JOSTLE: Partitioning of Unstructured Meshes for Massively Parallel Machines

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

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

We outline the philosophy behind a new method for solving the graph-partitioning problem which arises in mapping unstructured mesh calculations to parallel computers. The method, encapsulated in a software tool, JOSTLE, employs a combination of techniques including the Greedy algorithm to give an initial partition, together with some powerful optimisation heuristics. A clustering technique is additionally employed to speed up the whole process. The resulting partitioning method is designed to work efficiently in parallel as well as sequentially and can be applied to both static and dynamically refined meshes. Experiments on graphs with up to a million nodes indicate that JOSTLE is up to an order of magnitude faster than existing state-of-the-art techniques such as Multilevel Recursive Spectral Bisection, whilst providing partitions of equivalent quality.

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

1. Introduction

The use of unstructured mesh codes on parallel machines is one of the most efficient ways to solve large Computational Fluid Dynamics (CFD) 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. However, unlike their structured counterparts, one must carefully address 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. [3]. In this paper we discuss the mesh partitioning problem in the light of the coming generation of massively parallel machines and the resulting implications for such algorithms.

1.1. Mesh partitioning issues

As CFD establishes itself as a reliable and invaluable tool for scientists and engineers, the trend is to solve larger and larger problems (for example 2D airfoil simulations grow to 3D wings and even whole aircraft). An immediate consequence is that, as mesh and machine sizes grow the need for **parallel** mesh-partitioning becomes increasingly acute, since an $O(N)$ overhead is simply not scalable. In addition, developing techniques such as parallel mesh-generation give rise to meshes which are already distributed across the memory of the parallel machine. In these cases it is extremely expensive to transfer the whole mesh back to a single processor for sequential load-balancing, if indeed the memory of that processor allows it. Löhner *et al.* have

advanced some powerful arguments in support of this proposition, [8].

Another extremely important aspect of partitioning arises from time-dependent unstructured mesh codes which adaptively refine the mesh. This is a very efficient way to track phenomena which traverse the solution domain but means that the position and density of the mesh points may vary dramatically over the course of an integration and hence require **dynamic load-balancing**. Again this calls for a parallel partitioning algorithm and, in addition, the issues of cost and reuse arise. 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 solution algorithm in between remeshing. This may seem to restrict us to computationally cheap algorithms but fortunately help is at hand if the mesh has not changed too much, for in this case 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, [15]. 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.

1.2. Overview

The strategy developed here to tackle these issues efficiently is to derive a partition as quickly and cheaply as possible, distribute the data and then *optimise* the partition *in parallel*. Of course, if the mesh is already distributed then the existing partition is used and optimisation can commence immediately. Currently the method has only been implemented sequentially, but the optimisation algorithm has been designed with parallelism in mind and here we seek to validate the method from the point of view of partition quality.

This multi-stage approach is similar to the work of Vanderstraeten, [12], although the techniques vary in that we employ deterministic heuristics to optimise the partition. The philosophy behind the optimisation algorithm is close to that of Löhner *et al.*, [8], although in addition we employ a heuristic which attempts to improve the 'shape' of the subdomains. In common with many such techniques in this field, the algorithms work with a graph based on the communication requirements of the mesh.

In the rest of this paper we present an overview of these algorithms which, when combined together, form a powerful and flexible technique for partitioning unstructured meshes. In particular we outline a parallel meta-heuristic for optimising partitions, §3, and a method for coarsening the communication graph arising from the mesh in order to substantially reduce the workload, §4.

The software tool written at Greenwich to implement these ideas is known as JOSTLE. This is not an acronym; rather it reflects the way the subdomains jostle one another to reach a steady-state. It is modular in nature and consists of three parts; the initial partitioning module, the partition optimisation module and the graph coarsening module. These modules are designed for interchangeable use: in the sequential implementation the code can coarsen the graph, partition the reduced graph, optimise the partition, interpolate onto the full graph and reoptimise. With parallel code one might partition the full graph sequentially, distribute the graph, coarsen each subdomain, optimise the partition of the reduced graph, interpolate and reoptimise.

