Parallel dynamic graph-partitioning for unstructured meshes

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

Mathematics Research Report 97/IM/20

March 27, 1997

Abstract

A parallel method for the dynamic partitioning of unstructured meshes is described. The method introduces a new iterative optimisation technique known as relative gain optimisation 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 or higher quality to static partitioners (which do not reuse the existing partition) and much more rapidly. 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, adaptive unstructured meshes, load-balancing, parallel computing.

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. An important consideration, however, is 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. [2, 11, 16].

An increasingly important area for mesh partitioning, however, arises from problems in which the computational load varies throughout the evolution of the solution. For example, heterogeneity in either the computing resources (e.g. processors which are unevenly matched or not dedicated to single users) or in the solver (e.g. solving for flow or stress in different parts of the domain in a multiphysics casting simulation, [14]) can result in load-imbalance and poor performance. Alternatively, 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, 13, 17, 21].

The dynamic evolution of load has three major influences on possible partitioning techniques; cost, reuse and parallelism. Firstly, frequent load-balancing may be required and so must have a low cost relative to that of the solution algorithm in between. This could potentially restrict the use of high quality partitioning 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, [21]. 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, [13]. Collectively these issues call for parallel load-balancing and, if a high quality partition is desired, a parallel optimisation algorithm.

In this paper we describe such a parallel optimisation technique (Section 2) which incorporates a distributed load-balancing algorithm and which provides an extremely fast solution to the problem of dynamically load-balancing unstructured meshes. In addition, a parallel graph contraction technique (described in Section 3) can be employed to enhance the partition quality and the resulting strategy (which can also be applied to static partitioning problems) outperforms or matches results from existing state-of-the-art static mesh partitioning algorithms.

Here, in particular, we focus on the case arising from adaptively refined meshes where we assume that the mesh will be repartitioned after each refinement phase. However, the method is also applicable to the more general case where load may be constantly varying, and in [1] a method for determining how frequently to partition (for maximum efficiency) is described, together with examples using the same partitioning techniques.

1.1 Notation and definitions.

Let $G = G(V, E)$ be an undirected graph of V vertices with E edges which represent the data dependencies in the mesh and let P be a set of processors. We assume that both vertices and edges are weighted (with positive integer values) and that $|v|$ denotes the weight of a vertex v, $|S| := \sum_{v \in S} |v|$ the weight of a subset $S \subset V$ and similarly for edges. Once the vertices are partitioned into P sets we denote the subdomains by S_p, for $p \in P$ and 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 $u, v \in V$, $u \leftrightarrow v$ if there exists $(u, v) \in E$.

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 a partition such that $S_p \leq W$ for $p \in P$ (although this is not always possible for graphs with non-unitary vertex weights) and such that $|E_c|$ is minimised (though see §2.2).

1.2 Parallelisation

The algorithms described run equally well in parallel or in serial and for the parallel version we use the single program, multiple data (SPMD) paradigm with message passing in the (reasonable) expectation that the underlying unstructured mesh application will do the same. To this end, each processor is assigned to a subdomain and stores a double linked-list of the vertices within that subdomain. However, each processor also maintains a read only 'halo' of neighbouring vertices in other subdomains. For the serial version the migration of vertices simply involves transferring them from one linked-list to another. In parallel, however, this task is far more complicated as migrating vertices, together with newly created halo vertices, must be packed into messages, sent off to the destination processor, unpacked, and the pointer based data structure recreated there. In addition, 'halo updates' must be regularly carried out to inform neighbouring processors of new values attached to their halo vertices. Since the existence of $|V|$ length arrays is simply not scalable in memory terms, location of vertices locally on a processor is carried out using their global index and some sophisticated hash table and binary tree searches.

2 Optimisation

In this section we present a new parallel iterative algorithm for load-balancing and optimising unstructured mesh partitions. The method is based on the concept of *relative gain*, first outlined in [18] and, in common with similar techniques, e.g. [11, 16], we localise the vertex migration with respect to the current partition by only allowing vertices on subdomain borders to migrate to neighbouring subdomains (i.e. to subdomains in which a neighbouring vertex lies).

