Parallel Adaptive Multilevel Methods with Full Domain Partitions

William F. Mitchell*

Mathematical and Computational Sciences Division
National Institute of Standards and Technology
Gaithersburg, MD 20899

Received 15 November 2003, revised 30 November 2003, accepted 2 December 2003
Published online 3 December 2003

Adaptive multilevel methods are methods for solving partial differential equations that combine adaptive grid refinement with multigrid solution techniques. These methods have been shown to be very effective on sequential computers. Recently, a technique for parallelizing these methods for cluster computers has been developed. This paper presents an overview of a particular adaptive multilevel method and the parallelization of that method via the full domain partition.

1 Introduction

This paper describes an approach for the fast solution of partial differential equations (PDEs). The individual aspects of the approach have been published previously [8, 9, 10, 11], but this paper brings them together in one source for the first time. Consequently, many of the details are omitted in this paper. The interested reader is referred to the references.

The objective is to develop a method that uses modern fast solution techniques on parallel computers. We consider two-dimensional second-order scalar linear elliptic partial differential equations of the form

$$\begin{aligned} -\frac{\partial}{\partial x}(p\frac{\partial u}{\partial x}) \; - \; \frac{\partial}{\partial y}(q\frac{\partial u}{\partial y}) + ru \; &= \; f \text{ in } \Omega \\ u \; &= \; g \text{ on } \partial\Omega \end{aligned} \tag{1}$$

where p, q, r, f and g are functions of x and y, and Ω is a bounded, connected, polygonal domain in $\Re^2$. This class of model problems is frequently used as a numerical test bed for new methods. However, a solver for this class of problems can often be used as a core elliptic solver for more interesting classes of problems, such as time dependent problems, nonlinear problems and systems of equations.

The target computational environment for the parallel methods described in this paper is a moderate sized (10's of processors) cluster of PCs or workstations. This is a distributed memory machine in which one uses message passing to communicate between the processors. This type of architecture is becoming common place in smaller research labs, and is sort of a "poor man's supercomputer". One of the main differences between a small inexpensive cluster and the massively parallel supercomputers found in large labs is the interconnection network. A large part of the expense of the supercomputers is the network that allows the processors to communicate relatively quickly. In contrast, the network on a cluster is slow, both in latency (the time to start a message) and bandwidth (the maximum number of bytes per second). The large latency can be particularly damaging to algorithms that send small messages with little computation between them, because the program execution time will be dominated by the latency of sending messages. Therefore, it is important in this environment to develop algorithms that send messages as infrequently as possible.

* email: william.mitchell@nist.gov.

The numerical methods we consider are adaptive multilevel methods. These types of methods were shown to be very effective for elliptic problems using sequential computers in the 1980's [1, 5, 7, 13]. Since then, much of the research has been toward parallelizing them. These methods are particularly challenging to parallelize due to the irregular structure of adaptively refined grids and the varying distances that arise in a multigrid cycle.

By an adaptive multilevel method we mean a method that combines a finite element method with adaptive mesh refinement and a multigrid solution technique (see Fig. 1). In the case of a parallel algorithm, it also includes a load balancing step, since otherwise the adaptive refinement would cause some processors to be responsible for more of the grid than others. One begins with a very coarse grid (possibly just fine enough to define the domain), and cycles through three phases until some criterion for termination is met (for example, an error estimate is small enough).

```
start with very coarse mesh
repeat
  if the load is too far out of balance
    repartition grid
    redistribute data
  endif
  adaptive mesh refinement
  multigrid cycles
until termination criterion is met
```

Fig. 1 High level algorithm for the parallel adaptive multilevel method.

The first phase is load balancing. When performed before refinement, it is called predictive load balancing, because you are predicting how much refinement will occur in different regions, and what redistribution of elements will balance the load after that refinement. It is crucial that load balancing be very fast so that it does not dominate the "real" work.

The second phase is adaptive grid refinement. The goal is to concentrate the effort into the regions where it will do the most good by placing a finer grid in those regions. This is achieved by computing a local error estimate for each element, and refining those elements with the largest error estimates, in an attempt to equilibrate the local error throughout the grid.

