Dynamic mesh partitioning & load-balancing for parallel computational mechanics codes *

C. Walshaw and M. Cross
Computing & Mathematical Sciences, University of Greenwich,
Park Row, Greenwich, London, SE10 9LS, UK.
C.Walshaw@gre.ac.uk; http://www.gre.ac.uk/~c.walshaw

Abstract

We discuss the load-balancing issues arising in parallel mesh based computational mechanics codes for which the processor loading changes during the run. We briefly touch on geometric repartitioning ideas and then focus on different ways of using a graph both to solve the load-balancing problem and the optimisation problem, both locally and globally. We also briefly discuss whether repartitioning is always valid. Sample illustrative results are presented and we conclude that repartitioning is an attractive option if the load changes are not too dramatic and that there is a certain trade-off between partition quality and volume of data that the underlying application needs to migrate.

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

1 Introduction

Mesh partitioning has been developed recently as an important enabling technology for mapping unstructured meshes arising from applications such as Computational Fluid Dynamics (CFD) and Computational Mechanics (CM) onto parallel machines. Typically the aim is to distribute the mesh so that each processor has an equal share of the computational load whilst ensuring that communication overhead is kept to a minimum. 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].

*Invited lecture. In: Proc. Parallel & Distributed Computing for Computational Mechanics, Weimar, Germany, 1999

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 regions, the size & shape of which change, in a multi-physics casting simulation, [22]) 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 an interesting overview can be found in [11].

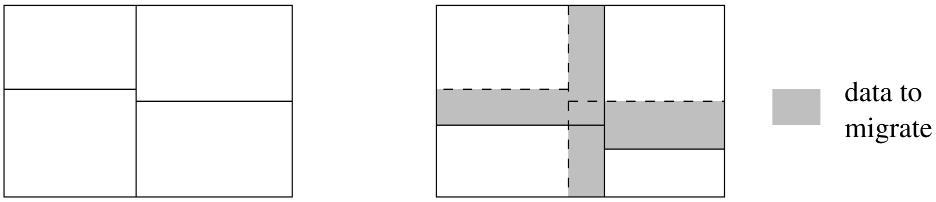
The dynamic evolution of load has three major influences on possible partitioning techniques; cost, reuse and parallelism. Firstly, frequent load-balancing may be required and so must have a low cost relative to that of the solution algorithm in between. This could potentially restrict the use of high 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, [34]. 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, [20]. Collectively these issues call for parallel load-balancing and, if a high quality partition is desired, a parallel partition optimisation algorithm.

For the purposes of this paper, particularly with regard to the results, we tend to concentrate on situations where the changes in load are at discrete points during the evolution of the solution. In particular this is likely to happen when either the mesh changes (as is the case for adaptive refinement) or the computational resources change. However, the techniques discussed apply equally to situations where the load changes are continuous (or at least quasi-continuous) such as the casting simulation mentioned above. In this sort of problem, a further issue is *when* to rebalance, the decision being based on a trade-off between the additional overhead for carrying out the repartitioning and resultant data migration, as against the inefficiency of continuing the simulation with an unbalanced solver. We do not address that issue here but an algorithm for determining whether or not a rebalance is likely to be profitable (and thus for deciding the frequency of repartitioning) can be found in [1].

1.1 Overview

In this paper we discuss several aspects of the load-balancing problem in the context of (re)partitioning unstructured meshes and consider several competing and complementary approaches. In Section 2 we briefly look at geometric approaches, but the main focus of this paper is graph based algorithms and so in Section 3 we discuss the use of graphs both to solve the specific load-balancing problem of how much load to move between any two processors (§3.2) and also to optimise the partition at the same time as balancing both in a local sense (§3.3) and in a global sense (§3.4). In Section 4 we briefly discuss the relevance of the existing partition and how it can sometimes be more beneficial to simply ignore it. In Section 5 we illustrate some of the issues raised with experimental results and finally in Section 6 we summarise the paper and present some conclusions.

2 Geometric repartitioning



(a) initial geometric partition (b) subsequent geometric repartition

Figure 1: Geometric repartitioning: (a) initial recursive coordinate bisection; (b) subsequent recursive coordinate bisection.

