Mesh Partitioning: a Multilevel Balancing and Refinement Algorithm

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 98/IM/35

March 27, 1998

Abstract

Multilevel algorithms are a successful class of optimisation techniques which address the mesh partitioning problem. They usually combine a graph contraction algorithm together with a local optimisation method which refines the partition at each graph level. In this paper we present an enhancement of the technique which uses imbalance to achieve higher quality partitions. We also present a formulation of the Kernighan-Lin partition optimisation algorithm which incorporates load-balancing. The resulting algorithm is tested against a different but related state-of-the-art partitioner and shown to provide improved results.

Keywords: graph-partitioning, mesh partitioning, load-balancing, multilevel algorithms.

1 Introduction

The need for mesh partitioning arises naturally in many finite element (FE) and finite volume (FV) applications. Meshes composed of elements such as triangles or tetrahedra are often better suited than regularly structured grids for representing completely general geometries and resolving wide variations in behaviour via variable mesh densities. Meanwhile, the modelling of complex behaviour patterns means that the problems are often too large to fit onto serial computers, either because of memory limitations or computational demands, or both. 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. It is well known that this problem is NP-complete, e.g. [5], so in recent years much attention has been focused on developing suitable heuristics, and some powerful methods, many based on a graph corresponding to the communication requirements of the mesh, have been devised, e.g. [3].

A particularly popular and successful class of algorithms which address this mesh 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 an enhancement of the technique which uses imbalance to achieve higher quality partitions. We also present a formulation of the Kernighan-Lin partition optimisation algorithm which incorporates load-balancing.

We focus here on serial partitioning. Although emphasis in the field is switching to parallel partitioning methods, we aim here to address one of the fundamental mechanisms of the multilevel paradigm and thus a serial implementation provides a clear and relatively parameter-free environment for establishing how imbalance can affect the overall performance of the strategy. In addition the algorithms described here are used directly as part of a parallel partitioner in [20].

1.1 Overview

Below, in Section 1.2, we introduce the mesh partitioning problem and establish some terminology. In Section 2 we then describe the multilevel paradigm and present a new enhancement in the idea of a multilevel balancing schedule. Related work in the area is discussed in §2.2 and a graph contraction algorithm is outlined. In Section 3 we then describe a Kernighan-Lin (KL) type optimisation algorithm which both balances a partition of the graph to within

some given tolerance and also refines it. In Section 4 we present results from the multilevel balancing and refinement algorithm comparing it with a similar formulation which only incorporates multilevel refinement. We also compare different multilevel balancing schedules. Finally in Section 5 we draw some conclusions and present some ideas for further investigation.

The principal innovations described in this paper are twofold:

- in §2.2 we formalise the idea of combining multilevel refinement with a multilevel balancing schedule.
- in §3.4 we describe a new formulation of a KL type partitioning algorithm (incorporating hill-climbing) which both balances and refines.

Two implementation ideas are also described:

- in §3.3 we describe a ranking for prioritising vertices for migration which incorporates their weight as well as their gain.
- in §3.5 we describe how we deal with vertices which are neighbours to more than one subdomain.

1.2 Notation and Definitions

To define the mesh partitioning problem, let $G = G(V, E)$ be an undirected graph of vertices V, with edges E which represent 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 $G_\pi = G_\pi(S, C)$; there is an edge (S_p, S_q) in C 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 $|E_c| = |C|$). Vertices which have an edge in E_c (i.e. $\{v \in V :$ there exists $v' \in V,$ with $(v, v') \in E_c\}$) are referred to as *border* vertices.

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, although see §3.1 for further discussion on this point. A more precise definition of the graph-partitioning problem is therefore to find π such that $S_p \leq \overline{S}$ and such that $|E_c|$ is minimised. Note that perfect balance is not always possible for graphs with non-unitary vertex weights.

Throughout the paper we use some fairly specific terminology and in particular we shall refer to *refinement* as the improvement of partition quality (i.e. the cut-edge weight) without regard to load-balance; *balancing* will then refer to the improvement of imbalance and *optimisation* refers to refinement and balancing. We also make the distinction between those algorithms, such as that of Kernighan & Lin [13], which refine a *bisection* and algorithms which refine a partition of P subdomains. Such algorithms have been known as k-way (e.g. [12]) or multiway (e.g. [19]) algorithms but here we shall simply refer to them as *partition* (as opposed to bisection) refinement algorithms. Finally we shall use the words processor and subdomain interchangeably; the mesh is partitioned into P subdomains each of which will be mapped onto one processor.

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, recursively iterate this procedure until the graph size falls below some threshold and then successively refine these reduced size graphs. This sequence of contraction followed by repeated expansion/refinement loops is known as the multilevel paradigm and has been successfully developed as a strategy for overcoming the localised nature of the KL (and other) algorithms. The multilevel idea was first proposed by Barnard & Simon, [1], as a method of speeding up spectral bisection and improved by Hendrickson & Leland, [8] who generalised it to encompass local refinement algorithms.

