Dynamic load-balancing for parallel adaptive unstructured meshes

C. Walshaw, M. Cross and M. G. Everett*

In: Parallel Processing for Scientific Computing, M. Heath *et al.*, eds., SIAM, 1997.

Abstract

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

Key words. graph-partitioning, adaptive unstructured meshes, load-balancing, parallel scientific computation.

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. [1, 9, 14].

An increasingly important area for mesh partitioning arises from problems in which the computational load varies throughout the evolution of the solution. For example, time-dependent unstructured mesh codes which use adaptive refinement can give rise to a series of meshes in which the position and density of the data points varies dramatically over the course of an integration and which may need to be frequently repartitioned for maximum parallel efficiency. This dynamic partitioning problem has not been nearly as thoroughly studied as the static problem but related work can be found in [3, 4, 5, 11, 15, 18].

The dynamic evolution of load has three major influences on possible partitioning techniques; cost, reuse and parallelism. Firstly, the unstructured mesh may be modified every few time-steps and so the load-balancing must have a low cost relative to that of the

*Parallel Processing Group, Centre for Numerical Modelling & Process Analysis, University of Greenwich, London, SE18 6PF, UK. E-mail: C.Walshaw@gre.ac.uk

solution algorithm in between remeshing. This may seem to restrict us to computationally cheap algorithms but fortunately, if the mesh has not changed too much, it is a simple matter to interpolate the existing partition from the old mesh to the new and use this as the starting point for repartitioning, [18]. 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, [11]. 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 a parallel optimisation technique (Section 2) which incorporates a distributed load-balancing algorithm and which provides an extremely fast solution to the problem of load-balancing adaptive 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.

2 Optimisation

In this section we present a parallel iterative algorithm for load-balancing and optimising unstructured mesh partitions in order to share the workload equally between all subdomains and to carry out local refinement.

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 and similarly for edges and sets of vertices & edges. We define $\mathcal{P} : V \to P$ to be a partition of V and denote the resulting subdomains by S_p, for $p \in P$. The optimal subdomain weight is given by $W := \lceil |V|/P \rceil$. We denote the set of cut (or inter-subdomain) edges by E_c and the border of each subdomain, B_p, is defined as the set of vertices in S_p which have an edge in E_c. We shall use the notation $\leftrightarrow$ to mean 'is adjacent to', for example, for $u, v \in V$, $u \leftrightarrow v$ if $\exists (u, v) \in E$.

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 $\mathcal{P}$ 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 below).

The gain and preference functions. A key concept in the method 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 [12], Vanderstraeten *et al.* demonstrate that it can be extremely effective to vary the cost function based on a knowledge of the solver. Meanwhile, in [17] 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, $|E_c|$.

Localisation and the subdomain graph. An important aim for any parallel algorithm is to keep communication as localised as possible to avoid contention and expensive global operations; this issue becomes increasingly important as machine sizes grow. Throughout the optimisation algorithm we localise the vertex migration with respect to the partition $\mathcal{P}$ by only requiring subdomains to migrate vertices to neighbouring subdomains. In this way the partition $\mathcal{P}$ induces a subdomain graph on G, an undirected graph of P subdomains (vertices) and C connections (edges).

Load-balancing. The load-balancing problem, i.e. how to distribute V tasks over a network of P processors so that none have more than $\lceil V/P \rceil$, is a very important area for research in its own right with a vast range of applications. Here we use an elegant technique recently developed by Hu & Blake, [8], related to, but with faster convergence than the commonly used diffusive methods, e.g. [2], and which minimises the Euclidean norm of the transferred weight. The algorithm simply involves solving the system $L\mathbf{x} = \mathbf{b}$ where L is the Laplacian of the subdomain graph, ($L_{pp} =$ degree(S_p); $L_{pq} = -1$ if $S_p \leftrightarrow S_q$, $L_{pq} = 0$ otherwise) $b_p = |S_p| - W$ and the weight to be transferred across edge (S_p, S_q) is then given by $x_p - x_q$. Note that this method is closely related to diffusive algorithms except that the diffusion coefficients are not fixed but determined at each iteration by a conjugate gradient search.