Although the primary focus of this paper is graph based optimisation approaches, geometric algorithms for partitioning can also be attractive for dynamic repartitioning. As discussed in [30], the most common algorithms for geometric partitioning involve the recursive generation of a series of cutting planes which bisect the mesh (for example Recursive Coordinate Bisection, [28]). Each pair of planes is often orthogonal to the previous one (see for example Figure 1(a) where the first bisection is vertical and the second pair of bisections are horizontal) and the algorithm can be fairly easily implemented in parallel using a parallel sorting algorithm. One distinct advantage in terms of repartitioning is that the same algorithm (using the same orientation for all the cutting places) will implicitly minimise the amount of data to migrate (as in Figure 1(b) where the shaded area represents regions of the mesh that need to migrate). The disadvantages of this approach are that it cannot guarantee the same quality partitions as graph based algorithms and, in particular, complex domain shapes can lead to subdomains which are long and thin or which are split into multiple disconnected components (both resulting in increased communications costs). However it is easy to implement and has been used with some success by Jones & Plassmann, [16].

3 Repartitioning via graph based optimisation

In this section we consider some graph based repartitioning approaches, in particular by considering the load-balancing problem (§3.2) and how this is integrated into a partition optimisation algorithm (§3.3). We also look at how the optimality of the partition (in a global sense) can be ensured by using multilevel techniques (§3.4). Finally we look at the difference between embedded and library based repartitioners and discuss their relative merits (§3.5). Firstly we establish some notation.

3.1 Notation and definitions

Let $G = G(V, E)$ be an undirected graph of vertices V with edges E which represent the data dependencies in the mesh. We assume that both vertices and edges can be weighted (with positive integer values) and that $|v|$ denotes the weight of a vertex v and similarly for edges and sets of vertices and edges. Given that the mesh needs to be distributed to P processors, define a partition π to be a mapping of V into P disjoint subdomains S_p such that $\bigcup_P S_p = V$. The partition π induces a *subdomain graph*, $G_\pi(S, L)$, on G with vertices S_p representing subdomains (the sets of vertices assigned to processor p) and edges or links $(S_p, S_q) \in L$ if there are vertices $v_1, v_2 \in V$ with $(v_1, v_2) \in E$ and $v_1 \in S_p$ and $v_2 \in S_q$. The weight of a subdomain is just the sum of the weights of the vertices in the subdomain, $|S_p| = \sum_{v \in S_p} |v|$. We denote the set of inter-subdomain or cut edges (i.e. edges cut by the partition) by E_c (note that the total weight of cut edges $|E_c| = |L|$ the total weight of edges in the subdomain graph). Finally, note that we use the words subdomain and processor more or less interchangeably: the mesh is partitioned into P subdomains; each subdomain S_p is assigned to a processor p and each processor p owns a subdomain S_p.

The definition of the graph partitioning problem is to find a partition which evenly balances the load (i.e. vertex weight) in each subdomain whilst minimising the communications cost. To evenly balance the load, the optimal subdomain weight is given by $\overline{S} := \lceil |V|/P \rceil$ (where the ceiling function $\lceil x \rceil$ returns the smallest integer greater than x) and the *imbalance*, θ, 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). Note that $\theta \geq 1$ and perfect balance is given by $\theta = 1$. As is usual, throughout this paper the communications cost will be estimated by $|E_c|$, the weight of cut edges or cut-weight (although see [30] 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. The additional objective for dynamic repartitioning is to minimise the amount of data that the underlying application will have to transfer.

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, (see §3.1) in order to determine a *balancing flow*, a flow along the edges of G_π which balances the weight of the subdomains. By keeping the flow localised in this way, vertices are not migrated between non adjacent subdomains and hence (hopefully) the partition quality is not degraded, as it almost certainly would be if vertices were migrated to non-adjacent subdomains.

This load-balancing problem, i.e. how to distribute N tasks over a network of P processors so that none have more than $\lceil N/P \rceil$, is a very important area for research in its own right with a vast range of applications. The topic is introduced in [27] and some common strategies described. Much work has been carried out on parallel or distributed algorithms and, in particular, on diffusive algorithms, e.g. [5, 10]; here we use an elegant diffusive technique developed by Hu & Blake, [15], with fast convergence. This particular method was derived to minimise the Euclidean norm of the transferred weight although it has recently been shown that all diffusion methods minimise this quantity, [6, 14]. Note that these algorithms are referred to as diffusive because the amount of load transferred between any two adjacent processors at each iteration is a linear function of the difference in their loading much in the same way that in the discretised heat equation, the amount of heat which diffuses between any two adjacent discretisation points is a linear function of the difference in temperature of those two points.