2.1 Implementation

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, [8]. 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 is to visit the vertices of the graph in a random order and pair up or match unmatched vertices with an unmatched neighbour. It has been shown, [11], that it can be beneficial to the optimisation to collapse the most heavily weighted edges and our matching algorithm uses this heuristic.

The initial partition. Having constructed the series of graphs until the number of vertices in the coarsest graph is smaller than some threshold, the normal practice of the multilevel strategy is to carry out an initial partition. Here, following the idea of Gupta, [7], we contract until the number of vertices in the coarsest graph is the same as the number of subdomains, P, and then simply assign vertex i to subdomain S_i. Unlike Gupta, however, we do not carry out repeated expansion/contraction cycles of the coarsest graphs to find a well balanced initial partition but instead, since our optimisation algorithm incorporates balancing, we commence on the expansion/optimisation sequence immediately.

Partition expansion. Having optimised the partition on 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 $v \in V_l$ is in subdomain S_p then the matched pair of vertices that it represents, $v_1, v_2 \in V_{l-1}$, will be in S_p.

2.2 Multilevel balancing schedule

It has been noted previously (e.g. [12, 17]) 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 expansion 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). In this paper 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 expansion 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 expansion 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 (see the end of this section for an example).

In fact it is often not possible to achieve perfect balance in the coarsest graphs because the vertices may be heavily weighted and very inhomogeneous (e.g. if balance requires moving a weight of 10 from one subdomain to another but all vertices are of weight 20 or over, perfect balance cannot be attained). Hence it could be argued that all multilevel algorithms employ this idea of multilevel balancing. Indeed, our previous work in this area, e.g. [19], employs a diffusive load-balancer *at every* refinement level and so the idea has been implicit in our work for sometime. In this paper, however, we formalise the idea of balancing and refinement at every level and also describe an optimisation algorithm which both achieves a given level of imbalance (if possible) and refines for quality. Note that, we make the distinction between this work and that in the multilevel diffusion algorithm of Schloegel *et al.*, [14] where a diffusive load-balancer is employed at each coarse level until balance is attained and thereafter partition quality refinement without active balancing is employed.

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 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 (see the end of this section for an example).

- if it ascends too slowly, the benefits for the partition quality of having a high imbalance tolerance may never be seen.

Some results with different functions for T_l are given in §4.2, but with the above in mind we derive T_l as follows:

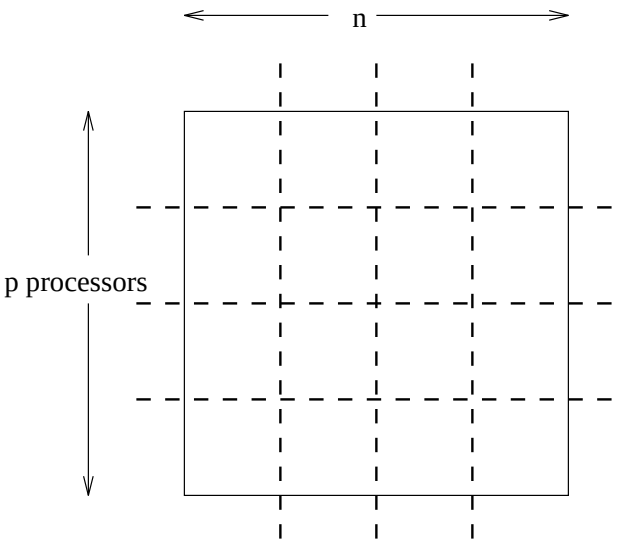

Figure 1: A regular domain of $N = n^2$ vertices perfectly partitioned into $P = p^2$ subdomains

Let $G(V, E)$ be regular graph with N ($= n \times n$) vertices perfectly partitioned into P ($= p \times p$) subdomains as in Figure 1. The border length of a subdomain is then given by

$$4\left(\frac{N}{P}\right)^{\frac{1}{2}}$$

The average weight of a vertex is $|V|/N$ and so we can estimate the weight of border vertices in the subdomain as

$$4\left(\frac{N}{P}\right)^{\frac{1}{2}} \times \frac{|V|}{N} = \frac{4|V|}{(PN)^{\frac{1}{2}}}$$

Recall that for each graph G_l we wish to define a target subdomain weight T_l which will not cause too much degradation in partition quality when balancing its parent G_{l-1} down to its target T_{l-1}. After some experimentation, we have chosen to allow an excess weight in any given subdomain of approximately half one border layer of a subdomain in the parent graph. The target weight is given by the optimal subdomain weight plus the excess weight and so, using the regular 2-dimensional (2D) model, we set T_l to be

$$T_l = \lceil |V|/P \rceil + \frac{1}{2} \times \frac{4|V|}{(PN_{l-1})^{\frac{1}{2}}}.$$

If we define the imbalance tolerance, θ_l, to be the maximum allowable subdomain weight expressed as a proportion of the optimal subdomain weight, then

$$\theta_l = \frac{\lceil |V|/P \rceil + 2|V|(PN_{l-1})^{-\frac{1}{2}}}{\lceil |V|/P \rceil} \approx 1 + 2\left(\frac{P}{N_{l-1}}\right)^{\frac{1}{2}}$$

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. [12], allow a slight imbalance. For the results in this paper we have chosen to set $\theta_0 = 1.03$ and then we set $\theta_l = \max(\theta_0, 1 + 2(\frac{P}{N_{l-1}})^{\frac{1}{2}})$ for $l > 0$. Note that we have chosen a 2D model of the regular partition; a 3D model using the same arguments gives $\theta_l = 1 + 3\left(\frac{P}{N_{l-1}}\right)^{\frac{1}{3}}$ and results using this model can be found in §4.2.

