Dynamic mesh partitioning: a unified optimisation and load-balancing algorithm

C. Walshaw, M. Cross and M. G. Everett *
Centre for Numerical Modelling and Process Analysis,
University of Greenwich,
London, SE18 6PF,
UK.

Mathematics Research Report 95/IM/06

December '95

Abstract

A parallel method for dynamic partitioning of unstructured meshes is described. The method employs a new unified iterative optimisation technique which both balances the workload and attempts to minimise the interprocessor communications overhead. Experiments on a series of adaptively refined meshes indicate that the algorithm provides partitions of an equivalent quality to static partitioners (which do not reuse the existing partition) and much more quickly. Perhaps more importantly, the algorithm results in only a small fraction of the amount of data migration compared to the static partitioners.

Key words. graph-partitioning, unstructured meshes, dynamic load-balancing.

1 Introduction

The use of unstructured mesh codes on parallel machines can be one of the most efficient ways to solve large Computational Fluid Dynamics (CFD) and Computational Mechanics (CM) problems. Completely general geometries and complex behaviour can be readily modelled and, in principle, the inherent sparsity of many such problems can be exploited to obtain excellent parallel efficiencies. However, unlike their structured counterparts, one must carefully address the problem of distributing the mesh across the memory of the machine at runtime so that the computational load is evenly balanced and the amount of interprocessor communication is minimised. It is well known that this problem is NP complete, 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. [8].

An increasingly important area for mesh partitioning arises from problems in which the computational load varies throughout the evolution of the solution. For example, time-dependent unstructured mesh codes which use adaptive refinement can give rise to a series of meshes in which the position and density of the data points varies dramatically over the course of an integration and which may need to be frequently repartitioned for maximum parallel efficiency. This dynamic partitioning problem has not been nearly as thoroughly studied as the static problem but related work can be found in [4, 5, 6, 15, 25].

*Supported by the Engineering and Physical Sciences Research Council under grant reference number GR/K08284.

The dynamic evolution of load has three major influences on possible partitioning techniques; cost, reuse and parallelism. Firstly, the unstructured mesh may be modified every few time-steps and so the load-balancing must have a low cost relative to that of the solution algorithm in between remeshing. This may seem to restrict us to computationally cheap algorithms but fortunately, if the mesh has not changed too much, it is a simple matter to interpolate the existing partition from the old mesh to the new and use this as the starting point for repartitioning, [25]. In fact, not only is the load-balancing likely to be unnecessarily computationally expensive if it fails to use this information, but also the mesh elements will be redistributed without any reference to their previous 'home processor' and heavy data migration may result. Finally, the data is distributed and so should be repartitioned *in situ* rather than incurring the expense of transferring it back to some host processor for load-balancing and some powerful arguments have been advanced in support of this proposition, [15]. Collectively these issues call for parallel load-balancing and, if a high quality partition is desired, a parallel optimisation algorithm.

Load-balancing of this nature is referred to as quasi-dynamic, [27], or synchronous, [18], as it is carried out by all processors working collectively. A more dynamic or asynchronous type of load-balancing can occur when the partitioning runs in conjunction with the solver, i.e. the partitioner takes an iteration at every time-step to diffuse excess load away to neighbouring processors. For example, in simulating the process of metals casting, the code may be called on to solve for flow in the molten metal and stress where it has solidified, [1], and these regions may be constantly changing. The partitions therefore need to adapt dynamically to reflect the changing workload and, if the solvers act simultaneously, this may be possible by weighting the graph. (Note that if the solvers act independently, however, each different part of the domain may need to be partitioned separately and in this case graph weights may not work, [16].)

In this paper we describe a localised optimisation technique which marries a Kernighan Lin type optimisation method with a distributed load-balancing algorithm. It can be used for quasi-dynamic or, in principle, dynamic load-balancing, although we have not tested the latter case. The algorithm works both in serial and in parallel and we describe the parallel version in particular detail.