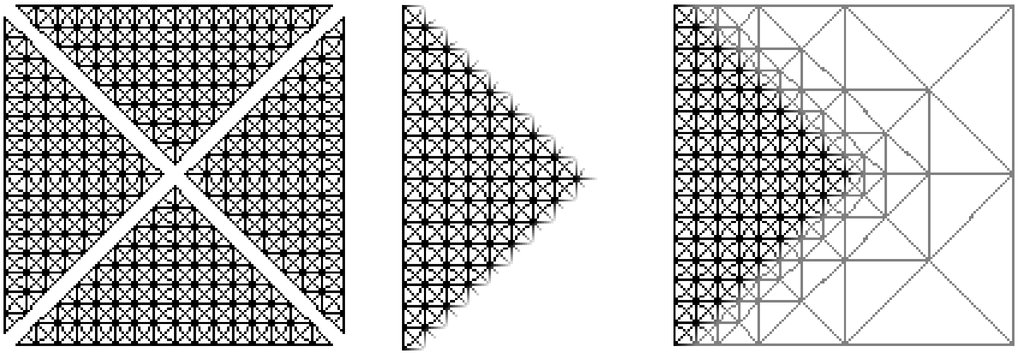
The third phase is the solution of the PDE on the current grid. This includes the discretization via finite elements, and the solution of the resulting algebraic linear system. The solution from the previous grid provides a good initial guess, and a sequence of grids related to the adaptive refinement provide a nested sequence of grids for the multigrid method.

The remainder of the paper is organized as follows. In section 2 we define the full domain partition, which forms the basis for the parallelizations in the remainder of the paper. In section 3 we describe the newest node bisection method for adaptive refinement, and parallel adaptive refinement. Section 4 describes a load balancing algorithm that is based on the refinement tree obtained from adaptive refinement. In section 5 we describe the hierarchical multigrid method and its parallelization.

2 Full Domain Partition

In this section we describe the full domain partition, which was first introduced in [8]. It provides the basis for the low-communication parallelization of the adaptive multilevel method.

In the traditional approach to data decomposition, the domain is partitioned into p regions, where p is the number of processors, and the data is distributed across the processor memories accordingly. In addition, each processor contains "shadow" (or "ghost") data which is a copy of neighboring data that the processor needs to use (see Fig. 2). This eliminates the need to communicate every time a piece of neighboring data is needed, but it does require a communication step whenever the neighbor changes the value, if this processor will use it before it changes again. For example, with a multigrid algorithm a communication step is needed after the computation on each level of the multigrid cycle. This is too often (i.e., not enough computation between communication steps) on parallel systems with a large start-up time for sending messages, such as clusters of PCs.



(a) (b) (c)

Fig. 2 (a) A uniform grid partitioned for four processors. (b) The grid seen by one processor with the traditional data distribution. (c) The grid seen by one processor with the full domain partition. Shadow nodes are shown in gray.

In the full domain partition, the domain is also partitioned and the data distributed across the processors with shadow copies of neighboring data. But, in addition each processor contains more shadow data to provide a grid that covers the full domain. See Fig. 2. The additional elements come from the coarser grids. It is as if each processor performed adaptive refinement of the global grid with refinement restricted to its own partition plus refinements outside its partition only when they are needed to maintain compatibility (see section 3). In fact, that is how the full domain partition is generated.

For two dimensional problems, the amount of extra data is $O(\sqrt{N/p})$ where N is the number of nodes in the global grid and p is the number of processors, and is about twice as much as the shadow data in the traditional approach [8].

Fig. 3 shows an example of the full domain partition of an adaptively refined grid partitioned for four processors. The top of the figure shows the global grid separated into four partitions. Each partition is assigned to a processor. The remainder of the figure shows the full domain partition grid on each processor.

