Implementation of Multigrid on Parallel Machines Using Adaptive Finite Element Methods

Linda Stals

1 Introduction

Multigrid methods and adaptive finite element methods are well established as powerful tools for the solution of partial differential equations. When implementing these methods, parallel computers are often considered because of their ability to solve large problems quickly. However, in practice the implementation of multigrid methods on these machines is non-trivial due to the intra-grid and inter-grid data dependencies. Furthermore, the non-uniformity of the grids generated by adaptive refinement leads to a 'conflict of interest' as parallel machines are better suited to uniform grids. Consequently the implementation of these methods in parallel is a sizeable software engineering problem. In this paper we describe a parallel implementation based upon the use of a node-edge data structure.

The node-edge data structure stores the grids by placing the geometrical information in the node table and the topological information in the edge table. To use the data structure in parallel we have also included a ghost-node table. The ghost-node table is a generalisation of the boundary layer or artificial boundary often used in the parallel implementation of structured grids.

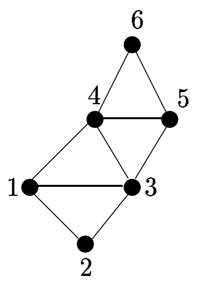
The grids are refined by using the newest node bisection method. We have developed a parallel extension of Mitchell's compatibly divisible triangle method ([Mit88], [Mit89], [Mit92]) to ensure that the angles remain bound away from 0 and π during adaptive refinement.

This paper also gives some example runs.

2 Node-Edge Data Structure

The basic idea behind the node-edge data structure is easily illustrated by an example. Consider the grid in figure 1. It can be broken down in terms of its geometrical and

Figure 1 Example grid stored in the node-edge data structure.



topological components and stored in the following node and edge tables:

Node: 1(0.0, 0.0), 2(3.0, -3.0), 3(5.0, 0.0), 4(3.0, 3.0), 5(7.0, 3.0), 6(5.0, 5.0)

Edge: 1-2, 1-3, 1-4, 2-1, 2-3, 3-1, 3-4, 3-5, 4-1, 4-3, 4-5, 4-6, 5-3, 5-4, 5-6, 6-4, 6-5

Our implementation of this data structure is based upon the one given by Rüde in [Rüd92], [Rüd93a] and [Rüd93b].

The advantage of this data structure is its flexibility. The same data structure can be used to store triangles, quadrilaterals and tetrahedrons. Most of our work has concentrated on triangular grids, but we have developed modules which use the node-edge data structure to store bilinear basis functions and we have started work on tetrahedrons.

The stiffness matrix is stored in a connection table. If the entry in the stiffness matrix corresponding to, say, node i and node j is non-zero then nodes i and j are connected. For linear basis functions, the connection table looks very similar to the edge table. The connection table is also used to store other algebraic information such as the interpolation and restriction operators used in the multigrid algorithm.

3 Parallel Implementation

To use the data structure in parallel we include a ghost-node table and a neighbour-node table. Note that in the parallel implementation we call the node table the full-node table, this helps to differentiate between the whole grid and the grid segments stored in the processors.

The concepts behind the ghost-node table are easily shown by an example. Consider the grid given in Figure 1, and suppose nodes 4, 5 and 6 were placed in the full-node table for one processor while nodes 1, 2 and 3 were placed into the full-node table of another processor. Then the edges for the first processor are completed by adding nodes 1 and 3 as ghost-nodes and the edges for the other processor are completed by adding nodes 4 and 5 as ghost-nodes (see Figure 2).

The example in Figure 2 showed how the ghost-nodes complete the edge table, but they are also used to complete the connection table and consequently complete the

Figure 2 Example grid spread over two processors. The ghost-nodes, shown by open circles, complete the edges.