2 Notation and Definitions

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

We shall use the notation $\leftrightarrow$ to mean 'is adjacent to'. For example, for $u, v \in V$, $u \leftrightarrow v$ iff $\exists (u, v) \in E$. Also if $v \in V$ and $S, T \subset V$, $v \leftrightarrow S$ iff $\exists (u, v) \in E$ with $u \in S$ and similarly $S \leftrightarrow T$ iff $\exists (u, v) \in E$ with $u \in S$ & $v \in T$. We then denote the neighbourhood of a set of vertices S (the vertices adjacent to S) by $\Gamma(S)$ and thus the halo of subdomain S_p by $H_p := \Gamma(S_p)$. Table 1 gives a summary of these definitions.

We assume throughout the rest of this paper that the graph is connected (i.e. for all $u, v \in V$ there exists a path of edges from u to v). If this is not the case there are several possible options, such as running the partitioning code on each connected subgraph or even adding edges of zero weight to the graph to make it connected. We also assume that the application from which the graph arises is such that the computational load can be expressed by weighting the graph vertices although this may not always be the case if, for example, the underlying application uses a frontal solver, [21]. Finally note that we work in the data-parallel paradigm so that the vertices in each subdomain, S_p, are assigned to processor p, which also holds a one

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

Table 1: Definitions

deep halo, H_p, or read only copy of vertices adjacent to S_p.

2.1 The graph-partitioning problem

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. More precisely we seek $\mathcal{P}$ such that $S_p \leq W$ (although this is not always possible for graphs with non-unitary vertex weights) for $p \in P$ and such that $|E_c|$ is approximately minimised. The definition of the partition optimisation problem is the same but working from an initial (unbalanced) partition $\mathcal{P}_0$. It is a matter of contention whether $|E_c|$ is the most important metric for minimisation but see §3.1 for a further discussion on this topic.

2.2 Localisation and the subdomain graph

An important aim for any parallel algorithm is to keep communication as localised as possible to avoid contention and expensive global operations; this issue becomes increasingly important as machine sizes grow. Throughout the optimisation algorithm we localise the vertex migration with respect to the partition $\mathcal{P}$ by only requiring subdomains to migrate vertices to neighbouring subdomains. Thus a subdomain S_p will only need to migrate vertices to S_q if $S_p \leftrightarrow S_q$.

In this way the partition $\mathcal{P}$ induces a subdomain graph on G which we shall refer to as $\mathcal{P}(G) = \mathcal{P}(G)(P, C)$, an undirected graph of P subdomains (vertices) and C connections (edges). There is an edge $(p, q) \in C$ if $S_p \leftrightarrow S_q$. In addition, the weight on each processor $p \in P$ is given by $|p| = |S_p|$ and so $|P| = \sum_{p \in P} |S_p| = |V|$. Figure 1 gives an example of a partitioned graph (with unit vertex and edge weights) and the resulting subdomain graph (with vertex and edge weights as shown).

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

3 The optimisation algorithm

3.1 The gain and preference functions

A key concept in the following method is the idea of gain and preference functions. Loosely, the gain $g(v,q)$ of a vertex $v \in S_p$ can be calculated for every 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 $f(v)$ is then just the value of q which maximises the gain – i.e. $f(v) = q$ where q attains $\max_{r \in P} g(v,r)$.

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, however, there has been some debate about the most important quantity to minimise and in [20], Vanderstraeten *et al.* demonstrate that it can be extremely effective to vary the cost function based on a knowledge of the solver. Meanwhile, in [24] we show that the architecture of the parallel machine and how the partition is mapped down onto its communications network can also play an important role. 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. Thus in Figure 1 (assuming unit edge weights) there would be one less edge cut if vertex u migrated to subdomain p and so $g(u,p) = 1$. Similarly $g(u,q) = 0$ and hence $f(u) = p$. For vertex v, $g(v,p) = -1$ and $g(v,r) = -1$ and so a random choice must be made for $f(v)$. Note that here we do not allow the preference of a vertex to be its current subdomain since we may have to migrate vertices even if the net gain is negative.