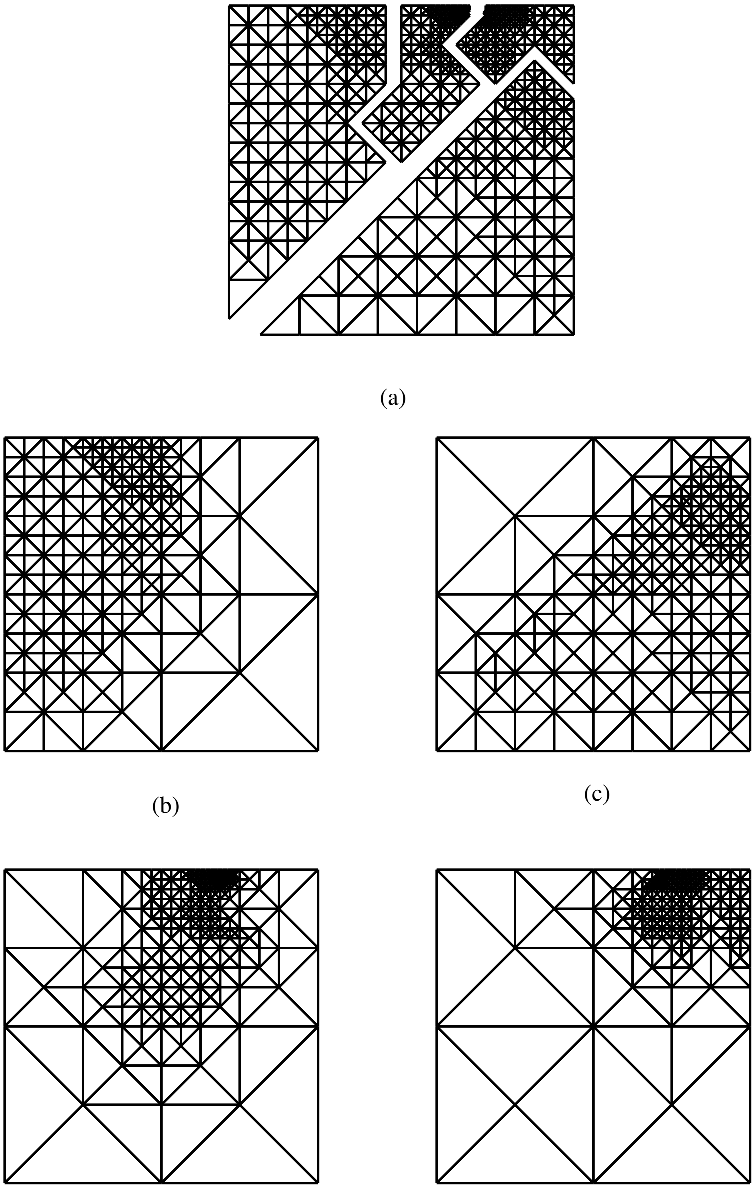
3 Adaptive Refinement

The objective of adaptive grid refinement is to generate a grid that is coarse where the solution is well behaved, fine near singularities, boundary layers, etc., and has a smooth transition between the coarse and fine parts. Such a grid can dramatically reduce the number of nodes needed to obtain an accurate solution for marginally smooth problems, and can recover the optimal order of convergence for non-smooth problems.

Many approaches to performing adaptive refinement have been presented over the years. In this section we first give a brief description of the method known as newest node bisection. Further details of this method can be found in [6]. We then show how to parallelize adaptive refinement algorithms by using the full domain partition. Further details of the parallel algorithm can be found in [10].

The general idea of adaptive refinement is to first compute an error indicator for each element and then refine those elements that have the largest error indicators. The computation of error indicators has been the subject of much research, and will not be addressed in this paper. See [15] for a good treatment of error indicators. We consider here the process of refining triangular elements.

The newest node bisection of triangles was first introduced in [14]. A triangle is divided to form two new triangles by connecting one of the vertices, called the peak, to the midpoint of the opposite side, called the base, as in Fig. 4. The new vertex created at the midpoint of the base is assigned to be the peak of the new triangles. As illustrated in Fig. 5, repeated application of this refinement will result in only four triangle shapes, and therefore the angles are bounded away from 0 and π. It can be shown with simple geometry that triangles labeled with the same number have the same shape, and further refinements continue with these shapes.