This algorithm (or, in principle, any other distributed load-balancing algorithm) defines how much weight to transfer across edges of the subdomain graph and we then use the optimisation mechanism to decide which vertices to move.

The parallel optimisation mechanism. Having determined the required flow across the edges of the subdomain graph 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 and this may lead to an *increase* the cost. We have previously addressed this problem by using a Kernighan-Lin type algorithm run in the boundary regions alone, [15, 16], and it has also been addressed by Karypis & Kumar, [10], who colour the graph vertices to avoid such collisions. Here we introduce a new strategy which uses the concept of the *relative gain*.

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 load to migrate. This formula, together with justification for its derivation is presented in full in [13]. Essentially though, 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 suppose that f_{pq} represents the required flow from p to q and g_{pq} the total weight of vertices in B_{pq} with gain > 0. If $d = \max(g_{pq} - f_{pq} + g_{qp} - f_{qp}, 0)$, which represents approximately the weight of vertices with positive gain after all flow has been satisfied, then load to be migrated from p to q, is given by $a_{pq} = f_{pq} + d/2$. Essentially this firstly

allows the flow to be satisfied and secondly, since the amount to be transferred from q to p is given by $a_{qp} = f_{qp} + d/2$, exchanges an equal amount of weight from S_p and S_q (based on the weight of vertices with positive gain) in order to both load-balance and optimise the cost function.

To determine which vertices to migrate we define the relative gain 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 $g(v,q) - \sum_{u \in \Gamma_q(v)} g(u,p)/\|\Gamma_q(v)\|$ (where $\|\Gamma_q(v)\|$ represents the number of vertices in $\Gamma_q(v)$). The relative gain then gives an indication of which are the best vertices to move in order to avoid collisions and 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.

3 Graph reduction

The algorithm described above 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 perception 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, [1, 7], and in an adaptive way analogous to dynamic refinement techniques, [18]. Several algorithms for carrying out the reduction can be found in [9].

Reduction. 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, [7] and improved by Karypis & Kumar in [9]. 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.

Parallel matching. 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, [9], that it can be beneficial to the optimisation to collapse the most heavily weighted edges and our matching algorithm uses this heuristic. For the parallel version we use more or less the same procedure; each processor visiting in parallel the vertices that it owns. We modify the matching algorithm, however, by always matching with a local vertex in preference to a vertex owned by another processor. The local matching can then take place entirely in parallel but usually leaves a few boundary vertices who have no unmatched local neighbours but possibly some unmatched non-local neighbours.

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. Vertices which are matched across interprocessor boundaries are migrated to one of the two owning processors and then the construction of the reduced graph can take place entirely in parallel. The algorithm is fully described in [13].

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 unpartitioned graph, employs the greedy algorithm, [6], to generate an initial partition of the coarsest graph and, finally, together with the algorithm described in Section 2, uses an optimisation technique, fully described in [14], which attempts to minimise the 'surface energy' of the subdomains.

In order to demonstrate the quality of the partitions we have compared the method with 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, freely available by anonymous ftp from `ftp.cs.umn.edu` in `dept/users/kumar/metis/metis-2.0.5.tar.gz`. The GREEDY algorithm, [6], is actually performed as part of the JOSTLE code 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, [1]. The MRSB code was made available to us by one of its authors, Horst Simon, and run unchanged with a contraction thresholds of 100.

The test meshes have been taken from an example contained in the DIME (distributed irregular mesh environment) software package, [19], 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 [20].