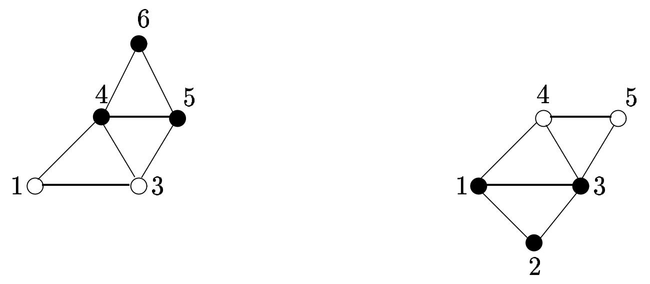
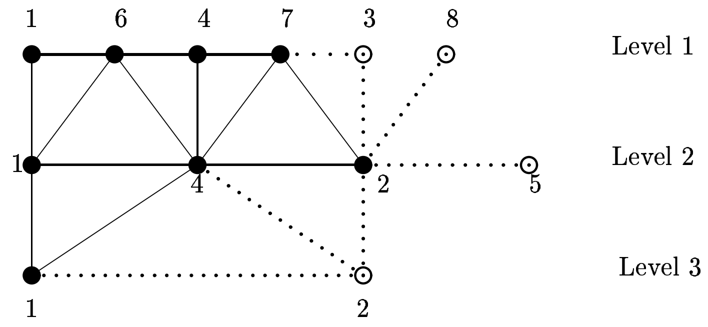


Figure 3 Example of a 1D grid with three levels of refinement. The ghost-nodes complete the intra-grid and inter-grid connections. The full-nodes are drawn as dark circles while the ghost-nodes are drawn as open circles.



inter-grid connections. See Figure 3. This is the basis of our parallel implementation of the multigrid method.

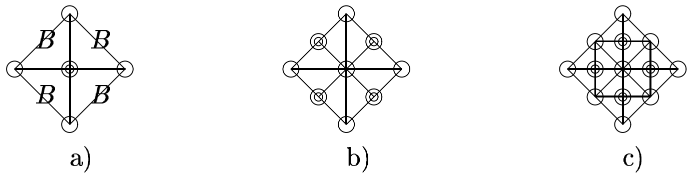
In order to communicate any updates, the full-nodes must know which processors contain any corresponding ghost-node and each ghost-node must know which processor contains the corresponding full-node. This information is stored in the neighbour node table.

The use of ghost-nodes as a communication buffer or as a way of storing updates from neighbouring processors is not new, see for example the ghost cells used in [BKFF94] and artificial boundary in [MFL$^+$91]. However, we have extended their application so that they also define the data dependencies. For example, during refinement the communication pattern has to be updated when new nodes are added and by exploiting the relationship between the ghost-nodes and full-nodes this can be done independently across the processors.

4 Refinement

Our method of refinement is based upon the newest node bisection method. This method refines the triangles by splitting the edges which sit opposite the newest nodes. See [Mit88], [Mit89], [Mit92] and [Sta95] for a more detailed discussion. In the node-edge data structure it is easier to work with the base edges rather then the newest nodes. The base edges are the edges which sit opposite the newest nodes. Figure 4 shows an example.

Figure 4 Example refinement using newest node bisection. Figure a) shows the initial grid. We assume that the centre node is our initial 'newest node'. The corresponding bases edges are marked by a B. Figure b) shows the result after one refinement sweep. Figure c) shows the final grid.



When studying the mechanics behind the refinement we see that the most difficult part is adding the nodes. When a new node is added the algorithm must determine which processor should get the new node as a full-node and which, if any, should get it as a ghost-node. By exploiting the dependencies within the data structure a set of rules can be developed which lets each processor resolve this problems without the need for any communication.

In his thesis ([Mit88]), Mitchell describes a method of adaptive refinement which uses bisection and compatibly divisible triangles. The disadvantage of this method, from our point of view, is that it only works on one triangle at a time. We have extended Mitchell's method so that several triangles may be refined at once by using interface-base edges. Interface-base edges are edges which sit between two different levels of refinement. For example, in Figure 5 a) we have marked the interface-base edges by an I and the base edges by a B. The neighbouring coarse triangles must be refined before the interface-base edges are split. So if the edge marked by I_1 needed to be split, then the base edge B_3 must be split first as shown in Figure 5 b). Note that the interface-base edge I_1 has been updated to a base edge B_1. The edge B_1 can now be split to give the final grid shown in Figure 5 c).

By using this approach it is possible to split more then one triangle at a time. In Figure 5 a) I_2 could be split the same time as I_1.

Keeping track of the interface-base edges guarantees that the angles remain bound away from 0 and π. The disadvantage is that several of the neighbouring coarse grid triangles may need to be refined, in Figure 5 c) edges B_4 and I_8 must be split before I_{11}, and hence the refinement may travel across several processors. The algorithm uses communication to control the order of refinement around the boundary of the grid in each processor.

5 Load Balancing

After a refinement sweep (whole grid or adaptive) we may find that the load needs to be rebalanced. Rebalancing the load is a deceptively difficult problem. We only give a very brief overview of our four step heuristic algorithm, but this hides a lot of the detail and the subtle problems which arise in the actual implementation.

