
ELSEVIER

Parallel Computing 26 (2000) 1635–1660

PARALLEL COMPUTING

www.elsevier.com/locate/parco

Parallel optimisation algorithms for multilevel mesh partitioning

C. Walshaw *, M. Cross

School of Computing and Mathematical Sciences, University of Greenwich, London SE18 6PF, UK

Received 12 February 1999; received in revised form 13 September 1999; accepted 13 September 1999

Abstract

Three parallel optimisation algorithms, for use in the context of multilevel graph partitioning of unstructured meshes, are described. The first, interface optimisation, reduces the computation to a set of independent optimisation problems in interface regions. The next, alternating optimisation, is a 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 partitioning tool and shown to be of higher quality although taking longer to compute. © 2000 Elsevier Science B.V. All rights reserved.

Keywords: 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

* Corresponding author.
E-mail address: c.walshaw@gre.ac.uk (C. Walshaw).

PII: S0167-8191(00)00046-6

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 problems of:

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

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

(iii) *dynamic load-balancing/partitioning* 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 cases (ii) and (iii), the initial data can be represented by a distributed graph which may be neither load-balanced nor optimally partitioned. However, it is fairly clear that this should be repartitioned *in parallel* rather than shipping the graph back to some host processor, e.g., [34]. On the other hand, it could be argued that case (i) can be handled by a serial algorithm (of which many exist) but this is unattractive for many reasons. Firstly, an $O(N)$ start-up cost for the mesh partitioning may not be acceptable if the solver will subsequently be 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. Finally, assuming that a parallel machine is available to run the solver, it makes sense to also use it for the initial partition.

With this in mind, our focus has always been to develop parallel optimisation-based partitioning algorithms (rather than serial direct algorithms) in order to handle all three problems at once. In this paper, we concentrate 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) and (iii), and this has been demonstrated for one of the algorithms in the dynamic case [3,34].

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. 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 [31]). 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. In Section 4, we present results

and comparisons of the three optimisation algorithms together with a hybrid scheme and a comparison with ParMETIS, another parallel partitioner [21]. Finally in Section 5, we draw some conclusions and present some ideas for further investigation. Related work in the area is discussed in Section 1.3.

The primary new development of the paper is the interface optimisation algorithm (Section 3.4) and the main focus is to compare it with two competing algorithms (to which some enhancements have been made) within the context of multilevel mesh partitioning. Unifying ideas behind all three algorithms are:

- The inclusion of a load-balancing component in order that the algorithms can successfully handle the three partitioning problems described above.
- The use of variable imbalance tolerances within a multilevel context to enhance the optimisation.

Some key parts of the paper are:

- The motivation behind different algorithms, Section 3.3.1.
- The impact of the initial distribution on the final partition, Section 4.4.

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 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; 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 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* 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 Section 3.2 for further discussion on this point. A more precise definition of the graph partitioning problem is therefore to find π such that $S_p \leqslant \overline{S}$ 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 [28], 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 [4,8,9,24], 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 and Kumar [21,26]. This uses an alternating tolerance-based optimisation algorithm similar to the one described in Section 3.5 (although we have additionally enhanced the algorithm with the use of a multilevel balancing schedule, Section 2.3, and by incorporating flow directly into the optimisation process). Perhaps the major difference in strategy 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 [32]). The effects of these different strategies on the optimisation are discussed in Section 4.3.

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 graphs 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 and Simon [2], as a method of speeding up spectral bisection and improved by both Hendrickson and Leland [17] and Bui and Jones [5], 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 graph contraction algorithm proposed by Hendrickson and Leland [16]. 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 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. Due to 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 [34].

2.2. The initial partition and 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 [14], 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 may involve 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 [31], 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, Section 3.4. 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.

2.3. Multilevel balancing schedule

It has been noted previously 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., $\max |S_p| \leqslant T_l$), 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, then 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, then the benefits for the partition quality of having a high imbalance tolerance may never be seen.