3.2 Load-balancing

The load-balancing problem, i.e. how to distribute N tasks over a network of P processors so that none have more than $\lceil N/P \rceil$, is a very important area for research in its own right with a vast range of applications. The topic is introduced in [18] and some common strategies described. Much work has been carried out on parallel or distributed algorithms recently and, in particular, on diffusive algorithms, [3, 10]. Here, however, we use an elegant technique recently developed by Hu & Blake, [13], 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, ($L_{pp} =$ degree(S_p); $L_{pq} = -1$ if $S_p \leftrightarrow S_q, L_{pq} = 0$ otherwise), $b_p = |S_p| - W$ and 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) thus defines how much weight to transfer across edges of the subdomain graph and we use the optimisation technique to decide which vertices to move. We employ the algorithm as suggested in [13], solving iteratively with a parallel conjugate gradient solver, except that here we calculate the flow iteratively from the updates of $\mathbf{x}$ (rather than from the converged values of $\mathbf{x}$). In this case it is important for convergence to keep the Laplacian fixed and thus to

(a) restart the algorithm if an existing edge disappears from the subdomain graph (as can sometimes happen as vertices migrate)

(b) not include edges which have appeared since the most recent restart (or alternatively restart if a new edge appears).

3.3 A parallel optimisation mechanism

An algorithm which comes to mind for optimisation purposes is the Kernighan-Lin (KL) heuristic, [14], which maintains load-balance by employing pairwise exchanges of vertices. Unfortunately it has $O(n^2 \log n)$ complexity but a linear-time variant has been proposed by Fiduccia & Mattheyses (FM), [9]. The FM algorithm achieves this reduction partially by calculating swaps one vertex at a time rather than in pairs. Both algorithms have the possibility of escaping local minima traps by continuing to compute swaps even if the swap gain is negative in the hope that some later total gain may be positive. Here we introduce an algorithm largely inspired by the KL/FM algorithms but with several modifications to better suit our purposes.

Sublinearity. In common with Hendrickson & Leland, [12], we remark that the gains of each vertex do not have to be recalculated at each pass of the algorithm – having calculated them initially, we need only recalculate gains for migrating vertices and their immediate neighbours. However we extend the simplification by noting that, since edge weights are always positive integers, any vertex internal to a subdomain (i.e. $\in S_p - B_p$) will have negative gain. Although it is possible to devise arbitrary graphs in which an internal vertex may have maximum gain over the subdomain it seems to very occur very rarely for our sort of applications. Thus we only calculate initial gains for border vertices and thereafter for vertices coming in to the border. Of course, the full KL/FM algorithm continues to evaluate swaps even when the step gain is negative and if it explores the entire graph in this way it must eventually calculate the gain of every vertex. However, if as suggested by several authors, [11, 12], we terminate this process at some early stage, it is very likely that much of the graph will remain unexplored and this can make the whole algorithm cost sublinear and potentially insignificant for very coarse granularities.

Data organisation. It might seem that there is no saving in only calculating gains for border vertices (as the cost of calculating the gain is about the same as detecting border vertices) but this is not the case as every processor maintains a list of local vertices divided into three parts: internal, border and the hotlist (which are vertices either migrating or adjacent to migrating vertices and which may or may not be in the border). Thus, if a vertex moves from one subdomain to another it is moved into the hotlist of its destination subdomain and, in addition, any vertices adjacent to that vertex are moved into the hotlists of their respective subdomains. After all migration for that iteration has taken place each subdomain has simply to examine all the vertices in its hotlist and move them into the border or internal lists as required. Neither does this process absorb a lot of memory – just one double linked list per subdomain and hence one pair of pointers, forward and backward, per vertex. The list is ordered to have all the internal vertices, followed by the border vertices followed by the hotlist; two additional pointers, one to the start of the border vertices and one to the start of the hotlist, are kept. There is no need for sorting: transferring a vertex into a hotlist just means moving it to the end of the list, transferring a vertex out of the hotlist into the internal list means moving it to the start of the list and rather than transferring vertices out of the hotlist, we reset the hotlist start pointer to point to the end of the subdomain list after the rebordering phase. This means that detection of border vertices by testing *all* vertices need only take place once at the start of the optimisation.

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

These simplifications considerably ease the effort of finding the vertex gains since we need only calculate them for the borders of each subdomain. Also most vertices in B_p will only be adjacent to one subdomain, q say, and in this case the preference is simply q. Where 3 or more subdomains meet, the preferences need to be explicitly calculated and if more than one subdomain attains the maximum gain, a pseudo-random choice is made (see below).