The algorithm simply involves solving the system $L\mathbf{x} = \mathbf{b}$ where L is the Laplacian of the subdomain graph:

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

where degree(S_p) is the degree of (or number of edges incident on) the vertex S_p and where $b_p = |S_p| - \overline{S}$ (the weight of S_p less the optimal subdomain weight). The weight to be transferred across edge (S_p, S_q) is then given by $x_p - x_q$. The algorithm is employed as suggested in [15], solving iteratively with a conjugate gradient solver; it is solved for a real solution and the (integer) flow is determined by rounding. However, whilst this is an algorithm which is easily parallelised, we have found it more cost effective, for the numbers of processors which we used for testing (up to 128), to broadcast a copy of the subdomain graph around the parallel machine and duplicate the (serial) solution of the problem on every processor. Clearly for large numbers of processors this will not scale and so we have also implemented a fully parallel version (although it is not used for the tests here). Note finally that the Laplacian of any undirected graph contains a zero eigenvalue with the corresponding eigenvector $[1, 1, \ldots, 1]$ and the solution iterates are orthogonalised against this, [15]. If any other singularities are detected (for example if the graph is disconnected) the software will switch to another method, an intuitive and entirely localised parallel load-balancing algorithm due to Song, [29].

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

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

3.3 Integrating balancing flow & optimisation

In the companion paper to this, [30], three iterative optimisation techniques are outlined for refining in parallel the quality of a partition (essentially by treating each inter-subdomain interface as a separate problem and then using a bisection optimisation algorithm such as that of Kernighan & Lin, [19], within each interface region). Such algorithms are not difficult to combine with a balancing flow by, at each iteration, simply satisfying (as far as possible) any required flow of vertex weight prior to swapping vertices back and forth for refinement purposes, [31]. Indeed if the load-balancing algorithm is iterative (as above), the two algorithms can be interleaved. This sort of combination of load-balancing and refinement forms the basis for several parallel optimisation implementations (e.g. [7, 25, 32]).

Schloegel *et al.* have also extended the technique for dynamic repartitioning purposes by choosing vertices for migration not just on the basis of minimising cut-weight, but also to minimise data migration, [25]. Although they describe it in terms of ‘dirty’ and ‘clean’ vertices, it is perhaps easiest to understand in abstract terms by thinking of each vertex having an additional edge between it and the processor it started on. So-called *dirty* vertices, which have migrated away from their original ‘home’ processor, then result in an additional contribution to the cost function (because of the cut edge). In this sense it is similar to other graph manipulation techniques such as those found in [13, 23, 33, 34] where additional vertices are included in the graph to represent processors. Schloegel *et al.* experiment with the trade-off between minimising data migration and optimisation (essentially by changing the weighting of the additional vertex-processor edges relative to that of the original graph edges).

3.4 Using multilevel techniques

The combination of flow & optimisation algorithms can be an effective solution for the dynamic load-balancing and repartitioning of unstructured meshes, however repeated application of these localised algorithms can, after some time, result in serious degradation of the partition quality. In recent years it has been recognised that an effective way of both speeding up partition refinement and, perhaps more importantly giving it a global perspective is to use multilevel techniques. The idea is to group vertices together to form *clusters*, use the clusters to define a new graph and recursively iterate this procedure to create a series of increasingly coarse graphs until the size of the coarsest graph falls below some threshold. A fast and possibly crude initial partition of the coarsest graph is calculated and then successively interpolated onto and optimised on each of the graphs in reverse order. This sequence of contraction followed by repeated expansion/refinement loops is known as the multilevel paradigm and has been successfully developed as a strategy for overcoming the localised nature of the Kernighan-Lin, [19], (and other) algorithms. The multilevel idea was first proposed by Barnard & Simon, [2], as a method of speeding up spectral bisection and improved by both Hendrickson & Leland, [12] and Bui & Jones, [4], who generalised it to encompass local refinement algorithms. The implementation of a parallel graph contraction algorithm is fully described in [32] and some sample results in §5.1 demonstrate the benefits of the technique.