In [31], we derive (and report results from) different balancing schedules but here use the most successful formula from [31] and set $T_l = \theta_l \overline{S}$, where $\overline{S} = \lceil |V|/P \rceil$ is just the optimal subdomain weight (see Section 1.2) and

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

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 $\theta_l = 1 + 2(P/N_{l-1})^{1/2}$

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., [20], allow a slight imbalance. For the results in this paper we have chosen $\theta_0 = 1.05$ and then we set $\theta_l = \max(\theta_0, 1 + 2(P/N_{l-1})^{1/2})$ 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 (Section 3.1) and gain and preference functions (Section 3.2) and then in Section 3.3, we describe the outer iterative loop of the optimisation common to all three algorithms. The three algorithms are motivated in Section 3.3.1 and then described in detail in Sections 3.4–3.6.

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 Section 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, as it almost certainly would be if vertices were migrated to non-adjacent subdomains.

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 [27] and some common strategies described. Much work has been carried out on parallel or distributed algorithms and, in particular, on diffusive algorithms, e.g., [6,12]; here we use an elegant diffusive variant developed by Hu and Blake [19], with fast convergence. This method was derived to minimise the Euclidean norm of the transferred weight although it has recently been shown that all diffusion methods minimise this quantity [7,18]. The algorithm simply involves solving the system $L\mathbf{x} = \mathbf{b}$, where L is the Laplacian of the subdomain graph:

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

where degree(S_p) is the degree of (or number of edges incident on) the vertex S_p 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 [19], 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). Note finally that the Laplacian of any undirected graph contains a zero eigenvalue with the corresponding eigenvector $[1, 1, \ldots, 1]$ and the solution iterates are orthogonalised against this [19]. If any other singularities are detected (for example if the graph is disconnected), then the software will switch to another method, an intuitive and entirely localised distributed load-balancing algorithm due to Song [29].

The 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 ($F_{pq} \geqslant 0$ and $F_{qp} \geqslant 0$) 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, then 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

Key concepts 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 $\max_{r \in P} \texttt{gain}(v, r)$. In the event of ties, the choice is made on the basis of (a) maximum flow, $\max F_{pq}$, then (b) minimum subdomain weight, $\min |S_q|$ and finally, if all of these are tied, then a random choice is made. Throughout the following, vertices are only allowed to migrate to the subdomain to which their preference is set. Also it is usual for vertices to be sorted by gain using a

bucket sort [11], and in our implementation we use a binary tree of buckets to achieve this [32], referred to in the sections below as a *bucket tree*.

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., [15]) and, for example, in [30], 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 [33], 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.

3.3. Parallelising a serial iterative optimisation algorithm

Consider the graph depicted in Fig. 1. The subdomains S_p, S_q and S_r contain 21, 15 and 9 vertices, respectively, and so we can calculate a balancing flow to be 2 from

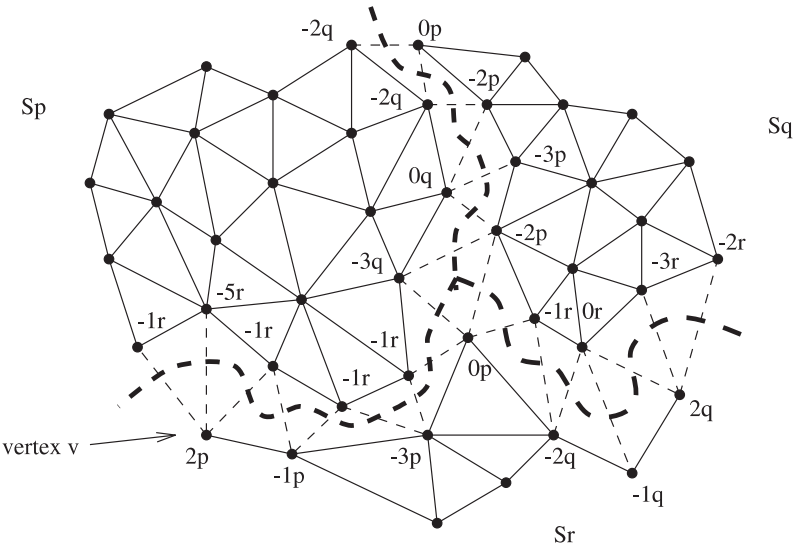

Fig. 1. An example graph with subdomains S_p, S_q and S_r.