The initial partitioning code uses a sequential algorithm and is only employed if the input data is stored or generated on a single processor. The optimisation code consists of three complementary techniques grouped together to form a meta-heuristic. Collectively they take as input a graph and partition $(G, \mathcal{P}_1)$ and iterate to try and output a better partition $(\mathcal{P}_2)$. The graph reduction code takes as input a graph (G) and outputs another reduced size graph (g).

Ideally the reduced graph should be considerably smaller than the original, but still capture the essential geometric or topological features.

Due to the modular nature of the code and the fact that the optimisation and graph coarsening use parallelisable algorithms, all the issues raised by adaptive refinement are addressed by our mesh-partitioning techniques. They do not rely on the load being balanced, they can reuse an existing partition and the methods are designed to work efficiently in parallel.

2. The initial partition

The aim of the initial partitioning code is to divide up the graph as rapidly as possible in order that it can be distributed and the partition optimised in parallel. Whilst a block-based partition (e.g. the first N/P nodes to processor 1, etc.) is probably the fastest method for doing this, intuition suggests and experience shows that it is worth a little extra effort in order to reduce the amount of optimisation that must be carried out.

The code therefore utilises a version of the Greedy algorithm [2]. This is clearly seen to be the fastest *graph-based* method as it only visits each graph edge once. The variant employed here differs from that proposed by Farhat only in that it works solely with a graph rather than the nodes and elements of a finite element mesh.

3. The parallel optimisation method

Once the graph is partitioned, optimisation can take place to improve the quality of the partition. The method outlined here uses a combination of three heuristics to both achieve load-balance and to minimise the interprocessor communication. Initially the subdomain heuristic attempts to 'improve' the 'shape' of the subdomains. However, this heuristic cannot guarantee load-balance and so a second heuristic is applied to share the workload equally between all processors. Finally a localised version of the Kernighan-Lin algorithm, [7], is applied to minimise the communication cost.

Throughout each of the three phases, it is assumed that the final partition will not deviate too far from the initial one. Thus, in general, only border nodes are allowed to migrate to neighbouring subdomains and the only time that internal nodes migrate is in the first phase, §3.1, when the heuristic can detect small subsets of a subdomain which are disconnected from the main body.

3.1. The subdomain heuristic

This new heuristic has been designed according to two constraints. We want the algorithm to

(a) address the optimisation at a subdomain level to try and attain a *global* minimum;

(b) carry out the optimisation *locally* to give an efficient parallel algorithm.

At first these two constraints may appear to frustrate each other. However, a useful analogy with a simple schoolroom experiment can be drawn. The experiment consists of bubbling a gas through a pipette into a detergent solution and the result is that the bubbles make a regular (hexagonal) pattern on the surface of the liquid. They achieve their regularity without any global 'knowledge' of the shape of the container or even of any bubbles not in immediate contact; it is simply achieved by each bubble minimising its own surface tension.

Translating this analogy to a partitioned graph we see that each subdomain must try to minimise its own surface area. In the physical 2D or 3D world the object with the smallest

surface to volume ratio is the circle or sphere. Thus the idea behind the subdomain heuristic is to determine the centre of each subdomain (in some graph sense) and to then measure the radial distance from the centre to the edges and attempt to minimise this by migrating nodes which are furthest from the centre.

Determining the 'centre' of the subdomain is relatively easy and can be achieved by moving in level sets inwards from the subdomain border until all the nodes 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 nodes. The reverse of this process can then be used to determine the radial distance.

Having derived these sets each node can be marked by its radial distance. Nodes which are not connected to the centre are not marked and this is useful for migrating small disconnected parts of the subdomain to more appropriate processors. The code decides which nodes to migrate based on a combination of radial distance, load-imbalance and the change in cut-edges. This decision process is fully described in [13].

