_I	t_A	t_A/t_I	t_A	t_A/t_I	t_A	t_A/t_I
4elt	0.44	0.90	0.56	0.84	0.77	0.92	1.04	0.92
t60k-nodal	0.84	0.97	0.75	0.91	0.73	0.92	1.24	0.93
t60k-dual	0.50	0.93	0.48	0.92	0.66	0.94	1.21	0.92
dime20	1.44	0.97	1.20	1.03	1.14	0.90	1.67	0.85
t60k-full	3.20	0.92	3.23	0.97	2.62	0.91	2.93	0.92
fe-rotor	7.61	0.91	6.27	0.87	5.07	0.77	6.85	0.85
598a	16.51	0.96	11.57	0.92	8.95	0.86	9.39	0.89
mesh100	2.62	0.92	2.38	0.99	2.23	0.85	3.26	0.88
cyl3	11.98	0.97	7.68	0.96	5.92	0.93	6.14	0.91
fe-ocean	5.95	0.91	4.32	0.94	3.06	0.85	3.52	0.82
Average		0.94		0.93		0.89		0.89

Table 4: A comparison of parallel timing results for alternating (A) & interface (I) optimisation.

Table 4 meanwhile shows a comparison in the same format of the parallel timing results t for alternating & interface optimisation. Here we can see that alternating optimisation is almost always faster than interface optimisation (up to 23% faster (fe-rotor, $P = 64$) with the average improvement in speed ranging between 6% to 11% over the different values of P and an overall average of 8.9%. However, although the partitioner should be ideally as fast as possible, this variation in partitioning time would generally be insignificant for most solvers which may run for several minutes or even hours (once again, especially for a static partition).

5.2 Comparison with relative gain optimisation

	$P = 16$		$P = 32$		$P = 64$		$P = 128$	
mesh	C_R	C_R/C_I	C_R	C_R/C_I	C_R	C_R/C_I	C_R	C_R/C_I
4elt	1147	1.07	1922	1.15	2991	1.10	4730	1.09
t60k-nodal	1872	1.07	3096	1.06	4768	1.09	7163	1.09
t60k-dual	1037	1.12	1651	1.05	2516	1.06	3787	1.07
dime20	1372	1.05	2454	1.09	3896	1.07	5799	1.08
t60k-full	5445	1.05	8304	1.05	12907	1.07	19233	1.06
fe-rotor	23655	1.04	38460	1.06	52951	1.05	75898	1.07
598a	28738	1.06	42300	1.00	62240	1.04	85913	1.04
mesh100	4615	0.99	7459	1.10	11422	1.14	15637	1.12
cyl3	10959	1.10	15692	1.07	21919	1.08	30082	1.09
fe-ocean	10653	1.25	16849	1.19	25445	1.16	36391	1.16
Average		1.08		1.08		1.09		1.09

Table 5: A comparison of cut-weight results for relative gain (R) & interface (I) optimisation.

The cut-weight figures for relative gain optimisation are shown in Table 5 and compared with interface optimisation as before. Once again, with the same exception (mesh100, $P = 16$), the results for relative gain optimisation are always worse and can be up to 25% larger (fe-ocean, $P = 16$) with an overall depreciation of 8.3%. The average final imbalance was 1.002 (with a maximum of 1.008); this is considerably better than both the interface and alternating optimisation algorithms. We do not show the timing comparison but relative gain optimisation was on average 7% faster than interface optimisation.

5.2.1 A hybrid algorithm

During development work on these algorithms it was noticed that both alternating & relative gain optimisation tend to converge to a good solution very rapidly (usually in less than 10 iterations) but then have difficulty in resolving one or more small areas of the partition and ended up cyclically swapping a few vertices backwards and forwards between subdomains. It is for this reason that the termination criteria of §3.4.1 were included. Interface optimisation, on the other hand, converges even more rapidly (although at greater cost per iteration).

