Load-balancing for parallel adaptive unstructured grids

C. Walshaw
M. Cross

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

Abstract

A parallel method for the dynamic partitioning of unstructured meshes is outlined. The method includes diffusive load-balancing techniques and an iterative optimisation technique known as relative gain optimisation which both balances the workload and attempts to minimise the interprocessor communications overhead. It can also optionally include a multilevel strategy. 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 need for mesh partitioning arises naturally in many finite element (FE) and finite volume (FV) applications. Meshes composed of elements such as triangles or tetrahedra are often better suited than regularly structured grids for representing completely general geometries and resolving wide variations in behaviour via variable mesh densities. Meanwhile, the modelling of complex behaviour patterns means that the problems are often too large to fit onto serial computers, either because of memory limitations or computational demands, or both. Distributing the mesh across a parallel computer so that the computational load is evenly balanced and the data locality maximised is known as mesh partitioning. It is well known that this problem is NP-complete, so in recent years much attention has been focused on developing suit-

able heuristics, and some powerful methods, many based on a graph corresponding to the communication requirements of the mesh, have been devised.

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, [7]) 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.

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, [10]. In fact, not only is the load-balancing likely to be unnecessarily computationally expensive if it fails to use this information, but also (as can be seen in Section 4) 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. Collectively these issues call for parallel load-balancing and, if a high quality partition is desired, a parallel optimisation algorithm.

In this paper we outline 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 (outlined 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 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.

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.

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 Optimisation

In this section we outline a parallel iterative algorithm for load-balancing and optimising unstructured mesh partitions. The method is based on the concept of *relative gain* and is described more fully in [9].

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 (the graph of connections between subdomains), in order to determine a *balancing flow*, a flow along the edges of G_π which balances the weight of the subdomains. By keeping the flow localised in this way, vertices are not migrated between non adjacent subdomains and hence (hopefully) the partition quality is not degraded (since a vertex migrating to a subdomain to which it is not adjacent is almost certain to have a negative gain).

Much work has been carried out on parallel or distributed algorithms and, in particular, on diffusive algorithms, e.g. [3], and we use an elegant technique developed by Hu & Blake, [5], which converges faster than diffusive methods, minimises the Euclidean norm of the transferred weight and simply involves solving the an $O(P)$ linear system. This algorithm (or, in principle, any other distributed load-balancing algorithm) is used to determine *how much* weight to transfer across edges of the subdomain graph and the optimisation technique below is then used to decide *which* vertices to move.

Relative gain optimisation

A key concept in many graph partition optimisation algorithms is the idea of gain. Loosely, the gain $g(v, q)$ of a vertex v in subdomain S_p can be calculated for every other subdomain, S_q, $q \neq p$, and expresses some 'estimate' of how much the partition would be 'improved' were v to migrate to S_q. The gain is usually directly related to some cost function which measures the quality of the partition and which we aim to minimise. In this paper, as is typical, the cost function used is simply the total weight of cut edges, $|E_c|$, and then the gain expresses the change in $|E_c|$, but see [9] for more discussion on this point.

Having determined the required flow across the edges of the subdomain graph we need to migrate vertices between 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 address this issue with an optimisation algorithm fully described in [9] and known as *relative gain* optimisation. Essentially, 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.

3 Graph contraction

The optimisation 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 contraction. The idea is to group vertices together to form *clusters*, use the clusters to define a new graph, recursively iterate this procedure to produce a series of graphs $G_1, G_2, \ldots, G_l$ until the graph size falls below some threshold and then successively optimise these reduced size graphs. The optimisation is carried out on each graph using the algorithm outlined in Section 2 and we enhance the technique by using a multilevel balancing schedule. This idea, first described in [8], allows a given amount of imbalance in each of the reduced graphs (the amount is given by the balancing schedule and decreases as the procedure progressively optimises the graphs $G_l, G_{l-1}, \ldots, G_1$) with the intention that the leeway given by the imbalance will allow the optimisation technique to find a better partition. The implementation of the parallel graph contraction algorithm is fully described in [9].

4 Experimental results

The software tool written at Greenwich and which we have used to test the optimisation and graph contraction algorithms is known as JOSTLE. For the purposes of this paper it can be run in three configurations, dynamic (JOSTLE-D), multilevel-dynamic (JOSTLE-MD) and multilevel-static (JOSTLE-MS). The dynamic configuration, JOSTLE-D, starts from an existing partition and uses the algorithm outlined in Section 2 to balance and optimise the partition. The multilevel-dynamic, JOSTLE-MD, uses graph contraction (Section 3) on the existing partition to a given threshold (in the case of the results given here the threshold is 20 vertices per processor) while the static version, JOSTLE-MS, carries out graph contraction on the graph partitioned with a simple block based partition (the first V/P vertices on processor 0, etc.). Both multilevel configurations then use the optimisation algorithm from Section 2 to successively balance and optimise the partition on each graph.

The test meshes have been taken from an example contained in the DIME (distributed irregular mesh environment) software package, [11], available by anonymous ftp. 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 [12]. 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.

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 contraction technique and iterative optimisation. The version used here is `kmetis`, available by anonymous ftp [6]. The GREEDY algorithm, [4], is fast but not particularly good at minimising $|E_c|$, while 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 [10] (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	898	0.45	0.84	1568	0.48	1.66	2579	0.55	3.58
JOSTLE-MD	798	2.54	2.17	1399	2.73	3.66	2314	3.10	5.93
JOSTLE-MS	878	2.48	94.69	1506	2.76	98.07	2371	3.29	98.17
METIS	890	4.69	94.29	1519	4.83	95.92	2398	5.15	97.93
MRSB	939	13.02	83.54	1577	16.28	90.01	2520	19.57	95.07
GREEDY	1816	0.63	81.62	2897	0.69	90.64	4300	0.84	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 contraction, 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 and JOSTLE-MS are considerably faster than 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.