S_p to S_q ($F_{pq} = 2$), 4 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) [22], type algorithm for optimising this partition (such as described in [31]) 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.

The main problems in parallelising this procedure lie within the inner loop. Firstly, if the graph is distributed, then 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 Fig. 2. It contains three communication steps, a halo update of the border vertices gain and preference values (where each processor communicates to update these values for the copies on the neighbouring processors), the migration of vertices to their neighbours and the global update of the `optimising` flag.

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, [25]. 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

```
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);
}
```

Fig. 2. The outer iterative loop.

Fig. 3. An example collision when vertices with positive gains migrated simultaneously result in an *increase* in cost.

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 Fig. 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. Note that collisions can occur despite the fact that the balancing flow is required to be unidirectional (see Section 3.1) because typically each iteration will involve the migration of weight F_{pq} from subdomain S_p to S_q to satisfy the balancing flow plus the additional equal and opposite migration exchange of weight W between S_p and S_q for optimisation purposes. This additional exchange is necessary because in a perfectly balanced system the balancing flow component, F_{pq} is zero and so without it optimisation could not occur.

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*. Fig. 4(a) shows the six subdomain faces for the example graph in Fig. 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.

3.3.1. Motivation

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. Fig. 4(b) shows the three interface regions for the example graph in Fig. 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. Fig. 4(c) shows an example of the three selected regions in a given iteration of the outer loop for the graph in Fig. 1.

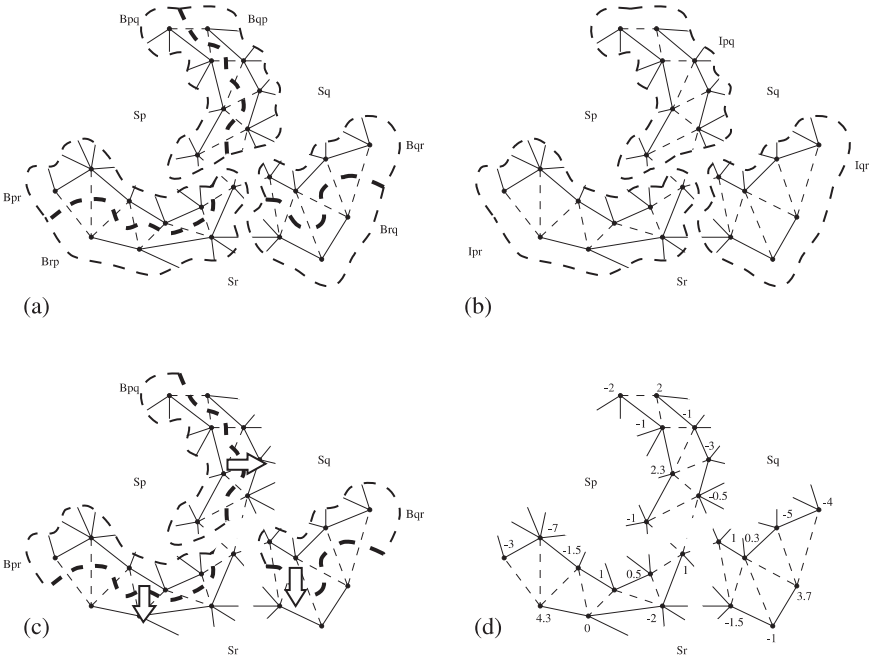

Fig. 4. An example graph showing: (a) subdomain faces; (b) interface regions for independent optimisation; (c) one of each pair of faces selected of alternating optimisation; (d) relative gains as a 'potential field'.

- *Relative gain optimisation*. If we think of the gain as a force or potential, then we can imagine a relative gain for every border vertex v according to the neighbouring vertices in the opposite face. Fig. 4(d) shows the relative gains of the border vertices for the graph in Fig. 1. Intuitively, if the gain of the opposite vertices is high, then 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. Once the relative gains are calculated, an 'appropriate' proportion of vertices are moved, highest relative gain first.

3.3.2. Global convergence

Although all three algorithms tend to converge robustly (especially the interface algorithm) with the global cost, $|E_c|$, decreasing monotonically, this cannot be guaranteed (unlike the serial algorithm [31]). To prevent cyclic 'thrashing', therefore, the outer loop is terminated either after no migration has taken place during an iteration or after some predetermined number of iterations if the cost has not decreased.