To re-balance the load we let the nodes 'flow' out of the processors with too many nodes into the processors which do not have enough. By flow we mean that the nodes follow the edges between neighbouring processors.

The algorithm consists four steps. The first step calculates how many nodes should be moved in order to balance the load, the second steps picks which nodes should be moved, the third step finds which processors the nodes should be moved to and the final step moves the nodes.

More information is given in [Sta95].

6 Example Runs

The program is written in a mixture of C^{++} and PVM.

The results given in this paper were obtained on the Fujitsu AP1000. The AP1000 is a distributed memory MIMD machine with 128 processors arranged in a 2D torus. Each processor uses a 25MHz SPARC chip. For further information see [Aus91], [Aus92], [Fuj90], [Haw91] and [IHI$^+$].

The example we shall consider solves the equation $-\Delta u + u = -e^x e^y$ on the octagon shown in Figure 6, with the boundary condition chosen so that the exact solution is $u = e^x e^y$. The results are given in table 1.

The whole grid was refined to the given number of levels by using the newest node bisection method. Two iterations of the V-scheme were applied with two pre and two post smoothers. The efficiency results are calculated as $T_1/(p\,T_p) \times n_p/n_1$, where T_p is the total time for p processors and n_p is the number of nodes for p processors.

The time has been broken up into the four major modules; the FEM module which calculate the stiffness matrix, the V-scheme module which solves the system of equations, the Refine module which refines the grid and the Load module which balances the load.

The efficiency for the FEM module is very high. The ghost-nodes have been used to complete the connections, so this module does not need to do any communication.

The efficiency for the Refine module is also high. By exploiting the relationship between the ghost-nodes and full-nodes we were able to refine the grids in parallel without any communication.

The efficiency for the V-scheme module drops off for large number of processor. This is as expected since the V-scheme module spends more time in the coarse grids then the other modules. Note that the coarsest grid only contains nine nodes so there will be a lot of idle processors when doing computations on the coarse grids.

The overall efficiency decreases as we increase the number of processors. However, we can see from the times for the Load module that most of the increased cost comes from spreading the grids across the processors (after each new level of refinement is built we spread the grid out to fill up as many processors as possible). Once we have

Table 1 Efficiency results for $-\Delta u + u = -e^x e^y$ on a octagon domain.

No. of Processors	1	4	16	64
No. of Levels	5	6	7	8
No. of Nodes	4225	16641	66049	263169
Total (sec)	61.0	64.3	81.7	129.0
V-scheme (sec)	13.0	15.2	20.1	42.3
FEM (sec)	31.5	32.0	34.6	30.2
Refine (sec)	16.1	14.6	14.6	11.0
Load (sec)	0.0	3.3	19.0	67.9
Efficiency (%)		93	73	46

Table 2 The efficiency results for $-\Delta u = \sin(\pi x)\sin(\pi y)$ on the unit square domain. Note that the coarse grid size is increased as the number of processors is increased.

No. of Processors	1	4	16	64
No. of Levels	6	7	8	9
No. of Fine Nodes	4225	16641	66049	263169
No. of Coarse Nodes	81	289	1089	4225
Total (sec)	51.0	50.9	54.5	56.7
V-scheme (sec)	12.9	14.3	16.4	17.3
FEM (sec)	22.1	22.7	23.9	24.4
Refine (sec)	15.6	12.7	11.2	12.3
Load (sec)	0.0	1.3	5.5	6.3
Efficiency (%)		99	91	88

enough nodes to fill the processors the efficiency increases markedly. To verify this statement, we tried another example where the coarse grid size was also increased as the number of processors was increased. Table 2 gives the results for solving the equation $-\Delta u = \sin(\pi x)\sin(\pi y)$ on the unit square domain.

We have recently started working on non-linear problems. Figures 7 and 8 shows the result after solving the equation $-\Delta u - 2e^{-5\times 10^4} = -2$ using four levels of adaptive refinement. The domain is as shown in Figure 7 with $u = 1$ on the inner boundary and $u = 0$ on the outer boundary. As this problem is more computationally expensive then the previous example, the initial cost of setting up the grid (i.e. the cost of the Load Routine) is less significant. On a network of workstations the Load Routine took less then 5% of the overall time.