2.1 Load-balancing

The method includes an iterative load-balancing scheme which runs alongside the optimisation to evenly distribute the workload over the processors. The generalised load-balancing problem is a very important area for research in its own right with a vast range of applications and here we use an elegant technique recently developed by Hu & Blake, [9]. It is related to the commonly used diffusive methods, e.g. [3], but has faster convergence 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, ($L_{pp} = \text{degree}(S_p)$; $L_{pq} = -1$ 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 $x_p - x_q$. Typically this system is solved iteratively with a conjugate gradient algorithm and is similar to diffusive methods in that each step is like a diffusive step, but with the diffusion coefficients being determined iteratively using a conjugate gradient search rather than being fixed throughout the procedure.

This algorithm (or, in principle, any other distributed load-balancing algorithm) then determines how much weight to transfer across the edges of the subdomain graph and we use f_{pq} to denote the required flow from S_p to S_q. Note that the flow is positive ($f_{pq} \geq 0$) and unidirectional (i.e. if $f_{pq} > 0$ then $f_{qp} = 0$ and vice-versa) and even if this is not the case we can force it to be so by setting $f_{pq} = f_{pq} - \max(f_{pq}, f_{qp})$ and $f_{qp} = f_{qp} - \max(f_{pq}, f_{qp})$. Note also that if there is not a sufficient weight of vertices in a particular border to satisfy the required flow, then outstanding flow is recorded and added in at the next iteration.

2.2 The gain and preference functions

A key concept in many graph partition optimisation algorithms is the idea of gain and preference functions. 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 preference $f(v)$ is then just the value of q which maximises the gain – i.e. $f(v) = q$ where $g(v, q)$ 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 [15], Vanderstraeten *et al.* demonstrate that it can be extremely effective to vary the cost function based on a knowledge of the solver. Meanwhile, in [20] 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, $|E_c|$.

2.3 Relative gain optimisation

Having determined the required flow across the edges of the subdomain graph (§2.1) we need to migrate vertices from adjacent subdomains in order to satisfy that flow. Choosing appropriate vertices to migrate is not an easy task because we also wish to optimise the partition quality with respect to the cost function. Indeed, in order to obtain partitions of the highest quality, it is likely that vertices will need to be exchanged even if there is no flow required. Simply moving vertices with the highest gain is not a satisfactory solution, however, as it means that adjacent vertices may be swapped simultaneously (an event often known as a collision) and this may lead to an *increase* in the cost. We have previously addressed this problem by using a Kernighan-Lin (KL) type algorithm run in the boundary regions alone, [17, 19], but this causes a loss of efficiency in parallel because, in order to retain the hill-climbing abilities of the algorithm, processors must maintain edges between halo vertices and this turns out to be a costly task. In addition, we have tried a red-black colouring strategy where, at each iteration, vertices from only one of each pair of neighbouring subdomains are migrated. The choice of which subdomain is decided dynamically according to which direction the required flow is in or which subdomain has the greatest weight of vertices with positive gain. The algorithm is fast and efficient but we have not succeeded in generating the highest quality of partitions using this technique. This method has been independently discovered by Karypis & Kumar, [12]. Here, however, we introduce a new strategy which uses the concept of *relative gain*.

Bulk migration. The first part of each iterative step is to use a simple formula based on both the flow and the total weight of vertices with positive gain to determine how much total load to migrate. Firstly, for the interface between subdomains S_p and S_q, define border regions, B_{pq}, as the set of vertices in B_p (the border of S_p) whose preference is q, i.e. $B_{pq} = \{v \in B_p : f(v) = q\}$ and let g_{pq} be the total weight of vertices in B_{pq} with gain > 0 (and similarly for B_{qp} and g_{qp}). Then if $d = \max(g_{pq} - f_{pq} + g_{qp} - f_{qp}, 0)$, the load to be migrated from p to 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.

Relative gain. To determine which vertices to migrate we use the concept of relative gain which we define as follows; for a vertex $v \in B_{pq}$ 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 $g(v, q) - \sum_{u \in \Gamma_q(v)} g(u, p)/O[\Gamma_q(v)]$ (where $O[\Gamma_q(v)]$ represents the number of vertices in $\Gamma_q(v)$). Put more simply, the relative gain of a vertex v is just the gain of v less the average gain of opposing vertices, and gives an indication of which are the best vertices to move in order to avoid collisions. Thus to prioritise the migration, 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. For the parallel version, at each iteration the gains of border vertices are calculated locally (in parallel) and propagated to neighbouring processors via a halo update operation (§1.2) and then the relative gains may be calculated locally.