(a)

(b) (c)

(d) (e)

Fig. 3 The partition of an adaptive grid into four parts, and the full domain partition seen by each processor.

It is important that the final triangulation be compatible, i.e., given any two triangles their intersection is either empty, a common vertex, or a common side. Compatibility can be maintained during the refinement process by dividing pairs of triangles rather than individual triangles. A triangle is said to be compatibly divisible if its base is either the base of the triangle that shares that side or part of the boundary of the domain. If a triangle is compatibly divisible, then divide the triangle and the neighbor that shares the base simultaneously as a pair. If the triangle is not compatibly divisible, then after a single bisection of the neighbor it will be. So in this case, first divide the neighbor by the same process, and then divide the triangle and the neighbor simultaneously, as illustrated in Fig. 6.

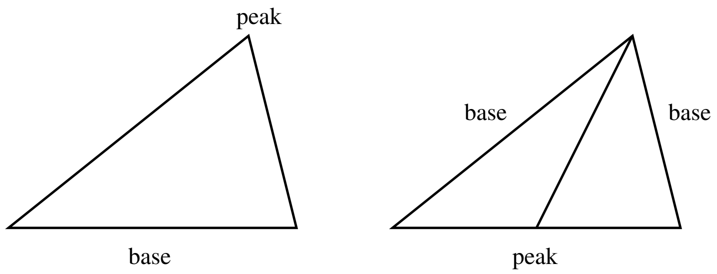


Fig. 4 Newest node bisection of a triangle.

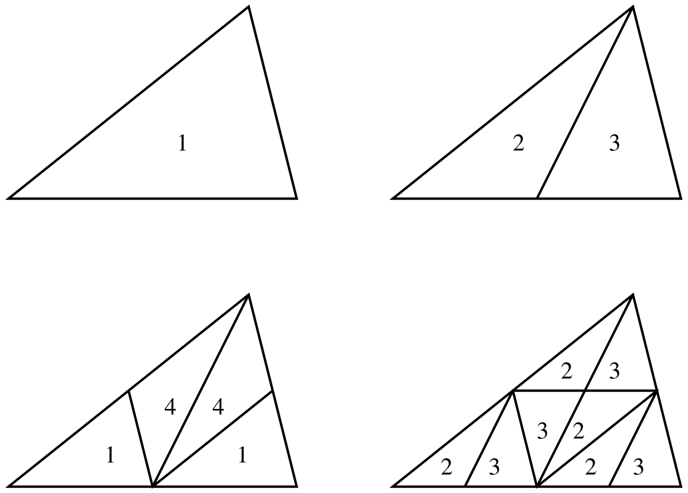


Fig. 5 The four shapes of triangles that arise during repeated application of newest node bisection.

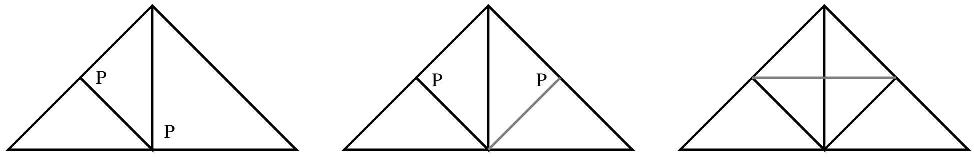


Fig. 6 Maintaining compatibility during refinement. Peaks are labeled P.

We now consider the parallelization of the adaptive refinement algorithm in the context of the full domain partition. This parallelization is applicable to any adaptive refinement method, not just the newest node bisection method.

With the full domain partition, each processor contains a compatible grid covering the entire domain, hence each processor can perform a refinement of that grid independent of the other processors without concern for conflict along the partition boundaries. To create the full domain partition, each processor examines only the triangles in its partition when selecting triangles that have large error indicators. Outside its partition, triangles are only refined for compatibility. This process does not require any communication, thus it performs a large block of computation without communication. A communication step to synchronize the grids occurs after the refinement is complete on each processor. Some triangles outside the processor's partition may be refined (for compatibility), and the owner of any such triangle must be informed so that it can insure that its copy of that triangle is refined. A list of any triangles owned by other processors that were refined by this processor is sent to the other processors. Upon receiving the list, a processor looks to see which triangles it owns, and refines them if they have not already been refined.