3.4. 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 Fig. 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 [31] 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 Section 3.4.1) 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, [13,17]; without it, the entire interface subgraph may be searched with diminishing prospect of finding a better solution along the search path.

3.4.1. Migration acceptance

Let T refer to the target subdomain weight for the graph (see Section 2.3). If the required flow from subdomain S_p to subdomain S_q is F_{pq}, then 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

$$2F_{pq} > |v| \tag{1a}$$

or

$$|S_q| + |v| \leqslant T \tag{1b}$$

These criteria are taken from our serial optimisation algorithm [31], 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 Section 3.4.3).

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 Section 3.4.3). For example, if $|S_p| = 23$, $|S_q| = 16$ and $F_{pq} = 3$, then if vertex v_1 with weight $|v_1| = 2$ is accepted for migration from S_p to S_q, then these values would be modified to $|S_p| = 21$, $|S_q| = 18$ and $F_{pq} = 1$. 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 $|S_p| = 19$, $|S_q| = 20$ and $F_{pq} = -1$ or alternatively $F_{qp} = 1$.

3.4.2. 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 (1a) and (1b), 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. Note that it is possible to find better partitions despite selecting some vertices with negative gain because, as the optimiser runs, the gains of adjacent vertices will change and so the migration of a group of vertices some or all of which start with negative gain can in fact decrease the overall cost (i.e., produce a net positive gain). 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 subdomain weight for the graph (see Section 2.3). Denoting $C(\pi^i)$, $F(\pi^i)$ and $W(\pi^i)$ by C^i, F^i and W^i, respectively (and similarly for $\overline{\pi}$) then π^i is confirmed as a new optimal partition if:

$$W^i \leqslant T \quad \text{and} \quad C^i < \overline{C} \tag{2a}$$

or

$$W^i \leqslant T \quad \text{and} \quad C^i = \overline{C} \quad \text{and} \quad W^i < \overline{W} \tag{2b}$$

or

$$T < W^i \quad \text{and} \quad F^i < \overline{F} \tag{2c}$$

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) and (2b) are more typical of KL type algorithms and confirm any partition which either improves on the optimal cost (2a) or on the optimal balance without raising the cost (2b).

3.4.3. Implementation issues

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 parallel graph coarsening and this imposes an additional communication cost.

- *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 Fig. 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 (1a) and (1b) and in determining $\overline{W}$ and W^i for (2a)–(2c), 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 although in practice this is almost always the case.

3.5. Alternating optimisation

The basis of the alternating optimisation approach, independently suggested by Karypis and Kumar, [21], and Walshaw et al. [34], 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 Fig. 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 in Section 3.4.

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 the conditions in (1a) and (1b) are satisfied. 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 all outgoing flow requirements have been satisfied and all vertices with non-negative gain have been visited. Finally note that the algorithm is restricted to migrating border vertices but in fact there is no reason why internal vertices should not be included once they join the border (as a result of neighbouring migrations); however, we have not tested this.

3.5.1. 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 Section 3.4.3). The problem of global consistency of data still arises as before though.

Our version of the alternating algorithm differs from that of Karypis and Kumar [21], 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 Section 2.3) enhances the algorithm (see also the results in Section 4.3). If, on the other hand, 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 (1a) and (1b) 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 Section 1.3). We believe this enhances the performance of the algorithm because it means that, during the course of the inner loop, vertices adjacent to those marked for migration (and thus more likely to be candidates for migration themselves) can have their gains adjusted. With virtual migration this may often not be possible, as two vertices in the same subdomain may not be owned by the same processor and such updating hence implies communication.

3.6. Relative gain optimisation

The third algorithm is a somewhat different approach which has already been described in [34] 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 v (calculated as the gain of v minus the average gain of neighbouring vertices in the opposite face) and use this as a mechanism to avoid collisions. Fig. 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 Fig. 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.

3.6.1. 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 $d = \max(g_{pq} - F_{pq} + g_{qp} - F_{qp}, 0)$, then 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.