One caveat about using the multilevel ideas is that to simply contract the graph down to P vertices, one for each processor (as in [30, 31]) can be counter-productive in terms of finding a new partition which minimises the amount of data migration. The reason for this is simple; each vertex in the coarsest graphs may represent hundreds or even thousands of vertices in the original graph and so moving them from subdomain to subdomain may give rise to very high data migration in the application. We address this issue by experimentally looking for a suitable contraction threshold in §5.2.

3.5 Embedded and library based repartitioners

It should be mentioned at this point that there are several existing implementations of combined load-balancing and optimisation techniques, typically 'in-house' subroutines embedded within parallel adaptive mesh codes (e.g. [9, 20]). In general such repartitioning subroutines never actually construct a graph, since they normally work directly with the mesh data structures and thus have the advantage of no memory overhead. Nevertheless the techniques used are identical in essence to graph based optimisation techniques and it is instructive to abstract the algorithms using graph partitioning terminology. In this sense they can be seen as equivalent to single-level graph partitioning techniques (even though the graph is never physically realised).

In contrast there are at least two public domain parallel multilevel (re)partitioning libraries, JOSTLE, [31], and ParMETIS, [17]. These 'plug and play' solutions have the disadvantage that the relevant mesh data must be copied and passed into subroutines as arrays with an additional memory overhead, however, for repeated KL type optimisation, where a graph vertex may be swapped back and forth several times between processors, their dedicated light-weight graph structures may be more efficient than moving mesh nodes & elements plus all the attendant solution components (although the final repartitioning solution must then be realised within the mesh afterwards). They can also use recent ideas more easily (such as the multilevel strategy) and save development time. Clearly however, the choice of which to use is very application dependent (see for example [21, 24] for some typical costs) and possibly the best long-term strategy would be to use a library for development purposes and then, if appropriate and the additional development time is warranted, switch to a dedicated special-purpose repartitioner for production versions.

4 Scratch remapping

The previous sections of the paper have an implicit assumption that the mesh or computational load has not changed 'too much' since the previous (re)partitioning and that it therefore makes sense to repartition by diffusing load from overloaded processors to underloaded ones. In cases where the load has changed dramatically, empirical evidence suggests that there are in fact some advantages to simply ignoring any existing distribution and partitioning from scratch. The reasons behind this are that the diffusion of load from extremely overloaded processors can significantly distort the partition quality (essentially the optimisation process is unable to compensate for large amounts of load being transferred across the system). In addition, by utilising heuristics for mapping subdomains to processors (based on the existing placement of data), the data migration can even be less when partitioning from scratch as compared with diffusive repartitioning. This area has been investigated by Biswas & Oliker, [3], who devised appropriate mapping heuristics and is often referred to as *scratch-remapping*.

During this work, Biswas & Oliker tended to use partitioners as 'black box' software, carrying out a partition from scratch and then using their remapping techniques to assign new subdomains to processors. However, in an extension of the technique, Schloegel *et al.*, [26], modified the strategy to use the same remapping heuristics on the coarsest graphs of the multilevel process (rather than the final partition) and reported that it minimised the data migration still further over the 'black-box' version.

5 Experimental results

The software tool written at Greenwich and which we use to illustrate some of the issues is known as JOSTLE[1]. 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 a single-level algorithm as suggested in §3.3 (and fully described in [32]) to balance and optimise the partition. The multilevel-dynamic, JOSTLE-MD, uses the same procedure but additionally uses graph contraction (§3.4) to improve the partition quality. The static version, JOSTLE-MS, carries out graph contraction on the unpartitioned graph, and a serial refinement algorithm to optimise the partition on each of the multilevel graphs, [32].

The test meshes have been taken from an example contained in the DIME (distributed irregular mesh environment) software package, [35]. 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 [36]. 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.