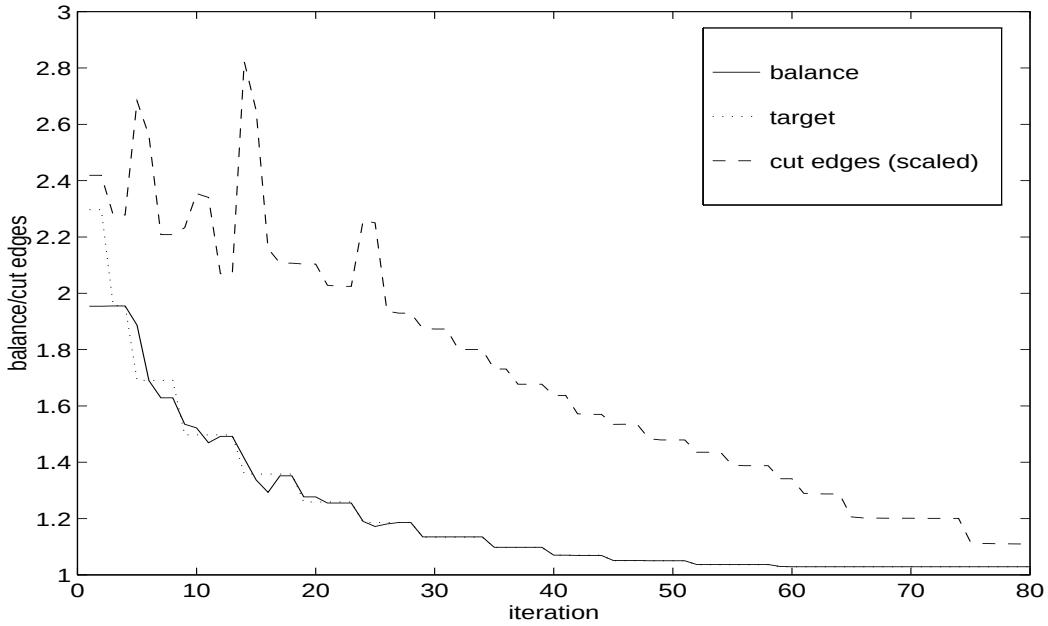

Figure 2: An example of the evolution of balance and cut-edge weight for the multilevel balancing and refinement algorithm

Figure 2 shows an example of some typical behaviour for the balancing schedule derived above and the algorithm described in Section 3. The dotted line plots the target weight or balancing schedule and each step down represents the transition from one graph level G_l to its parent G_{l-1}. Notice that at the start of the iterations the tolerance is around 2.3 – i.e. the graph is considered balanced if every subdomain is smaller than 2.3 times the optimal weight. The solid line represents the attained balance – this is below the target level most of the time and, by iteration 30, it tracks the target weight exactly, showing that the optimisation algorithm in Section 3 is very good at taking advantage of any leeway in the imbalance tolerance (the final imbalance tolerance for the method is set at $\theta_0 = 1.03$ which is why the balance never reaches 1.0). Finally the dashed line shows the evolution of the cut edges (scaled by a large factor to fit onto the graph). The peaks early on in the iterations correspond to balances which exceed the tolerance and as mentioned above this causes serious degradation in the partition quality as the algorithm balances the graph. However, after about iteration 30 the cut-edge weight decreases monotonically.

3 The balancing and refinement optimisation algorithm

In this section we describe an optimisation algorithm which combines load-balancing and partition quality refinement. It is a Kernighan-Lin (KL) type algorithm incorporating a hill-climbing mechanism to enable it to escape from local minima; in other words vertex migration from subdomain to subdomain can be *accepted* even if it degrades the partition quality and later, based on the subsequent evolution of the partition, either rejected or *confirmed*. The algorithm uses bucket sorting (see §3.3), the linear time complexity improvement of Fiduccia & Mattheyses, [4], and is a partition optimisation formulation; in other words it optimises a partition of P subdomains rather than a bisection. In this respect it most closely resembles the algorithm of Karypis & Kumar, [12], but additionally incorporates load-balancing (using the diffusive algorithm of Hu & Blake, [10], – see §3.2). This is similar to our previous work, [18], which described a KL type algorithm combining balancing and refinement but in a parallel formulation (essentially each inter-subdomain interface is treated as a separate problem). The algorithm described here is a strictly serial in nature, but an updated version of the parallel formulation can be found in [20].