Numerical computations were presented in [10] to examine the speedup obtained by performing adaptive refinement in parallel. These results were obtained with a cluster of four 200 MHz Pentium Pro-based computers with 128 MBytes of memory, connected by fast ethernet.

Laplace's equation is solved on the unit square with Dirichlet boundary conditions set such that the solution is

$$1.1786 - 0.1801p + .006q$$

where

$$p(x,y) = x^4 - 6x^2y^2 + y^4$$
$$q(x,y) = x^8 - 28x^6y^2 + 70x^4y^4 - 28x^2y^6 + y^8$$

An example adaptive grid for this problem is shown in Fig. 7. The program is terminated when the grid has approximately 20,000 vertices. Numerical results for this problem are given in Table 1. The wall clock time for the refinement portion of the program is given, and includes the communication time at the end of the refinement phases. The speedup is defined as the time spent on one processor divided by the time spent on p processors. The efficiency is defined as the speedup divided by the number of processors. These results show excellent speedup and efficiency when using a small number of processors.

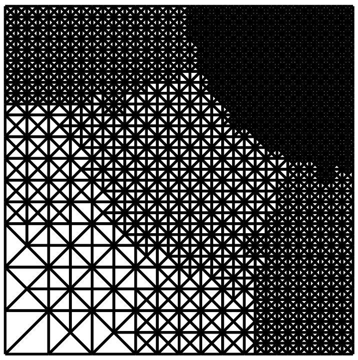


Fig. 7 Sample adaptive grid generated for the adaptive refinement speedup example.

Processors	Time(s.)	Speedup	Efficiency
1	10.49	-	-
2	5.92	1.77	.86
4	2.70	3.89	.97

Table 1 Speedup results for adaptive refinement.

4 Load Balancing

To effectively utilize a parallel computer it is important that the data be distributed over the processors in a balanced manner, so that each processor will complete its work load at approximately the same time, i.e., no processors will sit idle waiting for other processors to complete their work. In a parallel adaptive multilevel method, the load will quickly become unbalanced as more refinement occurs in some partitions than others. Thus in each loop through the phases of the algorithm a load balancing step is performed. This consists of computing a new partition that will balance the load, and redistributing the data among the processors in accordance with this partition.

The load balancing step may occur after the refinement phase (regular load balancing) or before the refinement stage (predictive load balancing). Predictive load balancing implicitly or explicitly estimates what the grid will look like after refinement, and might produce a less balanced load for the solution phase than can be obtained with regular load balancing. However, it has the advantage that the load is better balanced during the refinement phase, and communication costs are reduced by redistributing less data (the data of the coarser grid before refinement).

The goal of the partitioning method is to balance the load while keeping the size of the boundaries between partitions small, to minimize communication. In the context of adaptive multilevel methods, it is also crucial that the algorithm be very fast, so that it does not dominate the time to perform adaptive refinement and multigrid solution. Several methods to determine such a partition have been developed and implemented in Zoltan [3], a library of dynamic load balancing routines.

We describe one such method, the refinement tree based method, which was developed in the context of the full domain partition. Further details of this method can be found in [11].

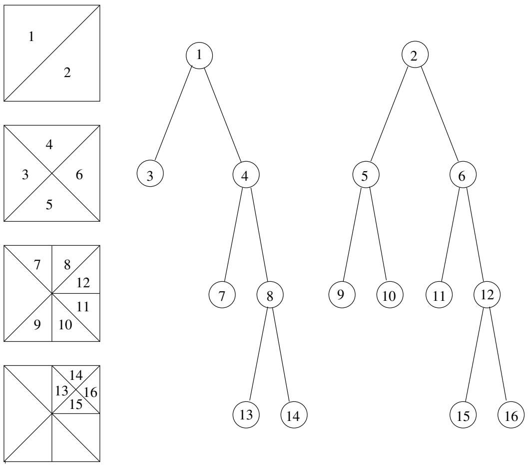


Fig. 8 The correspondence between elements of the grid and nodes of the refinement tree.