5.1 Comparison results

In order to demonstrate the quality of the partitions we have compared the method with three popular partitioning schemes, METIS, GREEDY and Multilevel Recursive Spectral Bisection (MRSB). Of the three METIS, [18], is the most similar to JOSTLE, employing multilevel iterative optimisation. The GREEDY algorithm, [8], 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 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. The experiments are run in serial to compare run-times but the JOSTLE configurations and METIS can all be run in parallel and, at least in the case of JOSTLE, achieve the same partition qualities.

For the two dynamic configurations, the initial mesh is partitioned with the static version – JOSTLE-MS. Subsequently at each refinement, the existing partition is in-

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

terpolated onto the new mesh using the techniques described in [34] (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	$\lvert E_c \rvert$	$t(s)$	M %	$\lvert E_c \rvert$	$t(s)$	M %	$\lvert E_c \rvert$	$t(s)$	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 the multilevel approach, 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.

5.2 Effect of the multilevel techniques

To further compare the JOSTLE-D and JOSTLE-MD configurations, we can look at how the results compare as the contraction threshold changes. The contraction threshold determines at what level the graph contraction procedure terminates and thus JOSTLE-D can be seen as the same configuration as JOSTLE-MD only with a very large threshold (so that the contraction never starts).

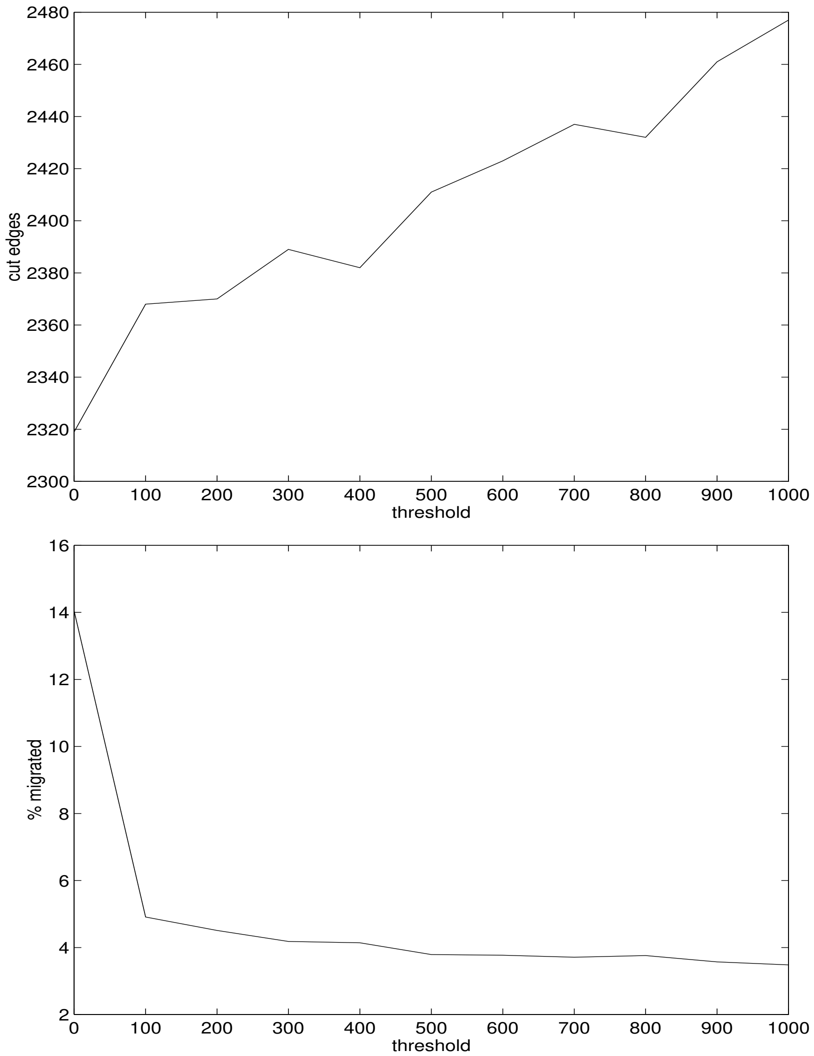


Figure 2: The effects of varying the contraction threshold on the cut-edge weight (top) and data migrated (bottom)