3.2. 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, [10]. Here we use a localised iterative algorithm for distributed load-balancing devised by Song, [11]. In common with the rest of our optimisation, the localised nature means that it only requires information from, and migrates nodes to, its neighbours in the processor graph. The description of the algorithm assumes that the workload consists of independent tasks of equal size and is proven to iterate to convergence with a maximum load-difference of $d/2$ where d is the diameter of the processor graph.

Translating this to the mesh partitioning problem, we use Song's algorithm to determine the *number* of nodes to migrate from a given processor to each of its neighbours and then use the change in cut-edges to determine *which* nodes to move. The integration of this algorithm into the JOSTLE code is fully described in [13].

3.3. Local partition refinement

Having achieved optimal (or near optimal) load-balance it may still be possible to move nodes around the processor network to further minimise the number of cut edges whilst retaining the load-balance. An algorithm which comes immediately to mind for this purpose is the Kernighan-Lin (KL) heuristic, which maintains load-balance by employing pairwise exchanges of nodes. Unfortunately it has $O(n^2 \log n)$ complexity but a linear-time variant which delivers similar results has been proposed by Fiduccia & Mattheyses (FM), [4]. The FM algorithm achieves this reduction partially by calculating swaps one node at a time rather than in pairs.

The algorithm used by JOSTLE is largely inspired by the FM algorithm but with several simplifications to enable an efficient parallel algorithm. In particular:–

- In our applications it is very unlikely that an internal node will have a higher gain than a border node – thus we only consider transferring border nodes.
- It is also unlikely that an overall gain will accrue by transferring a node to a subdomain to which it is not adjacent – thus nodes only transfer to neighbouring subdomains.

One immediate advantage of these modifications is that by only considering border nodes the problem has sublinear complexity.

The algorithm is fully described in [13].

4. Clustering: reducing the problem size

For coarse granularity partitions it is inefficient to apply the optimisation techniques to every graph node as most will be internal to the subdomains. A simple technique to speed up the load-balancing process, therefore, is to group nodes 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 this reduced size graph. 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, [1], and in an adaptive way analogous to dynamic refinement techniques, [15].

The technique used here for graph reduction is a variant of the Greedy algorithm, [2], although various algorithms have been successfully employed, [1,5,6]. It is used recursively to cluster nodes into small groups, each of which defines a node of the reduced graph. It is, of course, important that the groups of nodes are connected (in order to retain the features of the original graph) and so we relax the condition that each cluster should contain the same node weight in favour of guaranteeing that the nodes of each cluster form a connected set. The node and edge weights for the reduced graph derive simply from the sum of node weights over the cluster and sum of edge weights from the cluster to other clusters.

5. Experimental results

5.1. 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 of each algorithm. Unfortunately, there are no *ideal* metrics for assessing partition quality as the parallel efficiency of the problem from which the mesh arises 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.

We have compared the method with two of the most popular partitioning algorithms, Greedy and Multilevel Recursive Spectral Bisection (MRSB). The Greedy algorithm, [2], is actually performed as part of the jostle code and is fast but not particularly good at minimising cut edges. MRSB, on the other hand, is a highly sophisticated method, good at minimising $|E_c|$ but suffering from relatively high runtimes, [1].

5.2. The results

The following experiments were mostly carried out on a Silicon Graphics Indigo 2 with a 150 MHz CPU and 64 Mbytes of memory. The final set of results for the largest mesh came from a Sun SPARC station with a 50 MHz CPU and 224 Mbytes of memory; typically this processor is about one and a half times slower than the Silicon Graphics machine but the memory requirements forced its usage. The code has also been compiled and run on an IBM RS6000 with similar timing ratios. A more detailed analysis of selected results can be found in [14] and further results dealing with the optimisation technique found in [13].