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

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 of JOSTLE-MS. Subsequently at each refinement, the existing partition is interpolated onto the new mesh using the techniques described in [18] (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	917	0.66	0.67	1397	0.81	1.70	2433	1.13	4.31
JOSTLE-MD	811	2.62	5.79	1376	2.89	6.49	2310	3.40	9.55
JOSTLE-MS	838	3.81	94.19	1429	4.24	92.80	2300	4.92	98.98
METIS	867	4.39	94.36	1463	4.49	95.94	2301	4.85	97.95
MRSB	890	51.62	83.54	1494	66.40	90.01	2391	81.94	95.07
GREEDY	1719	0.67	81.62	2746	0.73	90.64	4071	0.91	94.42

TABLE 1

Average results over the 10 meshes

Table 1 compares the six different partitioning methods with the results averaged over the 10 meshes for $P = 16$, 32 and 64. 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. In general, JOSTLE-D (without the benefit of graph reduction) provides slightly lower quality partitions than the other two configurations, although for $P = 32$ it outperforms all the other algorithms except JOSTLE-MD. However, our experience suggests that this is not normally the case. In terms of execution time, JOSTLE-D is roughly as fast as GREEDY, and faster than any of the multilevel algorithms. JOSTLE-MD, however, is considerably faster than JOSTLE-MS and METIS; 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, as fast as 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.

		$P = 16$			$P = 32$		
V	E	$t_s(s)$	$t_p(s)$	speed up	$t_s(s)$	$t_p(s)$	speed up
31172	46309	0.39	0.10	3.9	0.66	0.16	4.1
40851	60753	0.57	0.13	4.3	0.65	0.12	5.4
53338	79415	0.81	0.18	4.5	0.88	0.14	6.3
69813	104034	0.93	0.19	4.9	0.93	0.14	6.6
88743	132329	1.08	0.21	5.1	1.18	0.16	7.3
115110	171782	1.36	0.24	5.7	1.65	0.21	7.9
146014	218014	1.93	0.32	6.0	1.95	0.22	8.9
185761	277510	2.22	0.35	6.3	2.46	0.26	9.5
224843	336024	3.85	0.41	9.4	3.01	0.31	9.7

TABLE 2

Serial and parallel timings for the JOSTLE-D configuration

		$P = 16$			$P = 32$		
V	E	$t_s(s)$	$t_p(s)$	speed up	$t_s(s)$	$t_p(s)$	speed up
31172	46309	2.21	0.56	3.9	2.55	0.48	5.3
40851	60753	2.77	0.70	4.0	3.18	0.56	5.7
53338	79415	3.54	0.73	4.8	4.05	0.64	6.3
69813	104034	4.58	0.90	5.1	4.91	0.67	7.3
88743	132329	5.58	1.04	5.3	6.25	0.80	7.8
115110	171782	7.14	1.17	6.1	7.83	0.91	8.6
146014	218014		1.47			1.10	
185761	277510		1.63			1.24	
224843	336024		2.23			1.39	

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 Edinburgh Cray T3D. 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 (no more than a couple of seconds) for the parallel partitioning. Some of the larger meshes would not fit in the memory for serial partitioning with JOSTLE-MD and so no serial timings are given. 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 the Edinburgh Parallel Computer Centre and the Engineering and Physical Sciences Research Council for access to the Cray T3D.

References

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

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

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

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

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

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

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

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

[9] 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.

[10] ——, *Parallel multilevel k-way partitioning scheme for irregular graphs*, TR 96-036, Computer Science Department, University of Minnesota, Minneapolis, MN 55455, 1996.

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

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

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

[14] ——, *A Localised Algorithm for Optimising Unstructured Mesh Partitions*, Int. J. Supercomputer Appl., 9 (1995), pp. 280–295.

[15] ——, *Dynamic mesh partitioning: a unified optimisation and load-balancing algorithm*, Tech. Rep. 95/IM/06, University of Greenwich, London SE18 6PF, UK, 1995.

[16] ——, *Parallel Partitioning of Unstructured Meshes*, in Proc. Parallel CFD '96, 1996. (in press).

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

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

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

[20] ——, *Performance of dynamic load balancing algorithms for unstructured mesh calculations*, Concurrency: Practice & Experience, 3 (1991), pp. 457–481.