Duplication. For either algorithm, 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, [17]. For our parallel implementation we overcome some of the problems by replicating the transfer decision process on both sides of each subdomain interface. Thus a processor p 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. It does not use the information about transfers of vertices that it does not own, but the duplication of this transfer scheme allows any pair of processors to arrive *independently* at the same optimisation. Although this may be computationally inefficient (because the workload is doubled) it avoids both the need for a 'hand-shaking' communication for each pair of vertices and also for pairwise exchanges between neighbouring processors.

3.4 A unified optimisation and load-balancing algorithm

We combine the load-balancing and optimisation as follows.

At each iteration a processor, p, calculates the preference and gain of its own border vertices and a halo update is carried out to inform the neighbouring processors. Next, for each S_p-S_q interface with a neighbouring processor q, it calculates the flow across the edge (S_p, S_q) and creates a list of border vertices $v \in B_p$ which have preference $f(v) = q$ and a list of halo vertices $u \in H_p$ which have preference $f(u) = p$. Vertices are then iteratively selected from either list so as to firstly satisfy the flow as far as possible and secondly maximise the gain as much as possible. The algorithm for this inner loop is shown in Figure 2. The whole optimisation terminates when the load is balanced and no further gains in cost are possible (although simple checks need to be included to trap cyclic behaviour).

```
optimum flow = flow
optimum gain = 0
while (untested border vertices) {
  find vertex to minimise |flow| & maximise gain
  gain = gain + vertex gain
  flow = flow +/- vertex weight
  adjust adjacent vertex gains
  if (|flow| < |optimum flow|
   || (|flow| == |optimum flow|
      && gain > optimum gain)) {
    optimum flow = flow
    optimum gain = gain
    optimum migration = vertices tested so far
   }
 }
```

Figure 2: The unified optimisation and load-balancing algorithm

Suppose, for example, that the required flow from S_p to S_q is F, a positive integer. The processor (on either side of the interface) searches the list of vertices owned by p for a vertex, v, which minimises the absolute value of $|v| - F$ (i.e. if possible we choose a vertex with weight F, or if F is larger than all the vertex weights we look for the largest weight, or if smaller we look for the smallest weight). In the case of several appropriate vertices we choose the one with the highest gain, and, in the case of another tie, a pseudo-random choice is made (see below). If $F = 0$ both lists are searched (with the same criterion – in this case we want to minimise $|v|$). Having selected a vertex v, it is removed from its list and the flow, total gain and the gains of all vertices adjacent to v are adjusted as if v had migrated (as in the Kernighan-Lin algorithm). If the current flow is smaller in magnitude than the optimal flow or, if equal in magnitude, if the gain is greater than the optimal gain then the optimal migration is recorded to be all vertices that have been selected so far and the optimal flow and gains are updated. The algorithm terminates when a list that is to be searched is exhausted and then vertices included in the optimal schedule are migrated.

Example flow and gain evolution is shown in Figure 3 and demonstrate how these quantities can vary as each vertex is selected and how the algorithm essentially has three (indistinct) phases. The first phase, corresponding to the left hand side of the chart, occurs when the required flow is greater than the total weight

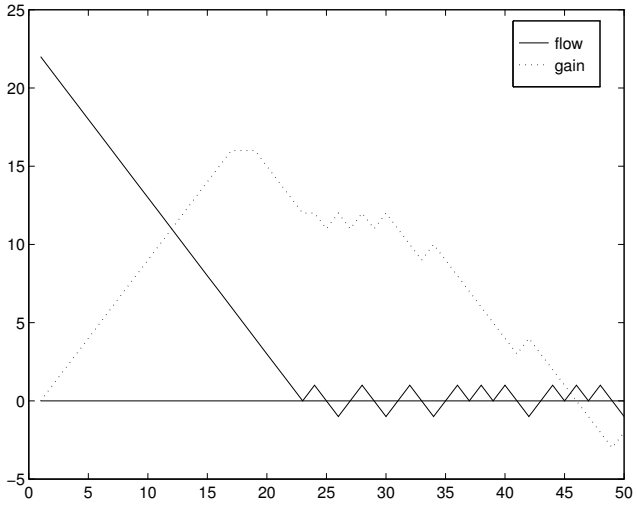

Figure 3: Example flow and gain

of all the vertices in the appropriate border and in this case no searching is necessary as all will be migrated. The third phase, corresponding to the left hand side of the chart, is when the load is balanced and the algorithm is just looking to optimise the gain while the flow oscillates around zero (much in the same way that the FM algorithm works). The middle phase is a mixture of balancing and optimisation. At the end of the iterations, optimal flow occurs at all points where the flow is zero (or as small in magnitude as possible if the border weight is not large enough) and the optimal migration is then the point of optimal flow where the gain is maximised.