We include results for runs on 5 different graphs. The first two, the Hammond mesh and the Barth5 mesh are available by anonymous ftp (from `riacs.edu/pub/grids`) and have been previously been used for benchmarking partitioning algorithms, [1,5]. The Hammond mesh is a small ($N = 4,720$, $E = 13,722$) two-dimensional finite-element mesh around a 3-element

airfoil and Barth5 is a similar but larger ($N = 15,606$, $E = 45,878$) mesh around a 4-element airfoil. The other meshes are homegrown: Tri60K is a two-dimensional finite-volume mesh ($N = 60,005$, $E = 89,440$) arising from a casting simulation, [9]. Tet100K and Tet1M are three-dimensional finite-volume meshes ($N = 103,081$, $E = 200,976$; $N = 1,119,663$, $E = 2,212,012$ respectively) in the shape of a Y standing on a baseplate.

For each mesh the jostle code has been run using the graph coarsening, §4, to create a reduced graph. A partition is generated and optimised on this reduced graph, interpolated onto the full graph and then reoptimised. For the Tet1M results the subdomain heuristic was not used on the full graph. The MRSB code was made available to us by one of its authors, Horst Simon, and run unchanged with a contraction threshold of 100.

It can be seen from the experiments that for almost all cases the $|E_c|$ results for JOSTLE are consistently better than those for MRSB. In addition, the execution times for JOSTLE are between 2.5 and 7 times faster. As would be expected, the execution times for the GREEDY algorithm are the fastest, but the $|E_c|$ results are much worse.

For any of these algorithms it is virtually impossible to derive a meaningful complexity function (although the GREEDY algorithm is approximately $O(E)$). In particular for JOSTLE and MRSB the clustering process obscures the problem size. It is interesting to note, however, that the execution times for MRSB grow much more rapidly with P than do those for JOSTLE. This suggests that for even larger machines JOSTLE should perform even faster relative to MRSB.

In the final set of results, table 5, we offer an example of how fast the code can be on a workstation for a huge mesh.

6. Conclusion

The work described above has outlined a new method for partitioning graphs with a specific focus on its application to the mapping of unstructured meshes onto massively parallel computers. In this context the 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. The resulting software tool, JOSTLE, has been designed for implementation in parallel and for both static and dynamically refined meshes. For the experiments reported in this paper on static meshes of up to one million nodes, the JOSTLE procedures are an order of magnitude faster than existing techniques, such as Multilevel Recursive Spectral Bisection, with equivalent quality.

Acknowledgements

We would like to thank Horst Simon for the copy of his Multilevel Recursive Spectral Bisection code and Peter Lawrence and Kevin McManus for supplying the Tri60K, Tet100K and Tet1M meshes.

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(2):101–117, 1994.
2. C. Farhat. A Simple and Efficient Automatic FEM Domain Decomposer. *Comp. & Struct.*, 28:579–602, 1988.
3. C. Farhat and H. D. Simon. TOP/DOMDEC – a Software Tool for Mesh Partitioning and Parallel Processing. Tech. Rep. RNR-93-011, NASA Ames, Moffat Field, CA, 1993.

