Parallel Unstructured Mesh Partitioning

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

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 readily be 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. [FS93]. Closely related to this graph partitioning problem is the problem of optimising existing mesh partitions and in this paper we discuss the partition optimisation problem and its bearing on the graph partitioning problem.

In particular, the algorithms outlined in this paper are designed to address the three problems that arise in partitioning of unstructured finite element and finite volume meshes. Specifically the:

(i) **static partitioning problem** (the classical problem) which arises in trying to distribute a mesh amongst a set of processors;
(ii) **static load-balancing problem** which arises from a mesh that has been generated in parallel;
(iii) **dynamic load-balancing/partitioning problem** which arises from either adaptively refined meshes or meshes in which the computational workload for each cell can vary with time. It can also arise from computing resources with changing patterns of external load (e.g. a network of workstations).

In the latter two cases, (ii) & (iii), the initial data is a distributed graph which may

Ninth International Conference on Domain Decomposition Methods
Editor Petter E. Bjørstad, Magne S. Espedal and David E. Keyes

be neither load-balanced nor optimally partitioned. One way of dealing with this is to ship the graph back to some host processor, run a serial static partitioning algorithm on it and redistribute. However, this is unattractive for many reasons. Firstly, an $O(N)$ overhead for the mesh partitioning is simply not scalable if the solver is running at $O(N/P)$. Indeed the graph may not even fit into the memory of the host machine and may thus incur enormous delays through memory paging. In addition, a partition of the graph (which may even be optimal) already exists, so it makes sense to reuse this as a starting point for repartitioning [WB95]. 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. Thus, because the graph is already distributed, it is a natural strategy to repartition it *in situ*.

The algorithms developed here are therefore designed to iteratively optimise and if necessary load-balance an existing partition in parallel. In the first case, (i) above, an initial partition is generated using a fast but suboptimal partitioner such as the greedy algorithm and then the data is distributed.

2 Optimisation

In this section we present two complementary iterative algorithms which combined together form a powerful and flexible technique for optimising unstructured mesh partitions. Initially the subdomain heuristic attempts to 'improve' the 'shape' of the subdomains. However, this heuristic cannot guarantee load-balance and so a second heuristic, a parallel version of the Kernighan-Lin algorithm, [KL70], which also incorporates load-balancing is applied to share the workload equally between all subdomains and to carry out local refinement.

The subdomain heuristic

The idea behind the subdomain heuristic is to minimise the surface energy of the subdomains (in some graph sense). This is achieved by each processor determining the centre of its subdomain and then measuring the radial distance from the centre to the border of the subdomain and attempting to minimise this by migrating vertices which are furthest away.

Determining the 'centre' of a subdomain is relatively easy and can be achieved by moving in level sets inwards from the subdomain border until all the vertices in the subdomain have been visited. The final set defines the centre of the subdomain and, if the graph is connected (assumed), the level sets will completely cover the subdomain, although the centre may not be a connected set of vertices. The reverse of this process can then be used to determine the radial distance.

Having derived these sets each vertex can be marked by its radial distance. Nodes that are not connected to the centre are not marked and this is useful for migrating small disconnected parts of a subdomain to more appropriate processors. Neighbouring processors are informed of the radial distances of vertices on their borders and the vertices are migrated according to a combination of load-imbalance, radial distance and the change in cut-edges. This decision process is fully described in [WCE95a].

Local refinement & load-balancing

Having achieved approximate load-balance and good global subdomain shapes, a process of local refinement and exact load-balancing takes place.

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 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 [VK95], Vanderstraeten *et al.* demonstrate that it can be extremely effective to vary the cost function based on a knowledge of the solver. Meanwhile, in [WCE$^+$95c] 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, however, we shall assume that the gain $g(v, q)$ just expresses the reduction in the cut-edge weight, $|E_c|$.

Load-balancing. The load-balancing problem, i.e. how to distribute N tasks over a network of P processors so that none have more than $\lceil N/P \rceil$, is a very important area for research in its own right with a vast range of applications. Here we use an elegant technique recently developed by Hu & Blake, [HB95], related to, but with faster convergence than the commonly used diffusive methods, e.g. [GMS95], and which minimises the Euclidean norm of the transferred weight.

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 local refinement mechanism to decide which vertices to move.

The parallel local refinement mechanism. An algorithm which comes to mind for local refinement purposes is the Kernighan-Lin (KL) heuristic, [KL70], and in particular a linear-time variant proposed by Fiduccia & Mattheyses (FM), [FM82]. We use an algorithm largely inspired by the KL/FM algorithms but with several modifications to better suit our purposes. In particular, only boundary vertices are allowed to migrate and only to neighbouring processors.