The refinement tree is a representation of the refinement process that produced an adaptive grid from an initial grid, as illustrated in Fig. 8. The nodes of the tree correspond to the grid elements that existed at some point during the refinement process. The children of a node correspond to the grid elements that were created when the corresponding element was refined. Leaf nodes correspond to the elements of the final grid. An artificial root node may also be added as the parent of all the initial elements.

The nodes of the tree are weighted with a number that represents the amount of work associated with the corresponding element. Interior nodes may be weighted (for example, to represent the work on a coarse grid of a multigrid algorithm), but usually weights are only applied to the leaf nodes. If all the leaf nodes are given a weight of 1.0, then the resulting partitions will all have the same number of triangles. For predictive load balancing, the error indicator used to guide the adaptive refinement is an appropriate weight, since those elements with a large error indicator will be refined more times than those with a small error indicator.

The refinement tree based partitioning algorithm consists of two phases. In the first phase, every node is labeled with the sum of the weights in the subtree rooted at that node. This is accomplished by a depth first traversal of the tree, and takes $O(N)$ operations where N is the number of nodes in the tree. In the second phase, a truncated depth first traversal of the tree is performed to create the partitions. The desired size of each partition is determined by dividing the summed weight at the root node by the number of partitions, and the partitions are initialized to be empty. During the traversal, the summed weights at the nodes are examined relative to the size of the partition under construction. If it is small enough to be added to the partition without exceeding the desired size, then it is added and the subtree is not traversed. Otherwise, the children are visited and the subtree will be split among two or more partitions. Fig. 9 illustrates this process for a simple grid partitioned into two parts. If there are p partitions and the depth of the tree is $O(\log N)$, one would expect the second phase to require $O(p \log N)$ operations.

In a parallel implementation, the grid is distributed over the p processors. Each processor contains a refinement tree that includes at least the initial grid and the part of the grid belonging to this processor. Parts of the grid belonging to other processors will have been pruned from the tree. Thus the summation of the weights must be

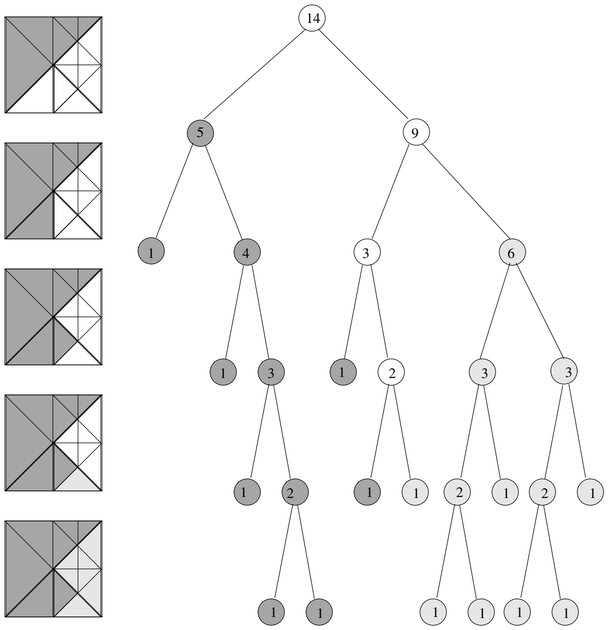


Fig. 9 Assignment of elements to two partitions via the refinement tree partitioning method.

done in a distributed manner. This can be accomplished with two tree traversals and one all-to-all communication step. In the first tree traversal, the weights are summed for nodes that belong to this processor. Leaf nodes that are pruning points are given the weight 0.0, but otherwise the summation occurs as usual. The processors then exchange information to provide the summed weights for the pruning points, by each processor sending what it has as the summed weight for each node that is a pruning point on a different processor. Note that a processor may receive contributions for a pruning point from more than one processor, and the sum of these is the correct summed weight for that node. Now with the summed weight available for the pruning points, a second traversal is performed to obtain the correct summed weights for the entire tree.