Figure 2 shows the effects of varying the contraction threshold for the final mesh of the adaptive series given a reasonably good fixed initial partition. Here the threshold refers to the number of graph vertices per processor below which the contraction process terminates. As can be seen (despite the noise in the results) the quality of the partition (as measured by the cut-edge weight) gradually falls off as the threshold increases (i.e. as the partitioner tends towards the JOSTLE-D configuration). Again, this is to be expected as the multilevel strategy tends to give a more global quality to the optimisation. Perhaps more interesting, however, is the way the volume of data migrated drops off very rapidly as the threshold increases. In fact the graph is even more exponential than shown as the intervals chosen for the threshold are multiples of 100. This suggests that, in terms of the data migrated, it is of no great benefit to choose a high threshold and that reasonably good performance can be achieved with a relatively low setting. It is for this reason that we have chosen a default setting of 20 for JOSTLE-MD as it is felt that this gives a good balance between high partition quality and low data migration.

5.3 Parallel timings

		JOSTLE-D			JOSTLE-MD		
V	E	$P = 16$	$P = 32$	$P = 64$	$P = 16$	$P = 32$	$P = 64$
31172	46309	0.06	0.06	0.07	0.35	0.26	0.25
40851	60753	0.07	0.07	0.08	0.40	0.32	0.32
53338	79415	0.06	0.08	0.11	0.97	0.30	0.32
69813	104034	0.10	0.09	0.13	0.48	0.32	0.33
88743	132329	0.13	0.10	0.09	0.49	0.40	0.38
115110	171782	0.11	0.11	0.11	0.61	0.44	0.39
146014	218014	0.16	0.13	0.13	0.75	0.56	0.55
185761	277510	0.21	0.15	0.16	0.87	0.63	0.55
224843	336024	0.19	0.18	0.14	0.95	0.67	0.59

Table 2: Parallel timings for the JOSTLE-D & JOSTLE-MD configurations

As discussed in [30], achieving high parallel performance for parallel partitioning codes such as JOSTLE is not as easy as, say, a typical CFD or CM code. However, Table 2 gives parallel timings in seconds for the JOSTLE-D and JOSTLE-MD configurations 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 extremely low overheads (always less than a second) for the parallel partitioning and, since in this case each initial partition is of high quality, confirm the conclusions in [30] that the partitioning time is strongly dependent on the quality of the initial partition.

6 Summary

We have discussed the load-balancing issues arising for parallel mesh based computational mechanics codes for which the processor loading changes during the run. In particular we have focussed on different ways of using a graph both to solve the load-balancing problems and the optimisation problem, both locally and globally. We have also briefly discussed whether repartitioning is always valid; sometimes, when there have been very dramatic load changes, it is better to simply repartition from scratch. We have looked at some sample illustrative results and seen that, for some adaptive refinement situations, 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. We have also seen that there is a certain amount of trade-off between partition quality and volume of data migration.

Finally, the dynamic repartitioning area is still very much a field of active research and in the near future we hope to address the *very* challenging problems which arise in a dynamic multiphase problem (similar to [33] only with the additional complication of changing processor loading).

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

References

[1] V. Aravinthan, S. P. Johnson, K. McManus, C. Walshaw, and M. Cross. Dynamic Load Balancing for Multi-Physical Modelling using Unstructured Meshes. In C.-H. Lai *et al.*, editor, *Proc. 11th Intl. Conf. Domain Decomposition Methods, Greenwich, UK, 1998*, pages 380–387. DDM.org, http://www.ddm.org, 1999.

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

[3] R. Biswas and L. Oliker. Experiments with Repartitioning and Load Balancing Adaptive Meshes. Tech. Rep. NAS-97-021, NASA Ames, Moffet Field, CA, 1997.

[4] T. N. Bui and C. Jones. A Heuristic for Reducing Fill-In in Sparse Matrix Factorization. In R. F. Sincovec *et al.*, editor, *Parallel Processing for Scientific Computing*, pages 445–452. SIAM, 1993.

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

[6] R. Diekmann, A. Frommer, and B. Monien. Efficient Schemes for Nearest Neighbor Load Balancing. *Parallel Comput.*, 25(7):789–812, 1999.

[7] R. Diekmann, B. Monien, and R. Preis. Load Balancing Strategies for Distributed Memory Machines. In B. H. V. Topping, editor, *Parallel & Distributed Processing for Computational Mechanics: Systems and Tools*, pages 124–157. Saxe-Coburg Publications, Edinburgh, 1999. (Proc. Parallel & Distributed Computing for Computational Mechanics, Lochinver, Scotland, 1997).