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 1 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.

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.

Table 2 gives serial timings in seconds ($t_s(s)$), parallel timings in seconds ($t_p(s)$) and speedup (t_s/t_p) results for the JOSTLE-MD configuration on the 512 node Cray T3E at HLRS, the High Performance Computer Centre at the University of Stuttgart. These demonstrate good speedups for this sort of code and more importantly, very low overheads (always less than a second) for the parallel partitioning. The results for JOSTLE-D are similar (see [9] for some examples) only with shorter serial and

	$P = 16$			$P = 32$			$P = 64$		
V	$t_s(s)$	$t_p(s)$	t_s/t_p	$t_s(s)$	$t_p(s)$	t_s/t_p	$t_s(s)$	$t_p(s)$	t_s/t_p
31172	0.71	0.28	2.54	1.02	0.27	3.78	1.80	0.37	4.86
40851	0.88	0.26	3.38	1.26	0.27	4.67	2.00	0.36	5.56
53338	1.16	0.32	3.62	1.56	0.33	4.73	2.41	0.39	6.18
69813	1.46	0.34	4.29	1.88	0.34	5.53	2.68	0.35	7.66
88743	1.76	0.36	4.89	2.26	0.37	6.11	3.16	0.39	8.10
115110	2.27	0.45	5.04	2.87	0.46	6.24	3.83	0.42	9.12
146014	2.83	0.52	5.44	3.45	0.46	7.50	4.53	0.46	9.85
185761	3.55	0.62	5.73	4.32	0.54	8.00	5.41	0.46	11.76
224843	4.29	0.70	6.13	5.03	0.57	8.82	6.06	0.52	11.65

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

parallel runtimes and (usually) slightly better speedups.

5 Conclusion

We have outlined a new method for optimising and load-balancing graph partitions with a specific focus on its application to the dynamic mapping of adaptive 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. In D. Emerson *et al*, editor, *Parallel Computational Fluid Dynamics: Recent Developments and Advances Using Parallel Computers*. Elsevier, Amsterdam, 1998. (Proc. Parallel CFD'97, Manchester, 1997; 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] C. Farhat. A Simple and Efficient Automatic FEM Domain Decomposer. *Comp. & Struct.*, 28(5):579–602, 1988.

[5] 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.

[6] 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.

[7] 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).

[8] C. Walshaw and M. Cross. Mesh Partitioning: a Multilevel Balancing and Refinement Algorithm. Tech. Rep. 98/IM/35, University of Greenwich, London SE18 6PF, UK, March 1998.

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

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

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

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

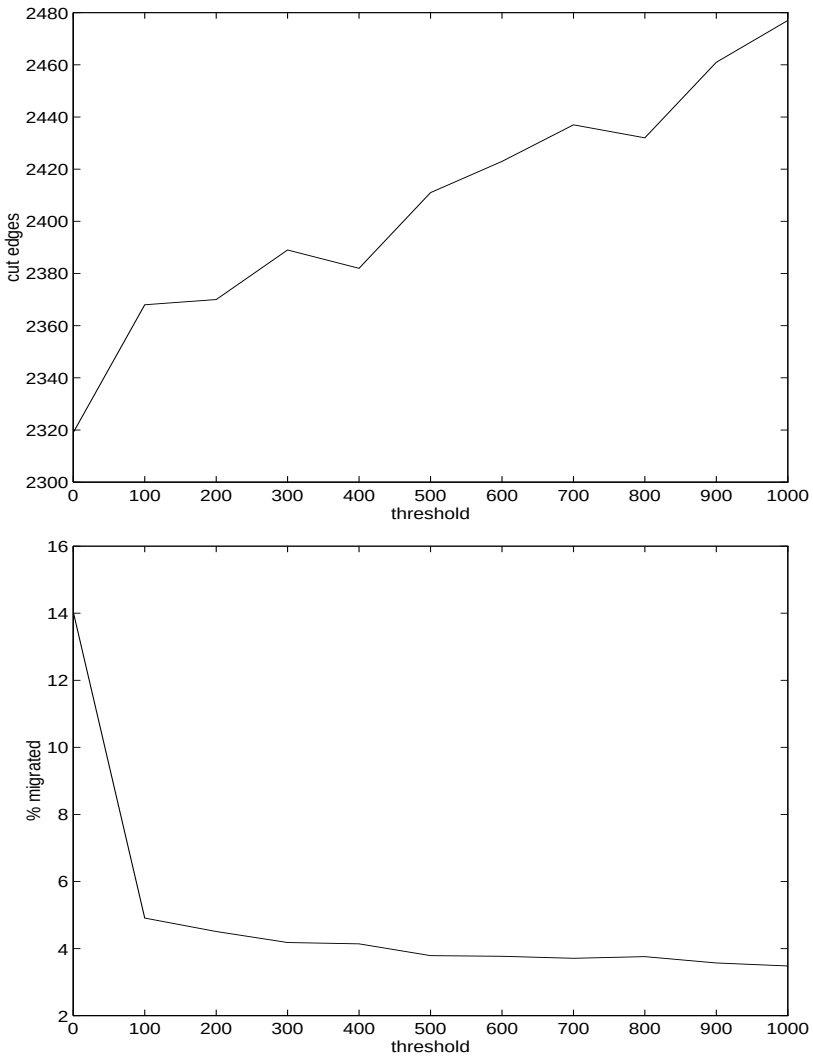

Figure 1: The effects of varying the contraction threshold on the cut-edge weight (top) and data migrated (bottom)