	$P = 16$		$P = 32$		$P = 64$		$P = 128$	
mesh	C_{R+I}	C_{R+I}/C_I	C_{R+I}	C_{R+I}/C_I	C_{R+I}	C_{R+I}/C_I	C_{R+I}	C_{R+I}/C_I
4elt	1093	1.02	1774	1.06	2870	1.05	4450	1.03
t60k-nodal	1875	1.07	2887	0.99	4537	1.04	6786	1.03
t60k-dual	964	1.04	1610	1.02	2396	1.01	3609	1.02
dime20	1255	0.96	2315	1.03	3688	1.02	5493	1.02
t60k-full	5037	0.97	8090	1.02	12271	1.01	18580	1.02
fe-rotor	23921	1.05	36593	1.01	51432	1.02	73477	1.04
598a	27810	1.03	41187	0.98	60080	1.00	83294	1.01
mesh100	4456	0.96	7113	1.05	10219	1.02	14473	1.04
cyl3	10137	1.02	14456	0.99	20381	1.01	27557	1.00
fe-ocean	9688	1.13	15477	1.09	23951	1.10	33688	1.07
Average		1.03		1.02		1.03		1.03

Table 6: A comparison of cut-weight results for the hybrid relative gain/interface (R+I) & interface (I) optimisation.

This prompted the idea of a hybrid approach, using either alternating or (as here) relative gain optimisation and once the cyclic behaviour appears (when the global cost starts to oscillate) to carry out an iteration using the interface optimiser to resolve the areas where cycling is occurring. This strategy turned out to be very effective; denoting the cut-weight for this hybrid algorithm C_{R+I} (for relative plus interface), the results are shown in Table 6 and compared with interface optimisation as before. Here we see that the hybrid algorithm is often better than interface optimisation although on average about 2.5% worse. We do

not show the timings here, but they were broadly similar with the hybrid algorithm just marginally better (about 0.4% on average). Generally though, the hybrid algorithm is slightly faster for large numbers of processors (7% on average for $P = 128$) and slower for smaller numbers (4% on average for $P = 16$). More interestingly, the hybrid algorithm had an average imbalance of just 1.007, because of the ability of relative gain optimisation to remove almost all imbalance.

5.3 Comparison with ParMETIS

	$P = 16$		$P = 32$		$P = 64$		$P = 128$	
mesh	C_M	C_M/C_I	C_M	C_M/C_I	C_M	C_M/C_I	C_M	C_M/C_I
4elt	1184	1.11	1832	1.09	2940	1.08	4631	1.07
t60k-nodal	1948	1.11	3109	1.06	4738	1.08	7124	1.08
t60k-dual	968	1.05	1706	1.08	2481	1.04	3773	1.07
dime20	1474	1.13	2339	1.04	3668	1.01	5704	1.06
t60k-full	5364	1.03	8657	1.09	13353	1.10	19960	1.10
fe-rotor	23284	1.02	37324	1.03	52577	1.04	74484	1.05
598a	30440	1.13	44059	1.04	63460	1.06	86841	1.06
mesh100	5133	1.10	7745	1.14	11067	1.11	15261	1.10
cyl3	11542	1.16	15964	1.09	22168	1.10	29677	1.07
fe-ocean	12692	1.49	20252	1.43	27580	1.26	37843	1.20
Average		1.13		1.11		1.09		1.09

Table 7: A comparison of cut-weight results for ParMETIS & interface (I) optimisation.

We have also checked the results against another state-of-the-art parallel partitioner ParMETIS, [23]. As discussed in §1.3 & §3.6.1, ParMETIS is a multilevel partitioner using an alternating optimisation algorithm, although without the addition of a multilevel balancing schedule and with virtual migration rather than realised migration. The cut-weight figures for ParMETIS, C_M, are shown in Table 7 and compared with the interface optimisation as previously. As can be seen, without exception, the results for ParMETIS are always worse than the interface algorithm and can be 49% larger (fe-ocean, $P = 16$), although it seems to perform particularly badly on this mesh. The average difference in the quality ranges between 13% and 9% over the different values of P. It is difficult to say exactly what the cause of this difference is but we believe that the virtual migration does hinder the optimisation slightly, particularly for vertices adjacent to more than one subdomain.

	$P = 16$		$P = 32$		$P = 64$		$P = 128$	
mesh	t_M	t_M/t_I	t_M	t_M/t_I	t_M	t_M/t_I	t_M	t_M/t_I
4elt	0.25	0.51	0.39	0.58	0.46	0.55	0.54	0.48
t60k-nodal	0.33	0.38	0.42	0.51	0.51	0.65	0.67	0.50
t60k-dual	0.24	0.44	0.27	0.52	0.40	0.57	0.63	0.48
dime20	0.45	0.30	0.39	0.33	0.48	0.38	0.72	0.37
t60k-full	0.68	0.20	0.74	0.22	0.88	0.31	1.09	0.34
fe-rotor	1.15	0.14	1.08	0.15	1.37	0.21	2.54	0.32
598a	1.97	0.11	1.61	0.13	1.79	0.17	2.57	0.24
mesh100	0.78	0.27	0.86	0.36	1.09	0.42	1.52	0.41
cyl3	2.02	0.16	1.72	0.22	2.13	0.34	3.37	0.50
fe-ocean	0.71	0.11	0.71	0.15	0.69	0.19	0.73	0.17
Average		0.26		0.32		0.38		0.38

Table 8: A comparison of parallel timing results for ParMETIS & interface (I) optimisation.