3.1 The gain function

A key concept in the method is the idea of *gain*. Loosely, the gain $g(v, q)$ 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 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, $|E_c|$, 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 [16], Vanderstraeten *et al.* demonstrate that it can be extremely effective

to vary the cost function based on a knowledge of the solver. Whichever cost function is chosen, however, the idea of gains is generic. For the purposes of this paper we shall assume that the gain $g(v,q)$ just expresses the reduction in the cut-edge weight, $|E_c|$.

3.2 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, 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 [15] and some common strategies described. Much work has been carried out on parallel or distributed algorithms and, in particular, on diffusive algorithms, e.g. [2, 6], but here we use an elegant technique developed by Hu & Blake, [10], which converges faster than diffusive methods and minimises the Euclidean norm of the transferred weight. 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 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.

This algorithm (or, in principle, any other distributed load-balancing algorithm) is used to determine *how much* weight to transfer across edges of the subdomain graph and the optimisation technique below is then used to decide *which* vertices to move. The algorithm is employed as suggested in [10], solving iteratively with a conjugate gradient solver.

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. We also limit the number of possible rebalances on any graph since otherwise it is possible to get cyclic behaviour.

3.3 Bucket sorting

The bucket sort is an essential tool for the efficient and rapid sorting and adjustment of vertices by their gain. The concept was first suggested by Fiduccia & Mattheyses in [4] 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 $g(v)$ the gain of a vertex v, we rank v with the formula:

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

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

$$\text{rank}(v) \leq g(v) \times W + W - 1 < g(v) \times W + W = [g(v) + 1] \times W.$$

Hence

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

In other words, for a given vertex v with gain $g(v)$, the upper bound on rank(v) is strictly less than the lower bound on rank(w) for any vertex w with gain $g(w) = [g(v) + 1]$.

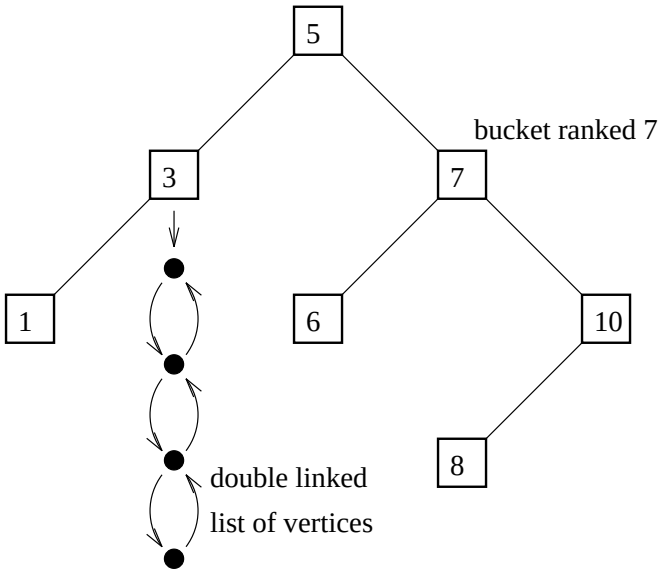

Figure 3: A bucket tree

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 (see Figure 3). 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 The iterative optimisation algorithm

The serial optimisation algorithm, as is typical for KL type algorithms, has inner and outer iterative loops with the outer loop terminating when no migration takes place during an inner loop. The algorithm is shown in pseudo-code form in Figure 4. The optimisation uses two bucket trees (see §3.3) and is initialised by calculating the gain for all border vertices and inserting them into one of the bucket trees. These vertices will subsequently be referred to as *candidate* vertices and the tree containing them as the *candidate tree*. The idea of only inserting the border vertices into the bucket tree (rather than all vertices) was first described in [17] and has subsequently been described as lazy initialisation, [9].

The inner loop proceeds by examining candidate vertices highest gain first (by always picking vertices from the highest ranked bucket), testing whether the vertex is acceptable for migration and then transferring it to the other bucket tree (the tree of *examined* vertices). This inner loop terminates when the candidate tree is empty although it may terminate early if the partition cost (i.e. the number of cut edges) rises too far above the cost of the best partition found so far. This type of early termination is typical of KL type algorithms; without it, the entire graph may be searched with diminishing prospect of finding a better solution along the search path. Once the inner loop has terminated any vertices remaining in the candidate tree are transferred to the examined tree and finally pointers to the two trees are swapped ready for the next pass through the inner loop.