3.6.2. 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

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

where $\mathrm{O}[\Gamma_q(v)]$ 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 Fig. 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 $2 - (-7/3) = 13/3$. 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. Results

The software tool written at Greenwich to implement the optimisation techniques is known as JOSTLE and is freely available for academic and research purposes

Table 1
Test meshes

Mesh	$\|V\|$	$\|E\|$	Mesh type
4elt	15 606	45 878	2D nodal graph
t60k-n	30 570	90 575	2D nodal graph
t60k-d	60 005	89 440	2D dual graph
dime20	224 843	336 024	2D dual graph
t60k-f	90 575	360 030	2D full graph
fe-rotor	99 617	662 431	3D nodal graph
598a	110 971	741 934	3D nodal graph
mesh100	103 081	200 976	3D dual graph
cyl3	232 362	457 853	3D dual graph
fe-ocean	143 437	409 593	3D semi-structured graph

under a licensing agreement. [1] Further details of the parallel implementation of the algorithms can be found in [32].

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 Section 4.4 for results on the impact of the initial distribution). The algorithm is allowed a 5% final imbalance tolerance (set at run-time); i.e., in the notation of Section 2.3, $\theta_0 = 1.05$.

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.

The results of the parallel multilevel partitioning using the interface optimisation algorithm from Section 3.4 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 an 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 [31]).

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

Table 2
The results of the interface algorithm showing the cut-weight C and parallel run-time in seconds t_s

Mesh	$P = 16$		$P = 32$		$P = 64$		$P = 128$	
	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-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	12 118	2.87	18 200	3.20
fe-rotor	22 789	8.36	36 345	7.20	50 580	6.58	70 933	8.04
598a	27 009	17.17	42 172	12.63	59 866	10.38	82 292	10.54
mesh100	4662	2.85	6795	2.41	9993	2.61	13 929	3.70
cyl3	9976	12.32	14 639	7.98	20 211	6.34	27 628	6.77
fe-ocean	8546	6.52	14 192	4.62	21 845	3.60	31 420	4.29

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

4.1. Comparison with alternating optimisation

Table 3 shows a comparison of the cut-weight C between alternating (A) and 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

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

Mesh	$P = 16$		$P = 32$		$P = 64$		$P = 128$	
	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-n	1997	1.14	3161	1.08	4742	1.08	7036	1.07
t60k-d	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-f	5846	1.13	8521	1.07	12 989	1.07	19 104	1.05
fe-rotor	24 823	1.09	36 571	1.01	52 188	1.03	72 802	1.03
598a	28 101	1.04	43 596	1.03	61 381	1.03	83 883	1.02
mesh100	4528	0.97	7274	1.07	10 634	1.06	14 633	1.05
cyl3	10 622	1.06	15 388	1.05	21 322	1.05	28 640	1.04
fe-ocean	10 269	1.20	15 792	1.11	25 008	1.14	34 119	1.09
Average		1.10		1.06		1.07		1.05

dramatic improvement for the interface optimisation, if, as is commonly assumed, the parallel communications overhead in the underlying solver are 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.

Space precludes a detailed comparison of timings (although one can be found in [32]), but alternating optimisation is almost always faster than interface optimisation with an overall average speed increase 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).

4.2. Comparison with relative gain optimisation

The cut-weight figures for relative gain optimisation are shown in Table 4 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%. However, 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 detailed timings but relative gain optimisation was on average 7% faster than interface optimisation.

4.2.1. A hybrid algorithm

During development work on these algorithms it was noticed that both alternating and 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

Table 4
A comparison of cut-weight results for relative gain (R) and interface (I) optimisation

Mesh	$P = 16$		$P = 32$		$P = 64$		$P = 128$	
	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-n	1872	1.07	3096	1.06	4768	1.09	7163	1.09
t60k-d	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-f	5445	1.05	8304	1.05	12 907	1.07	19 233	1.06
fe-rotor	23 655	1.04	38 460	1.06	52 951	1.05	75 898	1.07
598a	28 738	1.06	42 300	1.00	62 240	1.04	85 913	1.04
mesh100	4615	0.99	7459	1.10	11 422	1.14	15 637	1.12
cyl3	10 959	1.10	15 692	1.07	21 919	1.08	30 082	1.09
fe-ocean	10 653	1.25	16 849	1.19	25 445	1.16	36 391	1.16
Average		1.08		1.08		1.09		1.09

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 Section 3.3.2 were included. Interface optimisation, on the other hand, converges even more rapidly (although at greater cost per iteration).