On the other hand the use of virtual migration, without the need to possibly remap large amounts of the graph does mean that ParMETIS has faster execution time than JOSTLE. This is particularly true when the initial distribution of the data is poor (see §5.4) but not so relevant when this is not the case (for example in dynamic repartitioning, [36]). Table 8 shows this in a comparison of the parallel timing results. Here we see that ParMETIS is always faster than JOSTLE, on average taking 33% of the time to partition (i.e. in other words it is about 3 times faster on average). However, the difference is more marked for the mesh cyl3

which can be seen in §5.4 to have a very poor initial distribution and less marked for the mesh t60k-nodal which, because of the way the vertices are numbered, has a good initial distribution.

In summary these results demonstrate an optimisation rule of thumb, that the longer an algorithm takes to optimise the better the results it gets. Assuming that the parallel overhead in the solver is related to the cut-weight, the question must then be asked whether the gain in runtime from having a better partition outweighs the additional cost in partitioning time. This is obviously very machine and application dependent, but we would remark that the partitioning times are very small, always less than 20 seconds which in our experience is insignificant compared with the runtime of a typical parallel unstructured mesh application.

The average imbalance for ParMETIS was 1.032 and occasionally it exceeded the allowed imbalance of 1.05 (dime20, $P = 64$, imbalance $= 1.097$; 4elt, $P = 128$, imbalance $= 1.066$; fe-rotor, $P = 128$, imbalance $= 1.053$). As an experiment we tried adjusting the ParMETIS imbalance tolerance to 1% ($\theta_0 = 1.01$)[2] to compare with the high quality load-balance figures of the hybrid algorithm (an average imbalance of just 1.007). As expected, because of the trade-off between load-balance and partition quality, this made the cut-weight results worse (14.5% worse than the interface algorithm and 11.6% worse than the hybrid algorithm on average), however ParMETIS was unable to achieve the desired balance and gave an average imbalance in the results of 1.030.

5.4 The impact of the initial distribution

	initial	$P = 16$			$P = 32$			$P = 64$			$P = 128$		
mesh	distrib.	C_0	C	t	C_0	C	t	C_0	C	t	C_0	C	t
t60k-nodal	cyclic	86929	1821	2.08	88210	2863	1.51	89586	4476	1.45	90521	6595	2.30
t60k-nodal	random	84843	1737	2.06	87722	2863	1.55	89103	4385	1.43	89838	6554	2.39
t60k-nodal	block	6639	1753	0.88	7998	2930	0.80	10536	4378	0.80	15457	6592	1.29
t60k-nodal	greedy	2248	1758	0.44	3511	2908	0.48	5336	4476	0.61	7752	6608	1.23
cyl3	cyclic	432639	10299	14.71	445449	14796	9.07	451608	20564	6.86	454647	27460	6.83
cyl3	random	429109	10195	14.87	443501	14508	9.33	450678	20606	6.83	454248	27614	7.52
cyl3	block	351188	9976	12.31	375349	14639	7.91	388139	20211	6.34	394861	27628	6.70
cyl3	greedy	20014	10398	3.49	27858	14984	2.93	37442	20911	3.47	48152	27779	5.20

Table 9: 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.

It is of interest to ask what impact does the initial distribution have on the outcome of the final partition. In Table 9 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 all the previous tests and assigns the first V/P vertices to processor 0, etc., while the greedy algorithm is a (serial) graph-based implementation, [35], of Farhat's algorithm, [9]. 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.

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.

[2] by resetting `UNBALANCE_FRACTION` and `ORDER_UNBALANCE_FRACTION` in `defs.h` to 1.01

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-nodal, 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 9) 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.