Migration acceptance. Let T refer to the target weight for the graph (§2.2) and W represent the weight of the largest subdomain, $W = \max_P |S_p|$. If the required flow from subdomain S_p to subdomain S_q is F_{pq}, a candidate

```
while (optimising) { /* outer loop */
    optimising = 0
    best cost = cost
    while (vertices in candidate tree) { /* inner loop */
        vertex = best candidate
        if (migration acceptable for vertex) {
            optimising = 1
            migrate vertex {
                adjust flow & subdomain weights
                adjust gains of neighbouring vertices
                    and transfer to appropriate buckets
            }
            adjust gain of vertex and transfer to examined tree
            if (better partition) { /* confirm migration */
                best_cost = cost
                reset recent move list
            } else {
                append vertex to recent move list
                if (cost - best_cost > limit) /* early termination */
                    break
            } /* hill-climbing mechanism */
        }
        transfer vertex to examined tree
    } /* inner loop */
    for (vertices in recent move list) /* hill-climbing mechanism */
        migrate vertex back to previous partition
    for (vertices in candidate tree)
        transfer vertex to examined tree
    swap pointers to candidate and examined trees
} /* outer loop */
```

Figure 4: The Kernighan-Lin partition optimisation algorithm

vertex v with weight $|v|$ (> 0) is accepted for migration from S_p to S_q (with weights $|S_p|$ and $|S_q|$) if

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

These criteria reflect the aim of trying to balance the graph down to the target weight, T, and then keeping it there. If the graph is not yet within the imbalance tolerance (i.e. $W > T$) then (1a) only allows migration which reduces the required flow. Condition (1b) guarantees that once balance is achieved the graph cannot become unbalanced again.

Note that in order to satisfy the flow entirely we would only move a vertex if from S_p to S_q if the flow, F_{pq}, was greater than or equal to the vertex's weight (i.e. $F_{pq} \geq |v|$). The wider acceptance condition (1a), $2F_{pq} > |v|$, however, also allows moves where $|v|$ exceeds F_{pq} but which reduce the total required flow in the system. For example, if $|v| = 5$ and $F_{pq} = 3$ the migration would not be acceptable under the condition $F_{pq} \geq |v|$, but using (1a) the move is acceptable and F_{pq} changes to -2 (alternatively $F_{qp} = 2$) after migration, which is a reduction in the total.

When a vertex is accepted for migration, its partition is changed and the subdomain weights and flow are adjusted. The gains are recalculated for the vertex and all of its neighbours and they are transferred to the appropriately ranked buckets. Note that examined vertices are transferred between buckets in the examined bucket tree and candidate vertices are transferred between buckets in the candidate bucket tree. Neighbouring vertices which were not in the border at this point but which become border vertices as a result of the migration are put into the candidate tree. In this way it is actually possible for a vertex to be migrated more than once during the course of an inner loop (if it is moved out of and back into the border region by migration it becomes a candidate vertex at each stage) but vertices are never directly transferred from the examined tree to the candidate tree as this can lead to infinitely cyclic behaviour.

Migration confirmation and hill-climbing. The algorithm uses a KL type hill-climbing strategy. As can be seen from (1) migrations can be *accepted* even if they increase the partition cost (i.e. have negative gain). During each pass through the inner loop, a record of the optimal partition achieved by migration within that loop is maintained

together with a list of vertices which have migrated since that value was attained. If subsequent migration finds a 'better' partition then the migration is *confirmed* and the list is reset. Once the inner loop is terminated, any vertices remaining in the list (vertices whose migration has not been confirmed) are migrated back to the subdomains they came from when the optimal cost was attained.

To define a 'better' partition, let $\overline{\pi}$ represent the optimal partition reached so far and π^i the subsequent partition after some migration (i.e. after some iterations of the inner loop). Each partition has a cost associated with it, $C(\pi)$, (in this case just the total weight of cut edges) and an imbalance which depends on $W(\pi)$, the weight of the largest subdomain in that partition. Again let T represent the target weight for the graph (see §2.2). Denoting $C(\pi^i)$ and $W(\pi^i)$ by C^i and W^i (and similarly for $\overline{\pi}$) then π^i is confirmed as a new optimal partition if:

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

Condition (2c) simply states that, while the graph is unbalanced (i.e. $W^i > T$), any partition which improves the balance 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 Vertices adjacent to several subdomains

In general, for graphs arising from FE/FV meshes with coarse granularity partitions (i.e. $N \gg P$), most border vertices will only be adjacent to vertices in one other subdomain. However, those vertices that are adjacent to several subdomains are treated slightly differently in that, if a tested migration is not acceptable, they are replaced in the *candidate* tree at the level of their next highest gain. They are not transferred to the examined tree until either being successfully migrated or all possible migrations have been tested. This is best illustrated with an example: suppose a vertex is adjacent to 4 subdomains, S_p, S_q, S_r and S_s, and suppose that the respective gains are $g_p > g_q = g_r > g_s$. The vertex is initially placed in the candidate tree and ranked g_p. When subsequently tested, if migration is not acceptable (using the criteria in 1), the vertex is replaced in the candidate tree and ranked g_q $(= g_r)$. When the vertex next comes up for testing, migration to S_p, S_q and S_r is assessed (note that a move to S_p may now be acceptable due to the intervening migration) and if none are acceptable the vertex is replaced in the candidate tree with a rank g_s. When the vertex is again tested, migration to S_p, S_q, S_r and S_s is tested and if none are acceptable, the vertex is transferred to the examined tree ranked g_p. Of course, it might be considered unnecessary to retest moves which have already been tested (i.e. those with gains greater than the migration under consideration), but since the edge weights to all adjacent subdomains must be calculated to determine the next highest gain, there is no great extra expense involved in doing so.