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_H, the results are shown in Table 5 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 an 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.

4.3. Comparison with ParMETIS

We have also checked the results with another parallel partitioner ParMETIS [21]. As discussed in Sections 1.3 and 3.5.1, 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 6 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

Table 5
A comparison of cut-weight results for the hybrid relative gain/interface (H) and interface (I) optimisation

Mesh	$P = 16$		$P = 32$		$P = 64$		$P = 128$	
	C_H	C_H/C_I	C_H	C_H/C_I	C_H	C_H/C_I	C_H	C_H/C_I
4elt	1093	1.02	1774	1.06	2870	1.05	4450	1.03
t60k-n	1875	1.07	2887	0.99	4537	1.04	6786	1.03
t60k-d	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-f	5037	0.97	8090	1.02	12 271	1.01	18 580	1.02
fe-rotor	23 921	1.05	36 593	1.01	51 432	1.02	73 477	1.04
598a	27 810	1.03	41 187	0.98	60 080	1.00	83 294	1.01
mesh100	4456	0.96	7113	1.05	10 219	1.02	14 473	1.04
cyl3	10 137	1.02	14 456	0.99	20 381	1.01	27 557	1.00
fe-ocean	9688	1.13	15 477	1.09	23 951	1.10	33 688	1.07
Average		1.03		1.02		1.03		1.03

Table 6
A comparison of cut-weight results for ParMETIS and interface (I) optimisation

Mesh	$P = 16$		$P = 32$		$P = 64$		$P = 128$	
	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-n	1948	1.11	3109	1.06	4738	1.08	7124	1.08
t60k-d	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-f	5364	1.03	8657	1.09	13 353	1.10	19 960	1.10
fe-rotor	23 284	1.02	37 324	1.03	52 577	1.04	74 484	1.05
598a	30 440	1.13	44 059	1.04	63 460	1.06	86 841	1.06
mesh100	5133	1.10	7745	1.14	11 067	1.11	15 261	1.10
cyl3	115 42	1.16	159 64	1.09	22 168	1.10	29 677	1.07
fe-ocean	12 692	1.49	20 252	1.43	27 580	1.26	37 843	1.20
Average		1.13		1.11		1.09		1.09

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 (see Section 3.5.1).

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. Space considerations preclude a detailed comparison of timings in tabular form (although one can be found in [32]) but ParMETIS is always faster than JOSTLE, on average taking 33% of the time to partition (i.e., in other words it is about three times faster on average).

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 s which in our experience is insignificant compared with the runtime of a typical parallel unstructured mesh application. Note also that these are times for the block-based initial distribution which can significantly slow down the partitioning (see below Section 4.4); in a dynamic mesh partitioning scenario (e.g., repeated mesh refinement), where the partitioning time tends to be much more significant, the initial partition is likely to be of much higher quality and hence the times even faster (again see Section 4.4 and, for examples of dynamic repartitioning [34]).

The average imbalance for ParMETIS was 1.032 and occasionally it exceeded the allowed imbalance of 1.05 (dime20, $P = 64$, imbalance $\theta = 1.097$; 4elt, $P = 128$, $\theta = 1.066$; fe-rotor, $P = 128$, $\theta = 1.053$). 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

[2] By setting `UNBALANCE_FRACTION` & `ORDER_UNBALANCE_FRACTION` in `defs.h` to 1.01.

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 of 1.030.