REFERENCES

[Aus91] Australian National University, Department Of Computer Science, Canberra, ACT, 0200, Australia (November 1991) *Proceedings Of The Second Fujitsu-ANU CAP Workshop.*

[Aus92] Australian National University, Department Of Computer Science, Canberra, ACT, 0200, Australia (March 30 1992) *AP1000 User's Guide.*

[BKFF94] Baden S. B., Kohn S. R., Figueira S. M., and Fink S. J. (11 April 1994) The LPARX user's guide, v 1.0. Technical report, Department of Computer Science and Engineering, University of California, San Diego, La Jolla, CA 92093-0114 USA.

[Fuj90] Fujitsu Laboratories Ltd., Computer Based Systems Lab. Kawasaki. Fujitsu Laboratories Ltd. 1015 Kamikondanaka, Nakahara-ku, Kawasaki 211, Japan (March 1990) *Cap-II Program Development Guide [1]: C-Language Interface*, second edition.

[Haw91] Hawking D. (May 1991) About the Fujitsu AP1000. Technical report, Department Of Computer Science, Australian National University, Canberra, ACT 0200, Australia.

[IHI$^+$] Ishihata H., Horie T., Inano S., Shimizu T., and Kato S.*CAP-II Architecture.* Computer Based Systems Lab. Kawasaki. Fujitsu Laboratories Ltd. 1015 Kamikodanaka, Nakahara-ku, Kawasaki 211, Japan.

[MFL$^+$91] McBryan O. A., Frederickson P. O., Linden J., Schüller A., Solchenbach K., Stüden K., Thole C., and Trottenberg U. (1991) Multigrid methods on parallel computers - A survey of recent developments. *IMPACT Comput. Sci. Engng.* 3: 1–75.

[Mit88] Mitchell W. F. (1988) *Unified Multilevel Adaptive Finite Element Methods For Elliptic Problems.* PhD thesis, Department Of Computer Science, University Of Illinois at Urbana-Champaign, Urbana, IL. Technical Report UIUCDCS-R-88-1436.

[Mit89] Mitchell W. F. (December 1989) A comparison of adaptive refinement techniques for elliptic problems. *ACM Trans. Math. Software* 15(4): 326–347.

[Mit92] Mitchell W. F. (January 1992) Optimal multilevel iterative methods for adaptive grids. *SIAM J. Sci. Stat. Comput* 13(1): 146–167.

[Rüd92] Rüde U. (May 1992) Data structures for multilevel adaptive methods and iterative solvers. Technical Report I-9217, Institut für Informatik, TU München. Copy found in ftp: capser.cs.yale.edu, dir: mgnet/papers/Ruede, file: data_struct.*.

[Rüd93a] Rüde U. (1993) Data abstraction techniques for multilevel algorithms. In *Proceedings of the GAMM-Seminar on Multigrid Methods.* Institut für Angewandte Analysis und Stochastik. Copy found in ftp: capser.cs.yale.edu, dir: mgnet/papers/Ruede, file: programming.*.

[Rüd93b] Rüde U. (1993) *Mathematical and computational techniques for multilevel adaptive methods.* SIAM.

[Sta95] Stals L. (1995) *Parallel Multigrid On Unstructured Grids Using Adaptive Finite Element Methods.* PhD thesis, Department Of Mathematics, Australian National University, Canberra, 0200, Australia.

Figure 5 Example of adaptive refinement. In Figure a) I_1 and I_2 are two interface edges while B_3 and B_4 are two base edges. Note that we have not marked all of the base and interface-base edges to help to reduce the clutter. The base edge B_3 must be split before the interface-base edge I_1. When B_3 is split the interface-base edge I_1 is updated to a base edge B_1 as shown in b). The edge B_1 can now be split to give the final grid shown in c).

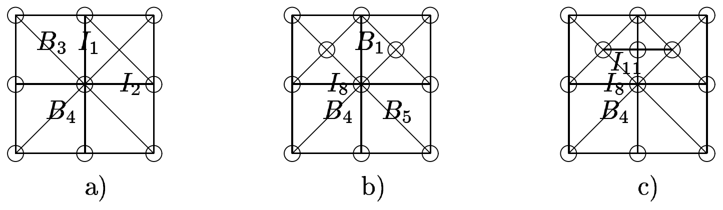


Figure 6 Example grid spread over four processor after three levels of refinement. The areas which are not shaded are shared by two or more processors.