4 Results

mesh	$\lvert V\rvert$	$\lvert E\rvert$	
crack	10240	30380	2D nodal graph
4elt	15606	45878	2D nodal graph
t60k	60005	89440	2D dual graph
dime20	224843	336024	2D dual graph
144	144649	1074393	3D nodal graph
m14b	214765	1679018	3D nodal graph
fe-ocean	143437	409593	3D dual graph
mesh1m	1119663	2212012	3D dual graph

Table 1: Test meshes

We have implemented the algorithms described here within the framework of JOSTLE, a mesh partitioning software tool developed at the University of Greenwich and freely available for academic and research purposes under a licensing agreement[1]. The experiments were carried out on a Sun SPARC 20 with a 50 MHz CPU and 320 Mbytes of memory. The test graphs 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.

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

	$P=16$		$P=32$		$P=64$		$P=128$	
mesh	$\|E_c\|$	t_s	$\|E_c\|$	t_s	$\|E_c\|$	t_s	$\|E_c\|$	t_s
crack	1173	0.73	1789	0.95	2689	1.38	3986	2.68
4elt	1012	0.92	1687	1.18	2772	2.32	4258	3.11
t60k	984	2.62	1588	3.20	2447	4.43	3585	6.54
dime20	1274	9.41	2282	10.82	3617	13.43	5517	19.74
144	41842	24.17	61145	32.17	83710	39.06	113376	62.43
m14b	45988	31.48	73238	39.92	105739	52.81	148561	77.36
fe-ocean	8879	12.65	14302	18.32	23098	26.03	31945	36.27
mesh1m	24522	84.65	35178	98.97	52418	132.63	72543	174.26

Table 2: The results of the multilevel balancing and refinement algorithm showing the cut-edge weight $|E_c|$ and CPU time in seconds t_s

The results of using the multilevel balancing and refinement algorithm are shown in Table 2 for four values of P (the number of processors/subdomains). The table shows the total weight of cut edges, $|E_c|$, and the run time in seconds, t_s. The algorithm is allowed a final imbalance tolerance of $\theta_0 = 1.03$ (although this may be reset at runtime). In the following sections we compare the results with different balancing schedules and with a similar multilevel mesh partitioner which does not use a multilevel balancing schedule. In these sections the $|E_c|$ results in Table 2 are also referred to as $|E_c(J)|$ and $|E_c(T_2)|$.

4.1 Comparison results

To demonstrate the quality of the partitions, we have compared the results in Table 2 with those produced by METIS, another state-of-the-art partitioning package. The version we have used, `kmetis 2.0.6`, provides multilevel coarsening with a heavy edge heuristic and we have used the option of a Kernighan-Lin partition refinement algorithm (the default is a greedy partition refinement algorithm which is slightly faster but provides slightly lower quality partitions). The primary distinctions between the two partitioners, aside from implementation details, is that METIS coarsens to 2000 vertices and then carries out a balanced initial partition, whilst JOSTLE coarsens to P vertices (one per subdomain) and then uses the multilevel balancing schedule described in §2.2 and the balancing refinement algorithm described in §3.4. A more recent version of METIS is now available, but only allows greedy refinement.

	$P=16$		$P=32$		$P=64$		$P=128$	
mesh	$\|E_c(M)\|$	$\frac{\|E_c(M)\|}{\|E_c(J)\|}$	$\|E_c(M)\|$	$\frac{\|E_c(M)\|}{\|E_c(J)\|}$	$\|E_c(M)\|$	$\frac{\|E_c(M)\|}{\|E_c(J)\|}$	$\|E_c(M)\|$	$\frac{\|E_c(M)\|}{\|E_c(J)\|}$
crack	1309	1.12	1959	1.10	2834	1.05	4318	1.08
4elt	1188	1.17	1831	1.09	2942	1.06	4637	1.09
t60k	1031	1.05	1648	1.04	2614	1.07	3702	1.03
dime20	1339	1.05	2328	1.02	3742	1.03	5711	1.04
144	42219	1.01	62482	1.02	87045	1.04	118079	1.04
m14b	49744	1.08	75743	1.03	108221	1.02	153154	1.03
fe-ocean	10108	1.14	16215	1.13	24786	1.07	34974	1.09
mesh1m	24000	0.98	36706	1.04	54015	1.03	75463	1.04
Average		1.07		1.06		1.05		1.06

Table 3: A comparison of cut edge results for METIS, $|E_c(M)|$, and JOSTLE, $|E_c(J)|$

Table 3 shows a comparison of the cut-edge weight $|E_c|$. For each value of P, the first column shows the value of $|E_c|$ for METIS, $|E_c(M)|$, while the second column shows the ratio of $|E_c|$ for METIS over that for JOSTLE, $|E_c(M)|/|E_c(J)|$. As can be seen, with one exception (mesh1m, $P = 16$), the results for METIS are always worse and can be 17% larger (4elt, $P = 16$). The average difference in the quality ranges between 5% and 7% over the different values of P. Although this does not demonstrate a dramatic improvement for our algorithm, it does indicate a consistent improvement on results perceived as state-of-the-art.