4.4. The impact of the initial distribution

It is of interest to ask what impact does the initial distribution have on the outcome of the final partition. In Table 7, we compare four different initial distribution schemes for 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 of Farhat's algorithm [10]. 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. Note that this

Table 7
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

Initial	$P = 16$			$P = 32$			$P = 64$		
	C_0	C	t	C_0	C	t	C_0	C	t
t60k-n									
Cyclic	86 929	1821	2.08	88 210	2863	1.51	89 586	4476	1.45
Random	84 843	1737	2.06	87 722	2863	1.55	89 103	4385	1.43
Block	6639	1753	0.88	7998	2930	0.80	10 536	4378	0.80
Greedy	2248	1758	0.44	3511	2908	0.48	5336	4476	0.61
Cyl3									
Cyclic	43 2639	10 299	14.71	445 449	14 796	9.07	451 608	20 564	6.86
Random	429 109	10 195	14.87	443 501	14 508	9.33	450 678	20 606	6.83
Block	351 188	9976	12.31	375 349	14 639	7.91	388 139	20 211	6.34
Greedy	20 014	10 398	3.49	27 858	14 984	2.93	37 442	20 911	3.47

discrepancy in runtime diminishes as P increases and we believe that the main reason for this is that the initial qualities, C_0, for the greedy algorithm and block distribution increase with P whilst for the cyclic and random distributions, which result in almost all edges being cut (93.7–99.9% in Table 7), C_0 stays relatively constant.

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 7) 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.

5. 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, Section 3.4, the most robust of the three and the one closest to our serial flow and tolerance algorithm [31] (and indeed the original Kernighan–Lin algorithm [22]) 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 [23]. We have also made comparisons with another partitioning tool, ParMETIS, and shown that our results are of higher quality although taking longer to compute. In Section 4.4, we demonstrate the global quality of the results and that the initial distribution strongly affects partitioning time.

Much work continues in the field of mesh partitioning, for example to optimise different cost functions, e.g., [33], 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 parallel algorithms to minimise alternative objective functions such as subdomain aspect ratio or machine mapping (rather than just cut-weight).

References

[1] S.T. Barnard, PMRSB: Parallel multilevel recursive spectral bisection, Cray Res. Inc., 1996.
[2] S.T. Barnard, H.D. Simon, A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems, Concurrency: Practice Experience 6 (2) (1994) 101–117.