There are a couple of implementation details to be noted about this method. Firstly, whenever a subdomain makes a decision about sorting (or randomly selecting the starting list) it is *essential* that its neighbour makes the same choice; otherwise they might reach different migration schemes and increase the cut-edge weight. Thus when searching, if two vertices have the same weight and gain, they are sorted on some other factor (and for this reason it is of no benefit to use the bucket sort of Fiduccia and Mattheyses). For simplicity we use the vertex index although we have also tried using a random number generator (seeded with the vertex index) without any great change in behaviour. For a similar reason, when adjusting the gain of vertices in B_p on the S_p-S_q interface, we do not adjust those vertices which have a different preference r as this could have a knock-on effect on the migration scheme for S_p which would not be seen by subdomains S_q and S_r. Also for adjusting the gain of vertices in the halo H_p, note that processor p needs to know about edges between vertices within the halo.

4 Experimental results

The software tool written at Greenwich and which we have used to test the new optimisation technique is known as JOSTLE. For the purposes of this paper it is run in two configurations, dynamic and static. The dynamic configuration reads in an existing partition and uses the algorithm described here to balance and optimise the partition. The static version employs the greedy algorithm, [7], to generate an initial partition and, in addition to the algorithm described here, uses an optimisation technique, fully described in [23], which attempts to minimise the 'surface energy' of the subdomains.

In order to demonstrate the quality of the partitions we have compared the method with two of the most popular partitioning algorithms, Greedy and Multilevel Recursive Spectral Bisection (MRSB). The Greedy algorithm, [7], is actually performed as part of the JOSTLE code. It is fast but not particularly good at min-

imising either or $|E_c|$. MRSB, on the other hand, is a highly sophisticated method, good at minimising $|E_c|$ but suffering from relatively high runtimes, [2]. The MRSB code was made available to us by one of its authors, Horst Simon, and run unchanged with a contraction thresholds of 100.

The test meshes have been taken from an example contained in the DIME (distributed irregular mesh environment) software package, [26], freely available by anonymous ftp from `ftp.ccsf.caltech.edu` in `dime/dime.src.tar.Z`. The particular application solves Laplace's equation with Dirichelet boundary conditions on a square domain with an S-shaped hole and using a triangular finite element discretisation. The problem is repeatedly solved by Jacobi iteration, refined based on this solution and then load-balanced. A very similar set of meshes has previously been used for testing mesh partitioning algorithms and details about the solver, the domain and DIME can be found in [27].

The following experiments were carried out on a Sun 20 with a 75 MHz CPU and 128 Mbytes of memory. The code runs in parallel using MPI for message passing but at the time of writing the authors are only able to test it on ethernet connected workstations and parallel performance results are therefore deferred to [22]. We use three metrics to measure the performance of the algorithms – the total weight of cut edges, $|E_c|$, the execution time in seconds of each algorithm, $t(s)$, and the percentage of vertices which need to be migrated.

		imbalance %		$\|E_c\|$		
mesh	V	initial	final	initial	final	migrated %
1	1552	30.93	0.00	133	150	13.72
2	2134	36.57	0.00	167	179	14.76
3	2886	19.34	0.00	206	206	8.00
4	3917	10.61	0.00	248	248	6.36

Table 2: The dynamic results over the 4 meshes

Table 2 shows the results for the dynamic technique on the series of meshes partitioned into 16 subdomains. The initial mesh, mesh 0, is partitioned with the static version of JOSTLE. Subsequently at each refinement, the existing partition is interpolated onto the new mesh using the techniques described in [25] (essentially, new elements are owned by the processor which owns their parent). The new partition is then optimised and balanced with the dynamic version of JOSTLE. The third and fourth columns of the table give the percentage imbalance before and after partitioning and show that it is always driven down to zero (although the fact that the load is discrete rather than continuous may prevent it from reaching zero). The fifth and sixth columns show the weight of cut edges, $|E_c|$, before and after repartitioning. In two cases there has been a slight increase and this is because suboptimal migration has been made to satisfy the load-balancing requirements. Finally, the last column shows the percentage of vertices that have been migrated by the partitioner.