The algorithm, which is fully described in [WCE97b, WCE95b], is thus run in the boundary regions of the subdomains and at each iteration a processor, p, calculates the preference and gain of its own border vertices and the desired flow across each p-q interface with neighbouring processors q and a halo update is carried out. Next, for each interface, the processor to which it has been assigned, p say, creates a bucket list structure (as in the FM algorithm) for border vertices v owned by itself which have preference $f(v) = q$ and halo vertices u owned by q which have preference $f(u) = p$. Vertices are then iteratively selected from either subdomain so as to firstly satisfy the flow as far as possible and secondly maximise the gain as much as possible.

3 Graph Reduction

For coarse granularity partitions it is inefficient to apply the optimisation techniques to every graph vertex as most will be internal to the subdomains. A simple technique to speed up the optimisation process, therefore, 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 apply the partitioning algorithm to these reduced size graphs. This is quite a common technique and has been used by several authors in various ways – for example, in a multilevel way analogous to multigrid techniques [BS94, HL95], and in an adaptive way analogous to dynamic refinement techniques, [WB95].

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, [HL95]. 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, $|V| = |V'|$. The total edge weight is reduced (by an amount equal to the weight of the collapsed edges), but the weight of the cut edges remains the same, $|E_c| = |E'_c|$.

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 random unmatched neighbour. 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 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. Nodes 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 [WCE97b].

4 Results

The software tool written at Greenwich to implement the optimisation techniques is known as JOSTLE. This software has been previously demonstrated to provide partitions of higher quality than MRSB, [WCE95a], and here, due to space constraints, we concentrate on parallel timings and the effect of the initial partition. Results which demonstrate the optimisation techniques applied to adaptively refined meshes can be found in [WCE97a] and the use of JOSTLE for mapping partitions onto machine topologies can be found in [WCE$^+$95c].

Metrics

We use two metrics to measure the performance of the algorithms, the total weight of cut edges, $|E_c|$ and $t(s)$, the execution time in seconds. The best measurement of the partition quality, and ultimately the only important one, is the parallel efficiency of the application from which the graph arises on a given machine. Unfortunately, however, this efficiency will depend on many things – typically the machine (size, architecture, latency, bandwidth and flop rate), the solution algorithm (explicit, implicit with direct linear solution, implicit with iterative linear solution) and the problem itself (size, no. of iterations) all play a part (see also Section 2). As a result it is impossible to fully assess a partitioning method independent of the solver and the machine to be employed and to do so goes beyond the scope of this paper. Here, therefore, we use $|E_c|$ to give a rough indication of the volume of communication traffic.

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 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 usually extremely inexpensive relative to the overall solution time of the problem.

Tables 1 and 2 give some typical results for 2D (`tri60k`) and 3D (`brack2`) meshes on the Edinburgh Cray T3D with up to 128 processors. These demonstrate very good speedups for this sort of code and more importantly, very low overheads (of the order of a few seconds) for the parallel partitioning. Note that the $|E_c|$ results obtained for the parallel version of JOSTLE may not be exactly the same as those of the serial version, due to different orderings of linked lists, but that, since these are random orderings, there are no consistent differences in quality.

Table 1 Results for `tri60k` mesh: $N = 60005$, $E = 89440$

	serial		parallel		
P	$\|E_c\|$	$t(s)$	$\|E_c\|$	$t(s)$	speed up
16	1104	14.64	1093	2.65	5.52
32	1669	15.67	1668	1.88	8.33
64	2530	19.79	2572	1.66	11.92
128	3698	24.32	3721	1.29	18.85

Table 2 Results for `brack2` mesh: $N = 62032$, $E = 121544$

	serial		parallel		
P	$\|E_c\|$	$t(s)$	$\|E_c\|$	$t(s)$	speed up
16	13717	35.19	13442	7.13	4.93
32	21098	40.85	21004	4.90	8.34
64	30407	54.21	30276	4.33	12.52
128	43109	59.40	42959	2.65	22.41

The initial partition

We have tested the optimisation techniques with a variety of initial partitioning algorithms. Two crude techniques are *random* partitioning which assigns the vertices randomly and *block* partitioning which assigns the first N/P vertices to processor 0, the next N/P to processor 1, etc. These are attractive as the data can, in principle, be input in parallel. However, random partitioning gives something close to a worst-case initial partition and block partitioning can be very poor, particularly in the case of an advancing front mesh generator (as used for the `whitaker3` mesh) where the mesh elements spiral in towards the centre. A slightly more effective technique is geometric sorting where the elements are sorted according to their x, y (and z for 3D) coordinates and each dimension is partitioned in a strip-wise fashion. This too can be carried out in parallel (using a parallel sorting algorithm) but can create long thin and sometimes multiply connected domains. The final algorithm we have tested is the Greedy algorithm [Far88]. This is clearly seen to be the fastest *graph-based* method as it only visits each graph edge once, but can only be applied in serial.