Convergence. The algorithm is not as predictable as either Kernighan-Lin optimisation, or the red-black 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 since the formulation takes into account the likelihood of opposite vertices migrating. The method usually converges, but because of this impreciseness, it is necessary to prevent cyclic 'thrashing' by terminating the optimisation after a couple of iterations if the cost has not decreased.

3 Graph reduction

The above algorithm provides what is essentially very localised optimisation and it has been recognised for some time that an effective way of both speeding up optimisation and, perhaps more importantly, giving it a more global perspective is to use graph reduction. 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 optimise these reduced size graphs. It is a common technique and has been used by several authors in various ways – for example, in a multilevel way analogous to multigrid techniques, [2, 8], and in an adaptive way analogous to dynamic refinement techniques, [21]. Several algorithms for carrying out the reduction can be found in [11].

To create a coarser graph $G'(V', E')$ from $G(V, E)$ we use a variant of the edge contraction algorithm proposed by Hendrickson & Leland, [8], and improved by Karypis & Kumar in [11]. 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, $u_1, u_2 \in V$ say, at either end of it are merged to form a new vertex $v \in V'$ with weight $|v| = |u_1|+|u_2|$. Edges which have not been collapsed are inherited by the reduced graph 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, $|V| = |V'|$ 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 a 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.

3.1 Local matching

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

3.2 Parallel matching

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

Note that for algorithm to avoid selection cycles (i.e. u selects v, v selects w and w selects u) and thus locking, the selection of a match is made in a pseudo-random way. For each edge (u, v) a random number generator is seeded with the sum of the global indices of u and v and random number generated. The edge with the highest number is then selected. The reasoning behind this is explained in [10].

3.3 Construction of the reduced graph

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

3.4 Interpolation

Having optimised a reduced graph G', the partition must be interpolated onto its parent G. The interpolation itself is a trivial matter; if a vertex $v' \in G'$ is in subdomain S'_p then the matched pair of vertices that it represents, v_1 & v_2, will be in S_p. However, during optimisation, we only migrate the vertices of G' (and not the entire parent hierarchy) and so if v' has migrated from its original processor then v_1 & v_2 will not be stored locally and may not even be stored on a neighbouring processor. Thus for each $v' \in S'_p$ which was not created locally, processor p must inform the originating processor (which can be easily determined by the global index of v' – see §3.3) where it now resides and the parent vertices must be migrated to S_p. Each processor therefore sends messages to the originators of all the vertices it owns, receives the corresponding messages about vertices it created and which have migrated away (by continually polling – waiting on incoming messages – until it knows the destination of all the vertices it created) and then migrates parent vertices to their child's new subdomain.

4 Experimental results

The software tool written at Greenwich and which we have used to test the optimisation and graph reduction algorithms is known as JOSTLE. For the purposes of this paper it is run in three configurations, dynamic (JOSTLE-D), multilevel-dynamic (JOSTLE-MD) and multilevel-static (JOSTLE-MS). The dynamic configuration, JOSTLE-D, reads in an existing partition and uses the algorithm described in Section 2 to balance and optimise the partition. The multilevel-dynamic, JOSTLE-MD, uses the same procedure but additionally uses graph reduction (Section 3) to improve the partition quality. The static version, JOSTLE-MS, carries out graph reduction on the unpartitioned graph, employs the greedy algorithm, [7], and an optimisation technique, fully described in [16], which attempts to minimise the 'surface energy' of the subdomains, to generate an initial partition of the coarsest graph and, then uses the algorithm described in Section 2 to optimise each of the multilevel graphs.

The test meshes have been taken from an example contained in the DIME (distributed irregular mesh environment) software package, [22], 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 [23]. The particular series of ten meshes and the resulting graphs that we used range in size from the first one which contains 23,787 vertices and 35,281 edges to the final one which contains 224,843 vertices and 336,024 edges.