4. C. M. Fiduccia and R. M. Mattheyses. A Linear Time Heuristic for Improving Network Partitions. In *Proc. 19th IEEE Design Automation Conf.*, pages 175–181, IEEE, 1982.
5. B. Hendrickson and R. Leland. A Multilevel Algorithm for Partitioning Graphs. Tech. Rep. SAND 93-1301, Sandia National Labs, Albuquerque, NM, 1993.
6. B. W. Jones. *Mapping Unstructured Mesh Codes onto Local Memory Parallel Architectures*. PhD thesis, School of Maths., University of Greenwich, London SE18 6PF, UK, 1994.
7. B. W. Kernighan and S. Lin. An Efficient Heuristic for Partitioning Graphs. *Bell Systems Tech. J.*, 49:291–308, 1970.
8. R. Lohner, R. Ramamurti, and D. Martin. A Parallelizable Load Balancing Algorithm. AIAA-93-0061, 1993.
9. K. McManus, M. Cross, and S. Johnson. Integrated Flow and Stress using an Unstructured Mesh on Distributed Memory Parallel Systems. In *Parallel CFD'94*. Elsevier, 1995.
10. N. G. Shivaratri, P. Krueger, and M. Singhal. Load distributing for locally distributed systems. *IEEE Comput.*, 25(12):33–44, 1992.
11. J. Song. A partially asynchronous and iterative algorithm for distributed load balancing. *Parallel Comput.*, 20:853–868, 1994.
12. D. Vanderstraeten and R. Keunings. Optimized Partitioning of Unstructured Finite Element Meshes. (accepted by *Int. J. Num. Meth. Engng.*), 1993.
13. C. Walshaw, M. Cross, and M. Everett. A Parallelisable Algorithm for Optimising Unstructured Mesh Partitions. (submitted for publication), 1994.
14. C. Walshaw, M. Cross, S. Johnson, and M. Everett. A Parallelisable Algorithm for Partitioning Unstructured Meshes. In *Proc. Irregular '94: Parallel Algorithms for Irregularly Structured Problems*, 1994.
15. C. H. Walshaw and M. Berzins. Dynamic Load-Balancing For PDE Solvers On Adaptive Unstructured Meshes. (accepted by *Concurrency: Practice & Experience*), 1993.

Table 1
Results for the Hammond mesh: $N = 4,720$, $E = 13,722$

	GREEDY		JOSTLE		MRSB	
P	$\|E_c\|$	$t(s)$	$\|E_c\|$	$t(s)$	$\|E_c\|$	$t(s)$
16	1061	0.05	669	0.66	696	2.36
32	1456	0.05	1078	0.66	1155	2.68
64	2102	0.05	1706	0.74	1784	3.21
128	3017	0.06	2571	1.32	2678	4.21
256	4270	0.06	3691	1.65	3895	5.12

Table 2
Results for the Barth5 mesh: $N = 15,606$, $E = 45,878$

	GREEDY		JOSTLE		MRSB	
P	$\|E_c\|$	$t(s)$	$\|E_c\|$	$t(s)$	$\|E_c\|$	$t(s)$
16	2033	0.26	1142	2.04	1282	8.60
32	2947	0.29	1850	2.16	1973	9.97
64	4046	0.29	2949	2.76	3122	11.45
128	5678	0.31	4622	3.25	4785	13.64
256	8121	0.30	6699	5.79	7109	15.86

Table 3
Results for Tri60K mesh: $N = 60,005$, $E = 89,440$

	GREEDY		JOSTLE		MRSB	
P	$\|E_c\|$	$t(s)$	$\|E_c\|$	$t(s)$	$\|E_c\|$	$t(s)$
16	1546	0.87	1048	4.45	953	23.33
32	2595	0.93	1665	4.68	1564	29.23
64	3834	1.00	2432	6.83	2435	35.22
128	5822	1.79	3684	7.01	3712	41.43
256	8107	2.35	5423	10.32	5424	50.88

Table 4
Results for Tet100K mesh: $N = 103,081$, $E = 200,976$

	GREEDY		JOSTLE		MRSB	
P	$\|E_c\|$	$t(s)$	$\|E_c\|$	$t(s)$	$\|E_c\|$	$t(s)$
16	10939	3.04	6212	14.70	6213	38.11
32	15549	3.17	9038	15.72	9459	61.17
64	20017	1.87	12426	16.56	13035	107.16
128	26274	2.07	15784	23.78	17751	148.30
256	33556	2.14	22250	24.51	22745	172.47

Table 5
Results for Tet1M mesh: $N = 1,119,663$, $E = 2,212,012$

P	$\|E_c\|$	$t(s)$
10	49549	133.37
20	64088	133.67
50	87060	132.32
100	116966	148.85
200	129963	167.92