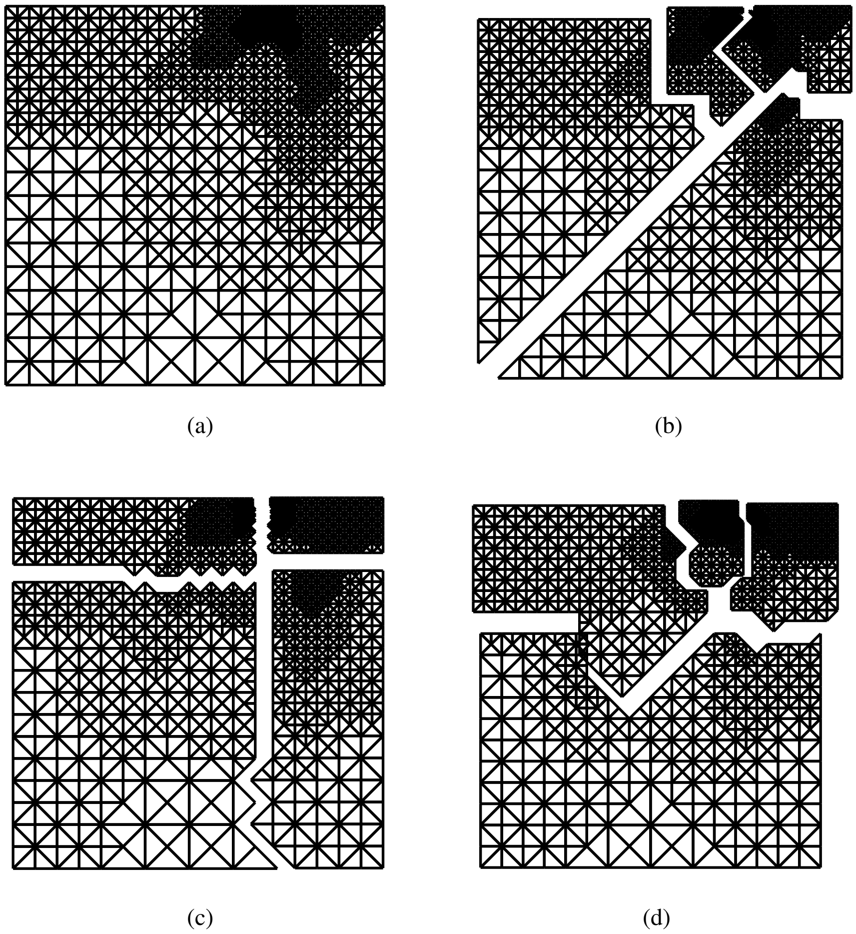
Each processor now has sufficient information to perform the second phase of the algorithm independently. Without further communication, all processors will obtain the same partition, except for details within parts of the grid that the processor does not have. These details are not needed since all the processor needs to know is the new assignment for the elements it currently has so that it can send those elements to the new owner during the redistribution of data.

For a grid with N elements, if each processor has $O(N/p)$ elements and $O(\sqrt{N/p})$ shadow elements, the expected number of operations on each processor is $O(N/p)$.

Numerical experiments were performed to compare the refinement tree partitioning algorithm (designated RTK) with the recursive coordinate bisection (designated RCB) algorithm from Zoltan [3] and a multilevel diffusive method (designated Metis) from ParMETIS [4]. This experiment was conducted on a cluster of eight 400 MHz Pentium II based PCs connected by a fast ethernet switch. Further details of the experiment can be found in [11].

Laplace's equation was solved on the unit square with Dirichlet boundary conditions with a singularity at (0.7,1.0). The program was terminated when there were 64,000 vertices in the grid. Repartitioning occured seven times during the load balance/refine/solve loop. Runs were made using from 2 to 8 processors. Figure 10 shows an example grid with approximately 1000 vertices and the four-set partitions produced by each of the three methods.