It is not the primary aim of this paper to compare run times for the algorithms, but Table 4 shows a similar comparison of t_s. In fact this table highlights a difference in implementations more than anything. For both codes the run time is approximately linearly dependent on the number of border vertices (which increase with P), however, JOSTLE loops over border vertices while METIS loops over all vertices in the graph. This means that for coarse granularities (larger meshes or smaller values of P), where a relatively small number of vertices are in the subdomain borders, JOSTLE is faster since it visits a much smaller proportion of the data. However, for finer granularities (smaller

mesh	$P=16$		$P=32$		$P=64$		$P=128$	
	$t_s(M)$	$\frac{t_s(M)}{t_s(J)}$	$t_s(M)$	$\frac{t_s(M)}{t_s(J)}$	$t_s(M)$	$\frac{t_s(M)}{t_s(J)}$	$t_s(M)$	$\frac{t_s(M)}{t_s(J)}$
crack	1.29	1.82	1.28	1.44	1.56	1.13	2.15	0.82
4elt	1.31	1.44	1.52	1.32	1.90	0.87	3.14	1.03
t60k	3.24	1.25	3.56	1.13	4.65	1.07	6.35	1.00
dime20	22.61	2.42	23.66	2.36	22.53	1.90	24.18	1.42
144	23.05	1.08	25.05	0.91	26.82	0.70	29.80	0.54
m14b	48.77	1.77	49.79	1.36	49.88	0.97	58.84	0.78
fe-ocean	16.40	1.32	17.97	0.99	20.99	0.82	24.90	0.70
mesh1m	168.91	2.06	169.59	1.78	178.69	1.37	191.75	1.11
Average		1.65		1.41		1.10		0.92

Table 4: A comparison of timings for METIS, $t_s(M)$, and JOSTLE, $t_s(J)$

meshes or larger values of P) METIS, by accessing the data contiguously, gains from a relatively good cache hit rate, while JOSTLE, which is essentially accessing the data at random, starts to lose out. These differences can be quite marked; for $P = 16$, JOSTLE is up to 2.42 times faster (dime20) while for $P = 128$, METIS can take almost half the time (144). For the medium to large scale examples given here, however, METIS ranges from an average 65% slower for $P = 16$ to 8% faster for $P = 128$.

4.2 Different balancing schedules

mesh	$P=16$		$P=32$		$P=64$		$P=128$	
	$\lvert E_c(T_c)\rvert$	$\frac{\lvert E_c(T_c)\rvert}{\lvert E_c(T_2)\rvert}$	$\lvert E_c(T_c)\rvert$	$\frac{\lvert E_c(T_c)\rvert}{\lvert E_c(T_2)\rvert}$	$\lvert E_c(T_c)\rvert$	$\frac{\lvert E_c(T_c)\rvert}{\lvert E_c(T_2)\rvert}$	$\lvert E_c(T_c)\rvert$	$\frac{\lvert E_c(T_c)\rvert}{\lvert E_c(T_2)\rvert}$
crack	1238	1.06	1981	1.11	2803	1.04	4063	1.02
4elt	1055	1.04	1762	1.04	2841	1.02	4363	1.02
t60k	976	0.99	1603	1.01	2645	1.08	3695	1.03
dime20	1431	1.12	2572	1.13	3873	1.07	5910	1.07
144	44543	1.06	62334	1.02	87376	1.04	118068	1.04
m14b	51533	1.12	78226	1.07	111147	1.05	155252	1.05
fe-ocean	9155	1.03	15901	1.11	24629	1.07	34181	1.07
mesh1m	25549	1.04	38176	1.09	54068	1.03	75399	1.04
Average		1.06		1.07		1.05		1.04

Table 5: A comparison of cut edge results for a constant balancing schedule, $|E_c(T_c)|$, and the 2D schedule, $|E_c(T_2)|$

The balancing schedule derived in §2.2 is essentially an arbitrary heuristic and in this section we test some different schedules. Firstly Table 5 shows a comparison of the cut-edge weight for a constant schedule, $|E_c(T_c)|$, where θ_l is set to 1.03 for every graph G_l. For each value of P, the second column compares the results from this fixed schedule with the results in Table 2 using the 2D schedule and referred to as $|E_c(T_2)|$. As can be seen the constant schedule provides partition qualities which are always worse (except for t60k, $P = 16$); the average difference in the quality ranges between 4% and 7% over the different values of P and can be as bad as 12%. This constant schedule strategy is similar to that used by METIS (which also has an imbalance tolerance of 1.03) where balance is established early on in the expansion/refinement process (in the case of METIS, during the initial partitioning) and maintained thereafter and interestingly the average difference in the quality is about the same for the METIS results (Table 3).