[3] R. Biswas, L. Oliker, Experiments with repartitioning and load balancing adaptive meshes, Technical Report NAS-97-021, NASA Ames, Moffat Field, CA, 1997.
[4] P. Buch, J. Sanghavi, A. Sangiovanni-Vincentelli, A parallel graph partitioner on a distributed memory multiprocessor, in: Proceedings of the Fifth IEEE Symposium on Frontiers of Massively Parallel Computation, IEEE, 1995, pp. 360–366.
[5] T.N. Bui, C. Jones, A heuristic for reducing fill-in in sparse matrix factorization, in: R.F. Sincovec, et al. (Ed.), Parallel Processing for Scientific Computing, SIAM, Philadelphia PA, 1993, pp. 445–452.
[6] G. Cybenko, Dynamic load balancing for distributed memory multiprocessors, J. Parallel Distrib. Comput. 7 (2) (1989) 279–301.
[7] R. Diekmann, A. Frommer, B. Monien, Efficient schemes for nearest neighbor load balancing, Parallel Computing 25 (7) (1999) 789–812.
[8] R. Diekmann, B. Meyer, B. Monien, Parallel decomposition of unstructured FEM-meshes, Concurrency: Practice Experience 10 (1) (1998) 53–72.
[9] P. Diniz, S. Plimpton, B. Hendrickson, R. Leland, Parallel algorithms for dynamically partitioning unstructured grids, in: D. Bailey et al. (Ed.), Parallel Processing for Scientific Computing, SIAM, Philadelphia PA, 1995, pp. 615–620.
[10] C. Farhat, A simple and efficient automatic FEM domain decomposer, Comput. Struct. 28 (5) (1988) 579–602.
[11] C.M. Fiduccia, R.M. Mattheyses, A linear time heuristic for improving network partitions, in: Proceedings of the 19th IEEE Design Automation Conference, IEEE, Piscataway, NJ, 1982, pp. 175–181.
[12] B. Ghosh, S. Muthukrishnan, M.H. Schultz, Faster schedules for diffusive load balancing via over-relaxation, Technical Report 1065, Department of Computer Science, Yale University, New Haven, CT 06520, USA, 1995.
[13] J.R. Gilbert, E. Zmijewski, A parallel graph partitioning algorithm for a message-passing multiprocessor, Int. J. Parallel Prog. 16 (6) (1987) 427–449.
[14] A. Gupta, Fast and effective algorithms for graph partitioning and sparse matrix reordering, IBM J. Res. Dev. 41 (1/2) (1996) 171–183.
[15] B. Hendrickson, Graph partitioning and parallel solvers: Has the emperor no clothes? in: A. Ferreira, J. Rolim (Eds.), Proceedings Irregular '98: Parallel Algorithms for Irregularly Structured Problems, vol. 1457 of LNCS, Springer, 1998, pp. 218–225.
[16] B. Hendrickson, R. Leland, A multilevel algorithm for partitioning graphs, Technical Report SAND 93-1301, Sandia National Labs, Albuquerque, NM, 1993.
[17] B. Hendrickson, R. Leland, A multilevel algorithm for partitioning graphs, in: S. Karin (Ed.), Proceedings of Supercomputing'95, San Diego, CA, ACM Press, New York, 1995.
[18] Y.F. Hu, R.J. Blake, The optimal property of polynomial based diffusion-like algorithms in dynamic load balancing, in: K.D. Papailiou et al. (Ed.), Computational Dynamics '98, Wiley, New York, 1998, pp. 177–183.
[19] Y.F. Hu, R.J. Blake, D.R. Emerson, An optimal migration algorithm for dynamic load balancing, Concurrency: Practice Experience 10 (6) (1998) 467–483.
[20] G. Karypis, V. Kumar, Multilevel k-way partitioning scheme for irregular graphs, J. Par. Dist. Comput. 48 (1) (1998) 96–129.
[21] G. Karypis, V. Kumar, A coarse-grain parallel formulation of multilevel k-way graph partitioning algorithm, in: M. Heath (Ed.), Parallel Processing for Scientific Computing, SIAM, Philadelphia, PA, 1997.
[22] B.W. Kernighan, S. Lin, An efficient heuristic for partitioning graphs, Bell Sys. Tech. J. 49 (1970) 291–308.
[23] D.E. Keyes, D.K. Kaushik, B.F. Smith, Prospects for CFD on Petaflops Systems, in: M. Hafez, K. Oshima (Ed.), CFD Review 1998, World Scientific, Singapore, 1998, pp. 1079–1096.
[24] R. Lohner, R. Ramamurti, D. Martin, A parallelizable load balancing algorithm, AIAA-93-0061, American Institute of Aeronautics and Astronautics, Washington, DC, 1993.
[25] J. Savage, M. Wloka, Parallelism in graph partitioning, J. Parallel Distrib. Comput. 13 (1991) 257–272.

[26] K. Schloegel, G. Karypis, V. Kumar, Multilevel diffusion schemes for repartitioning of adaptive meshes, J. Parallel Distrib. Comput. 47 (2) (1997) 109–124.
[27] N.G. Shivaratri, P. Krueger, M. Singhal, Load distributing for locally distributed systems, IEEE Comput. 25 (1/2) (1992) 33–44.
[28] H.D. Simon, A. Sohn, R. Biswas, HARP: A dynamic spectral partitioner, J. Parallel Distrib. Comput. 50 (1/2) (1998) 83–103.
[29] J. Song, A partially asynchronous and iterative algorithm for distributed load balancing, Parallel Comput. 20 (6) (1994) 853–868.
[30] D. Vanderstraeten, R. Keunings, Optimized partitioning of unstructured computational grids, Int. J. Numer. Meth. Engrg. 38 (1995) 433–450.
[31] C. Walshaw, M. Cross, Mesh partitioning: a multilevel balancing and refinement algorithm, SIAM J. Sci. Comput. 22 (1) (2000) 63–80.
[32] C. Walshaw, M. Cross, Parallel optimisation algorithms for multilevel mesh partitioning, Technical Report 99/IM/44, University of Greenwich, London SE18 6PF, UK, February 1999.
[33] C. Walshaw, M. Cross, R. Diekmann, F. Schlimbach, Multilevel Mesh Partitioning for Optimising Domain Shape, Int. J. High Performance Comput. Appl. 13 (4) (1999) 334–353.
[34] C. Walshaw, M. Cross, M. Everett, Parallel dynamic graph partitioning for adaptive unstructured meshes, J. Parallel Distrib. Comput. 47 (2) (1997) 102–108.