6 Conclusions

We have described three parallel optimisation algorithms for use in the context of parallel multilevel partitioning for unstructured meshes. We have compared the results they generate and seen that the interface optimisation algorithm, §3.5, the one closest to our serial flow & tolerance algorithm, [33] (and indeed the original Kernighan-Lin algorithm, [24]) 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 we have shown that a hybrid algorithm, using relative gain with a final clean-up step of interface optimisation, produces very similar results 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, [25, 26]. We have also compared the algorithms with another state-of-the-art partitioning tool, ParMETIS, and shown that the results are of higher quality although taking longer to compute. In §5.4 we have demonstrated the global quality of the results and that (as 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, e.g. [34], 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 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] R. Biswas and L. Oliker. Experiments with Repartitioning and Load Balancing Adaptive Meshes. Tech. Rep. NAS-97-021, NASA Ames, Moffat Field, CA, 1997.

[4] 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.

[5] G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *J. Par. Dist. Comput.*, 7(2):279–301, 1989.

[6] R. Diekmann, A. Frommer, and B. Monien. Efficient Schemes for Nearest Neighbor Load Balancing. Technical report, Dept. Maths. Comp. Sci., Univ. Paderborn, Furstenallee 11, D-33102 Paderborn, Germany, May 1998.

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

[8] 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.

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

[10] 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, Piscataway, NJ, 1982.

[11] B. Ghosh, S. Muthukrishnan, and M. H. Schultz. Faster Schedules for Diffusive Load Balancing via Over-Relaxation. TR 1065, Department of Computer Science, Yale Univ., New Haven, CT 06520, USA, 1995.

[12] 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.

[13] 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.

[14] B. Hendrickson and K. Devine. Dynamic Load Balancing in Computational Mechanics. (to appear in Comp. Meth. Appl. Mech. Engrg.), 1998.

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

[16] B. Hendrickson and R. Leland. A Multilevel Algorithm for Partitioning Graphs. In *Proc. Supercomputing '95*, 1995.

[17] Y. F. Hu and R. J. Blake. An optimal dynamic load balancing algorithm. Preprint DL-P-95-011, Daresbury Laboratory, Warrington, WA4 4AD, UK., 1995.

[18] Y. F. Hu and R. J. Blake. The optimal property of polynomial based diffusion-like algorithms in dynamic load balancing. In K. D. Papailiou *et al.*, editor, *Computational Dynamics '98*, pages 177–183. Wiley, 1998.

[19] Y. F. Hu, R. J. Blake, and D. R. Emerson. An optimal migration algorithm for dynamic load balancing. *Concurrency: Practice & Experience*, 10(6):467–483, 1998.

[20] M. Jones and P. Plassman. A Parallel Graph Coloring Heuristic. *SIAM J. Sci. Stat. Comput.*, 14:654–669, 1993.

[21] G. Karypis and V. Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. TR 95-035, Dept. Comp. Sci., Univ. Minnesota, Minneapolis, MN 55455, 1995.

[22] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. TR 95-064, Dept. Comp. Sci., Univ. Minnesota, Minneapolis, MN 55455, 1995.

[23] 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.

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

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

[26] D. E. Keyes, D. K. Kaushik, and B. F. Smith. Prospects for CFD on Petaflops Systems. In M. Hafez *et al.*, editor, *CFD Review*. Wiley. (to appear).

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

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

[29] K. Schloegel, G. Karypis, and V. Kumar. Multilevel Diffusion Schemes for Repartitioning of Adaptive Meshes. *J. Par. Dist. Comput.*, 47(2):109–124, 1997.

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

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

[32] D. Vanderstraeten and R. Keunings. Optimized Partitioning of Unstructured Computational Grids. *Int. J. Num. Meth. Engng.*, 38:433–450, 1995.

[33] C. Walshaw and M. Cross. Mesh Partitioning: a Multilevel Balancing and Refinement Algorithm. Tech. Rep. 98/IM/35, Univ. Greenwich, London SE18 6PF, UK, March 1998.

[34] C. Walshaw, M. Cross, R. Diekmann, and F. Schlimbach. Multilevel Mesh Partitioning for Optimising Domain Shape. Tech. Rep. 98/IM/38, Univ. Greenwich, London SE18 6PF, UK, July 1998.

[35] C. Walshaw, M. Cross, and M. Everett. A Localised Algorithm for Optimising Unstructured Mesh Partitions. *Int. J. Supercomputer Appl.*, 9(4):280–295, 1995.

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