method	$\|E_c\|$	$t(s)$	migrated
JOSTLE (dynamic)	196	0.04	10.71%
JOSTLE (static)	165	0.87	89.21%
MRSB	158	1.93	89.96%
GREEDY	273	0.02	88.14%

Table 3: Average results over the 4 meshes: average V = 2622, average E = 3812

Table 3 compares the four different partitioning methods with the results averaged over the 4 meshes. The three high quality partitioners all give similar values for $|E_c|$ with MRSB giving marginally the best results. The dynamic algorithm provides slightly lower quality partitions than the other two. In terms of execution time, the dynamic method is far faster than MRSB (˜50 times), considerably faster than static JOSTLE (˜20 times) and about twice as slow as the GREEDY algorithm although providing a far better partition. It is the final column which is perhaps the most telling though. Because the static partitioners take no account of the existing distribution they result in a vast amount of data migration. The dynamic algorithm, on the other hand, migrates very few of the vertices despite balancing from an average initial imbalance of 24.36%.

It is worth remarking here that the dynamic algorithm can provide higher quality partitions when used in conjunction with a graph reduction/multilevel technique which gives it a more global perspective by contracting the graph down to a much smaller size. However, there is a trade off between the quality of the partition and the amount of data needed to be migrated to obtain it and illustrative results are to be found in [22].

5 Conclusion

We have described a new method for optimising and load-balancing graph partitions with a specific focus on its application to the dynamic mapping of unstructured meshes onto parallel computers. In this context the graph-partitioning task can be very efficiently addressed by reoptimising the existing partition, rather than starting the partitioning from afresh. For the experiments reported in this paper, the procedures are an order of magnitude faster than static techniques, provide partitions of similar quality and, in comparison, involve the migration of a fraction of the data.

Acknowledgements

We would like to thank Horst Simon for the copy of his Multilevel Recursive Spectral Bisection code.

References

[1] C. Bailey, P. Chow, M. Cross, Y. Fryer, and K. Pericleous. Multiphysics Modelling of the Metals Casting Process. *Proceedings of the Royal Society*, 1995. (in press).

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

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

[4] R. Diekmann, D. Meyer, and B. Monien. Parallel Decomposition of Unstructured FEM-Meshes. In A. Ferreira and J. Rolim, editors, *Proc. Irregular '95: Parallel Algorithms for Irregularly Structured Problems*, pages 199–215. Springer, 1995.

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

[6] R. Van Driessche and D. Roose. An Improved Spectral Bisection Algorithm and its Application to Dynamic Load Balancing. Rep. TW 193, Dept. Computer Science, Katholieke Universiteit Leuven, 1993.

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

[8] 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.

[9] 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.

[10] 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.

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

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

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

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

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

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

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

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

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

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

[21] D. Vanderstraeten, R. Keunings, and C. Farhat. Beyond Conventional Mesh Partitioning Algorithms and the Minimum Edge Cut Criterion: Impact on Realistic Applications. In D. Bailey *et al*, editor, *Parallel Processing for Scientific Computing*, pages 611–614. SIAM, 1995.

[22] C. Walshaw, M. Cross, and M. Everett. Parallel Mesh Partitioning. (in preparation).

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

[24] C. Walshaw, M. Cross, M. Everett, S. Johnson, and K. McManus. Partitioning & Mapping of Unstructured Meshes to Parallel Machine Topologies. In A. Ferreira and J. Rolim, editors, *Proc. Irregular '95: Parallel Algorithms for Irregularly Structured Problems*, volume 980 of *LNCS*, pages 121–126. Springer, 1995.

[25] C. H. Walshaw and M. Berzins. Dynamic load-balancing for PDE solvers on adaptive unstructured meshes. *Concurrency: Practice & Experience*, 7(1):17–28, 1995.

[26] R. D. Williams. DIME: Distributed Irregular Mesh Environment. Caltech Concurrent Computation Report C3P 861, 1990.

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