4.1 Comparison results

In order to demonstrate the quality of the partitions we have compared the method with three of the most popular partitioning schemes, METIS, GREEDY and Multilevel Recursive Spectral Bisection (MRSB). Of the three METIS is the most similar to JOSTLE, employing a graph reduction technique and iterative optimisation. The version used here is `kmetis` from the most recent public distribution, `metis-2.0.5.tar.gz`, available by anonymous ftp from `ftp.cs.umn.edu` in `/dept/users/kumar/metis/`. The GREEDY algorithm, [7], is actually performed as part of the JOSTLE-MS configuration and is fast but not particularly good at minimising $|E_c|$. 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 following experiments were carried out in serial on a Sun SPARC Ultra with a 140 MHz CPU and 64 Mbytes of memory. 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, M.

For the two dynamic configurations, the initial mesh is partitioned with the static version – JOSTLE-MS. Subsequently at each refinement, the existing partition is interpolated onto the new mesh using the techniques described in [21] (essentially, new elements are owned by the processor which owns their parent) and the new partition is then optimised and balanced.

	$P = 16$			$P = 32$			$P = 64$		
method	$\|E_c\|$	$t(s)$	M %	$\|E_c\|$	$t(s)$	M %	$\|E_c\|$	$t(s)$	M %
JOSTLE-D	942	0.51	0.54	1551	0.64	1.80	2598	0.85	3.76
JOSTLE-MD	846	2.39	4.92	1447	2.60	6.26	2410	3.02	8.82
JOSTLE-MS	879	3.96	93.96	1488	4.19	92.77	2417	4.95	99.00
METIS	913	4.83	94.36	1543	4.91	95.94	2427	5.15	97.95
MRSB	939	55.85	83.54	1577	71.42	90.01	2520	87.34	95.07
GREEDY	1816	0.77	81.62	2897	0.83	90.64	4300	1.00	94.42

Table 1: Average results over the 9 meshes

Table 1 compares the six different partitioning methods for $P = 16$, 32 and 64 with the results averaged over the last 9 meshes (i.e. not including the static partitioning results for the first mesh). The high quality partitioners – both JOSTLE multilevel configurations, METIS and MRSB – all give similar values for $|E_c|$ with MRSB giving marginally the worst results and JOSTLE-MD giving the best. In general, JOSTLE-D, without the benefit of graph reduction, provides slightly lower quality partitions but approximately equivalent to those of MRSB. In terms of execution time, JOSTLE-D is slightly faster than GREEDY with both of them being much faster than any of the multilevel algorithms. Of these multilevel algorithms, however, JOSTLE-MD is considerably faster than JOSTLE-MS and METIS, and MRSB is by far the slowest. 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 configurations, JOSTLE-D and JOSTLE-MD, on the other hand, migrate very few of the vertices. As could be expected JOSTLE-MD migrates somewhat more than JOSTLE-D since it does a more thorough optimisation.

Taking the results as a whole, the multilevel-dynamic configuration, JOSTLE-MD, provides the best partitions very rapidly and with very little vertex migration. If a slight degradation in partition quality can be tolerated however, the JOSTLE-D configuration load-balances and optimises even more rapidly, faster than the GREEDY algorithm, with even less vertex migration.

4.2 Parallel timings

Achieving high parallel performance for parallel partitioning codes such as JOSTLE is not as easy as, say, a typical CFD or CM code. For a start the algorithms use only integer operations and so there are no MFlops to 'hide behind'. In addition, most of the work is carried out on the subdomain boundaries and so very little of the actual graph is used. Also the partitioner itself may not necessarily be well load-balanced and the communications cost may dominate on the coarsest reduced graphs. On the other hand, as was explained in Section 1, partitioning on the host may be impossible or at least much more expensive and if the cost of partitioning is regarded (as it should be) as a parallel overhead, it is usually extremely inexpensive relative to the overall solution time of the problem.