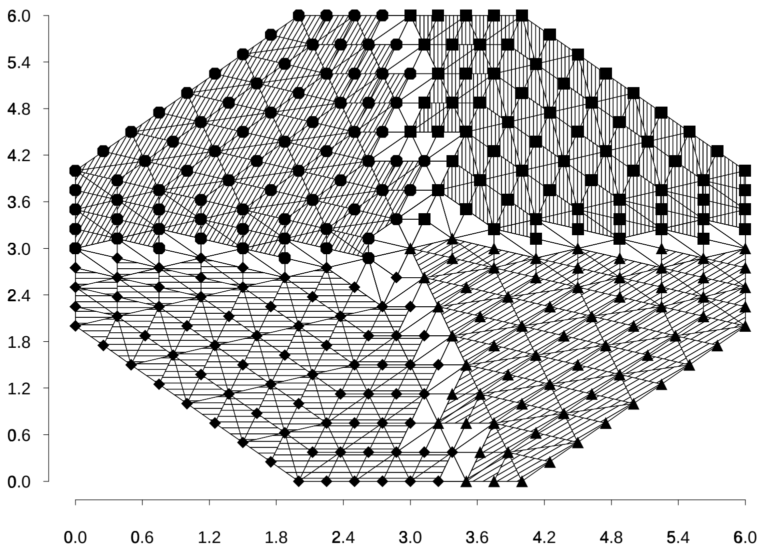


Figure 7 Resulting grid after four levels of adaptive refinement. This example was run a network of workstations using eight processors.

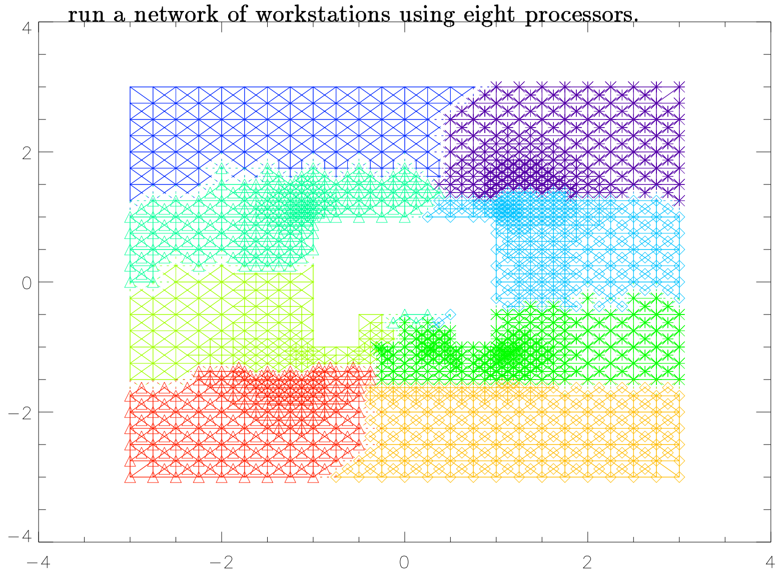
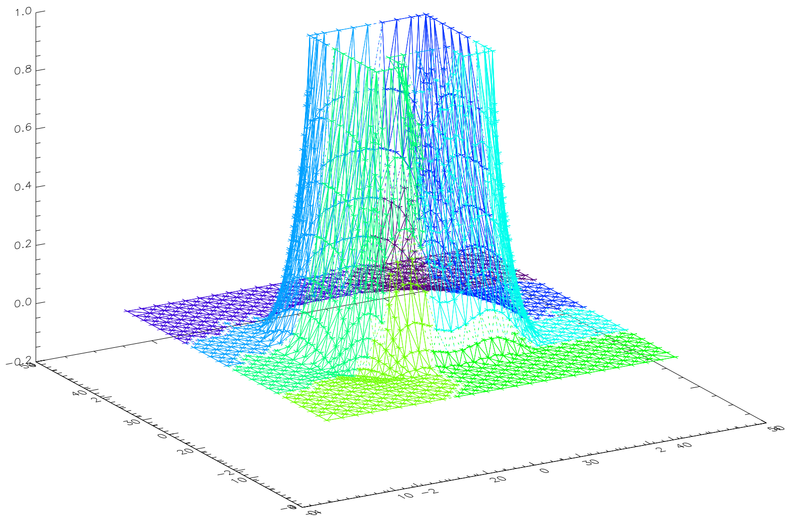


Figure 8 Result after four levels of adaptive refinement.


1.0
0.8
0.6
0.4
0.2
0.0
−0.2