Table 6 shows a comparison of the cut-edge weight for the 3D schedule, $|E_c(T_3)|$, mentioned in §2.2 and where the imbalance tolerance for graph G_l is set to $\theta_l = 1 + 3\left(\frac{P}{N_{l-1}}\right)^{\frac{1}{3}}$. Again for each value of P, the second column compares the results from this fixed schedule with the results in Table 2 using the 2D schedule, $|E_c(T_2)|$. Although there are some better results using the 3D schedule, on the average the results are slightly worse and the average difference in the quality ranges between 1% and 3%. One might suspect that the 3D meshes (the bottom four rows) would fare better with a 3D schedule than the 2D ones (the top 4 rows), but this does not seem to be borne out. In fact we have experimented with a number of different formulations and found the algorithm relatively insensitive to the schedule provided that the initial imbalance tolerance is sufficiently high.

Overall, we conclude from these results, together with the comparisons with METIS, and other experiments not presented here, that the use of a multilevel balancing schedule can improve the partition quality. The improvement is not enormous, because the multilevel paradigm with a static schedule already provides excellent results (and hence

mesh	$P = 16$		$P = 32$		$P = 64$		$P = 128$	
	$\lvert E_c(T_3)\rvert$	$\frac{\lvert E_c(T_3)\rvert}{\lvert E_c(T_2)\rvert}$	$\lvert E_c(T_3)\rvert$	$\frac{\lvert E_c(T_3)\rvert}{\lvert E_c(T_2)\rvert}$	$\lvert E_c(T_3)\rvert$	$\frac{\lvert E_c(T_3)\rvert}{\lvert E_c(T_2)\rvert}$	$\lvert E_c(T_3)\rvert$	$\frac{\lvert E_c(T_3)\rvert}{\lvert E_c(T_2)\rvert}$
crack	1182	1.01	1833	1.02	2700	1.00	3995	1.00
4elt	1066	1.05	1695	1.00	2751	0.99	4295	1.01
t60k	1010	1.03	1551	0.98	2513	1.03	3633	1.01
dime20	1274	1.00	2392	1.05	3811	1.05	5770	1.05
144	42744	1.02	61917	1.01	84407	1.01	113348	1.00
m14b	49133	1.07	72660	0.99	107222	1.01	147244	0.99
fe-ocean	9131	1.03	15086	1.05	22874	0.99	31856	1.00
mesh1m	22246	0.91	36319	1.03	53071	1.01	74225	1.02
Average		1.01		1.02		1.01		1.01

Table 6: A comparison of cut edge results for a 3-dimensional schedule, $|E_c(T_3)|$, and the 2-dimensional schedule, $|E_c(T_2)|$

the margin for improvement is small). However it does exist and provides on average a 5-6% decrease in the cut-edge weight.

5 Conclusions and future directions

We have presented an enhancement to the multilevel paradigm where the freedom allowed by a balancing schedule is used to find higher quality partitions. We have also presented a formulation of a Kernighan-Lin type partition optimisation algorithm which incorporates a diffusive balancing flow. The resultant algorithm has been shown to provide higher quality partitions than a state-of-the-art partitioner and, depending on granularity, to be up to 2.4 times as fast.

The algorithms are fairly simple to describe and relatively parameter-free and as a result provide an ideal setting for testing new ideas before implementing them within the framework of a fully parallel mesh partitioner. 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 and H. D. Simon. A Fast Multilevel Implementation of Recursive Spectral Bisection for Partitioning Unstructured Problems. *Concurrency: Practice & Experience*, 6(2):101–117, 1994.

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

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

[4] 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.

[5] M. Garey, D. Johnson, and L. Stockmeyer. Some simplified NP-complete graph problems. *Theoretical Computer Science*, 1:237–267, 1976.

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

[7] 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.

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

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

[10] Y. F. Hu and R. J. Blake. An optimal dynamic load balancing algorithm. Preprint DL-P-95-011, Daresbury Laboratory, Warrington, WA4 4AD, UK. (To be published in *Concurrency: Practice & Experience*), 1995.

[11] G. Karypis and V. Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. TR 95-035, Computer Science Department, University of Minnesota, Minneapolis, MN 55455, 1995.

[12] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. TR 95-064, Computer Science Department, University of Minnesota, Minneapolis, MN 55455, 1995.

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

[14] K. Schloegel, G. Karypis, and V. Kumar. Multilevel Diffusion Schemes for Repartitioning of Adaptive Meshes. TR 97-013, Dept. Computer Science, University of Minnesota, Minneapolis, MN 55455, 1997.

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

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

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

[18] C. Walshaw, M. Cross, and M. Everett. Dynamic mesh partitioning: a unified optimisation and load-balancing algorithm. Tech. Rep. 95/IM/06, University of Greenwich, London SE18 6PF, UK, 1995.

[19] C. Walshaw, M. Cross, and M. Everett. Parallel dynamic graph-partitioning for unstructured meshes. *J. Par. Dist. Comput.*, 1998. (in press).

[20] C. Walshaw, M. Cross, and M. Everett. Parallel Mesh Partitioning: a Multilevel Balancing and Refinement Algorithm. Tech. Rep. 98/IM/??, University of Greenwich, London SE18 6PF, UK, February 1998.