		$P = 16$			$P = 32$			$P = 64$		
V	E	$t_s(s)$	$t_p(s)$	speed up	$t_s(s)$	$t_p(s)$	speed up	$t_s(s)$	$t_p(s)$	speed up
31172	46309	0.26	0.06	4.33	0.32	0.06	5.33	0.42	0.07	6.00
40851	60753	0.34	0.07	4.86	0.44	0.07	6.29	0.59	0.08	7.37
53338	79415	0.38	0.06	6.33	0.55	0.08	6.88	0.78	0.11	7.09
69813	104034	0.56	0.10	5.60	0.71	0.09	7.89	0.85	0.13	6.54
88743	132329	0.71	0.13	5.46	0.82	0.10	8.20	1.00	0.09	11.11
115110	171782	0.82	0.11	7.45	1.03	0.11	9.36	1.30	0.11	11.82
146014	218014	1.14	0.16	7.12	1.29	0.13	9.92	1.60	0.13	12.31
185761	277510	1.47	0.21	7.00	1.58	0.15	10.53	2.06	0.16	12.88
224843	336024	1.63	0.19	8.58	1.97	0.18	10.94	2.30	0.14	16.43

Table 2: Serial and parallel timings for the JOSTLE-D configuration

		$P = 16$			$P = 32$			$P = 64$		
V	E	$t_s(s)$	$t_p(s)$	speed up	$t_s(s)$	$t_p(s)$	speed up	$t_s(s)$	$t_p(s)$	speed up
31172	46309	0.93	0.35	2.66	1.12	0.26	4.31	1.42	0.25	5.68
40851	60753	1.25	0.40	3.12	1.50	0.32	4.69	1.99	0.32	6.22
53338	79415	1.53	0.97	1.58	1.73	0.30	5.77	2.25	0.32	7.03
69813	104034	1.99	0.48	4.15	2.22	0.32	6.94	2.73	0.33	8.27
88743	132329	2.44	0.49	4.98	2.83	0.40	7.08	3.34	0.38	8.79
115110	171782	3.15	0.61	5.16	3.51	0.44	7.98	4.13	0.39	10.59
146014	218014	3.98	0.75	5.31	4.58	0.56	8.18	5.39	0.55	9.80
185761	277510	5.03	0.87	5.78	5.50	0.63	8.73	6.45	0.55	11.73
224843	336024	6.04	0.95	6.36	6.66	0.67	9.94	7.66	0.59	12.98

Table 3: Serial and parallel timings for the JOSTLE-MD configuration

Tables 2 and 3 give serial and parallel timings for the JOSTLE-D and JOSTLE-MD configurations respectively on the 512 node Cray T3E at HLRS, the High Performance Computer Centre at the University of Stuttgart. The parallel version uses the MPI communications library although we are working on a `shmem` version which could be expected to show even faster timings. These demonstrate good speedups for this sort of code and more importantly, very low overheads (always less than a second) for the parallel partitioning. Finally note that the partitions obtained for the parallel version of JOSTLE are exactly the same as those of the serial version.

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 dynamic procedures are much faster than static techniques, provide partitions of similar or higher quality and, in comparison, involve the migration of a fraction of the data.

Acknowledgements. We would like thank HLRS, the High Performance Computer Centre at the University of Stuttgart, for access to the Cray T3E.

References

[1] A. Arulananthan, S. Johnson, K. McManus, C. Walshaw, and M. Cross. A Generic Strategy for Dynamic Load Balancing of Distributed Memory Parallel Computational Mechanics Using Unstructured Meshes. Submitted to Parallel CFD '97, 1997.

[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. *Parallel Comput.*, 21:29–48, 1995.

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

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

[9] 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.

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

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

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

[14] K. McManus, C. Walshaw, M. Cross, P. Leggett, and S. Johnson. Evaluation of the JOSTLE mesh partitioning code for practical multiphysics applications. In A. Ecer *et al*, editor, *Parallel Computational Fluid Dynamics: Implementations and Results Using Parallel Computers*, pages 673–680. Elsevier, Amsterdam, 1996. (Proceedings of Parallel CFD'95, Pasadena, 1995).

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

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

[17] 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.

[18] C. Walshaw, M. Cross, and M. Everett. Dynamic load-balancing for parallel adaptive unstructured meshes. In M. Heath *et al*, editor, *Parallel Processing for Scientific Computing*. SIAM, Philadelphia, 1997.

[19] C. Walshaw, M. Cross, and M. Everett. Parallel Partitioning of Unstructured Meshes. In P. Schiano *et al*, editor, *Parallel Computational Fluid Dynamics: Algorithms and Results Using Advanced Computers*, pages 174–181. Elsevier, Amsterdam, 1997. (Proceedings of Parallel CFD'96, Capri, 1996).

[20] 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.

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

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

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