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

[9] J. E. Flaherty, R. M. Loy, C. Özturan, M. S. Shephard, B. K. Szymanski, J. D. Teresco, and L. H. Ziantz. Parallel structures and dynamic load balancing for adaptive finite element computation. *Appl. Numer. Math.*, 26:241–263, 1998.

[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 Univ., New Haven, CT 06520, USA, 1995.

[11] B. Hendrickson and K. Devine. Dynamic Load Balancing in Computational Mechanics. (to appear in *Comput. Meth. Appl. Mech. Engrg.*).

[12] B. Hendrickson and R. Leland. A Multilevel Algorithm for Partitioning Graphs. In S. Karin, editor, *Proc. Supercomputing '95*, New York, NY 10036, 1995. ACM Press.

[13] B. Hendrickson, R. Leland, and R. Van Driessche. Enhancing Data Locality by Using Terminal Propagation. In *Proc. 29th Hawaii Int. Conf. System Science*, 1996.

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

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

[16] M. T. Jones and P. E. Plassmann. Parallel Algorithms for the Adaptive Refinement and Partitioning of Unstructured Meshes. In *Proc. Scalable High Performance Comput. Conf. '94*, pages 478–485. IEEE, 1994.

[17] 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.

[18] G. Karypis and V. Kumar. Multilevel k-way Partitioning Scheme for Irregular Graphs. *J. Par. Dist. Comput.*, 48(1):96–129, 1998.

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

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

[21] G. Lonsdale, A. Basermann, J. Fingberg, T. Coupez, H. Digonnet, J. Clinckemaillie, G. Thierry, P. Dannaux, B. Maerten, D. Roose, R. Ducloux, and C. Walshaw. DRAMA: Dynamic Re-Allocation of Meshes for parallel Finite Element Applications. In B. H. V. Topping, editor, *Developments in Computational Mechanics with High Performance Computing*, pages 61–66. Civil-Comp Press, Edinburgh, 1999. (Proc. Parallel & Distributed Computing for Computational Mechanics, Weimar, Germany, 1999).

[22] 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. (Proc. Parallel CFD'95, Pasadena, 1995).

[23] F. Pellegrini and J. Roman. Experimental Analysis of the Dual Recursive Bipartitioning Algorithm for Static Mapping. TR 1038-96, LaBRI, URA CNRS 1304, Univ. Bordeaux I, 351, cours de la Libération, 33405 TALENCE, France, 1996.

[24] F. Schlimbach. Optimising Subdomain Aspect Ratios for Parallel Load Balancing. PhD thesis, Computing and Mathematical Sciences, Univ. Greenwich, London SE10 9LS, UK, 1999.

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

[26] K. Schloegel, G. Karypis, and V. Kumar. Wavefront Diffusion and LMSR: Algorithms for Dynamic Repartitioning of Adaptive Meshes. TR 98-034, Dept. Comp. Sci., Univ. Minnesota, Minneapolis, MN 55455, 1998.

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

[28] H. D. Simon. Partitioning of Unstructured Problems for Parallel Processing. *Computing Systems Engrg.*, 2:135–148, 1991.

[29] J. Song. A partially asynchronous and iterative algorithm for distributed load balancing. *Parallel Comput.*, 20(6):853–868, 1994.

[30] C. Walshaw and M. Cross. Parallel Mesh Partitioning on Distributed Memory Systems. In B. H. V. Topping, editor, *Parallel & Distributed Processing for Computational Mechanics*. Saxe-Coburg Publications, Edinburgh, 1999. (Proc. Euro-CM-Par, Weimar, Germany, 1999).

[31] C. Walshaw and M. Cross. Parallel Optimisation Algorithms for Multilevel Mesh Partitioning. To appear in *Parallel Comput.* (originally published as Univ. Greenwich Tech. Rep. 99/IM/44), 1999.

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

[33] C. Walshaw, M. Cross, and K. McManus. Multiphase Mesh Partitioning. Tech. Rep. 99/IM/51, Univ. Greenwich, London SE10 9LS, UK, November 1999.

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

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

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