Table 3 Different initial partitions for `whitaker3`: $N = 9800$, $E = 28989$, $P = 32$

initial algorithm	initial $\|E_c\|$	optimised $\|E_c\|$	$t(s)$
random	28083	1848	3.61
loop	11571	1882	2.02
geosort	1854	1818	2.26
greedy	2143	1805	2.12

Tables 3 & 4 show the results obtained from a Sun 20 with a 75 MHz CPU and 128 Mbytes of memory. As can be seen, the quality of the final optimised partition does

Table 4 Different initial partitions for `barth5`: $N = 15606$, $E = 45878$, $P = 64$

initial algorithm	initial $\lvert E_c\rvert$	optimised $\lvert E_c\rvert$	$t(s)$
random	45174	3096	7.38
loop	10643	2930	4.04
geosort	5416	2905	4.20
greedy	4046	2970	3.88

not vary significantly with the initial partitioner chosen (except for a little noise) and thus the optimisation techniques are demonstrated to be very powerful, in particular for the worst-case *random* partition. What is affected however is the partitioning time and in general, as might be expected, the poorer the quality of the initial partition, the longer it takes to optimise it.

5 Conclusion

We have outlined a new method for optimising graph partitions with a specific focus on its application to the mapping of unstructured meshes onto parallel computers. In this context the static graph-partitioning task can be very efficiently addressed through a two-stage procedure – one to yield a legal initial partition and the second to improve its quality with respect to interprocessor communication and load-balance. The method is further enhanced through the use of a clustering technique. For the experiments reported in this paper the cost of parallel partitioning is shown to be of the order of a few seconds even for relatively large graphs. In addition, the partition quality is shown to be reasonably independent of the initial partition.

Acknowledgement

We would like thank the Edinburgh Parallel Computer Centre and the Engineering and Physical Sciences Research Council for access to the Cray T3D.

REFERENCES

[BS94] Barnard S. T. and Simon H. D. (1994) A Fast Multilevel Implementation of Recursive Spectral Bisection for Partitioning Unstructured Problems. *Concurrency: Practice & Experience* 6(2): 101–117.

[Far88] Farhat C. (1988) A Simple and Efficient Automatic FEM Domain Decomposer. *Comp. & Struct.* 28(5): 579–602.

[FM82] Fiduccia C. M. and Mattheyses R. M. (1982) A Linear Time Heuristic for Improving Network Partitions. In *Proc. 19th IEEE Design Automation Conf.*, pages 175–181. IEEE, Piscataway, NJ.

[FS93] Farhat C. and Simon H. D. (1993) TOP/DOMDEC – a Software Tool for Mesh Partitioning and Parallel Processing. Tech. Rep. RNR-93-011, NASA Ames, Moffat Field, CA.

[GMS95] Ghosh B., Muthukrishnan S., and Schultz M. H. (1995) Faster Schedules for Diffusive Load Balancing via Over-Relaxation. TR 1065, Department of Computer Science, Yale University, New Haven, CT 06520, USA.

[HB95] Hu Y. F. and Blake R. J. (1995) An optimal dynamic load balancing algorithm. Preprint DL-P-95-011, Daresbury Laboratory, Warrington, WA4 4AD, UK.

[HL95] Hendrickson B. and Leland R. (1995) A Multilevel Algorithm for Partitioning Graphs. In *Proc. Supercomputing '95.*

[KL70] Kernighan B. W. and Lin S. (February 1970) An Efficient Heuristic for Partitioning Graphs. *Bell Systems Tech. J.* 49: 291–308.

[VK95] Vanderstraeten D. and Keunings R. (1995) Optimized Partitioning of Unstructured Computational Grids. *Int. J. Num. Meth. Engng.* 38: 433–450.

[WB95] Walshaw C. H. and Berzins M. (1995) Dynamic load-balancing for PDE solvers on adaptive unstructured meshes. *Concurrency: Practice & Experience* 7(1): 17–28.

[WCE95a] Walshaw C., Cross M., and Everett M. (1995) A Localised Algorithm for Optimising Unstructured Mesh Partitions. *Int. J. Supercomput. Applics.* 9(4): 280–295.

[WCE95b] Walshaw C., Cross M., and Everett M. (1995) Dynamic mesh partitioning: a unified optimisation and load-balancing algorithm. Tech. Rep. 95/IM/06, University of Greenwich, London SE18 6PF, UK.

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

[WCE97a] Walshaw C., Cross M., and Everett M. (1997) Dynamic load-balancing for parallel adaptive unstructured meshes. In Heath *et al* M. (ed) *Parallel Processing for Scientific Computing*. SIAM, Philadelphia.

[WCE97b] Walshaw C., Cross M., and Everett M. (1997) Parallel Unstructured Mesh Partitioning. (in preparation).