Table 2 shows the execution time of the partitioning algorithms. This is wall clock time and includes all of the time spent by the partitioner during the run, including communication. It is easily seen that RTK is slightly faster than RCB and two to three times faster than ParMETIS.



(a) (b) (c) (d)

Fig. 10 (a) Sample grid, and partitions for four processors from (b) RTK, (c) RCB, and (d) Metis.

Processors	RTK	RCB	Metis
2	1.19	0.90	2.45
3	0.84	0.92	1.99
4	0.74	0.98	1.85
5	0.71	0.72	1.84
6	0.63	0.71	1.84
7	0.60	0.85	1.90
8	0.64	0.97	2.04

Table 2 Time for partition (sec.).

Table 3 shows a measure of the quality of the partitions, the number of cut edges in the final grid. A cut edge corresponds to a triangle side that is shared by triangles in different partitions. The number of cut edges reflects the volume of data that must be communicated during the solution process. These results show no significant difference between the methods in terms of quality of partition.

5 Multigrid

The multigrid method has been established as perhaps the most efficient method for solving the linear systems that arise from the discretization of differential equations. The popularity of this method can be attributed to the

Processors	RTK	RCB	Metis
2	639	300	207
3	491	669	414
4	842	572	713
5	847	860	872
6	1120	1098	1019
7	1204	1212	1145
8	1343	1130	1337

Table 3 Number of cut edges.

fact that it is optimal in the sense that one multigrid cycle can reduce the norm of the error by a factor that is bounded away from 1 independent of N, the size of the linear system, while using only $O(N)$ operations. In this section we present a multigrid method based on the hierarchical basis, and the parallelization of the method in the context of the full domain partition. Further details of the sequential algorithm can be found in [7], and the parallel algorithm in [9].

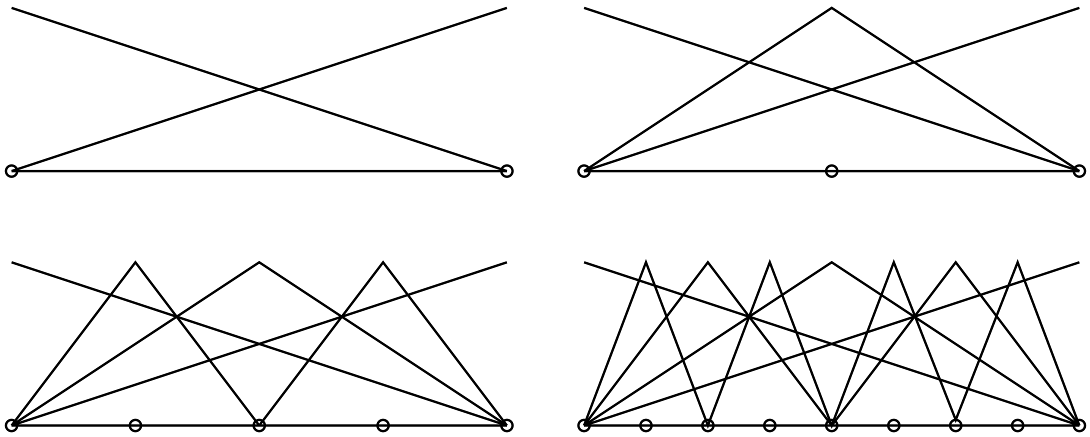


Fig. 11 Construction of the hierarchical basis for linear elements in 1D.

The hierarchical multigrid method uses both the nodal basis and the hierarchical basis. The usual nodal basis, $\{\phi_i\}_{i=1}^N$, for the space of piecewise linear functions on a given grid, T_L, is defined by

- ϕ_i is 1.0 at node i and 0.0 at all other nodes,
- ϕ_i is a linear function over each element,
- ϕ_i is continuous over the whole domain.

In contrast, the hierarchical basis is defined using the family of nested grids, $\{T_i\}_{i=1}^L$, from the refinement process. The hierarchical basis begins with the nodal basis on the initial grid, T_1. As refinement proceeds, with each element division one or more new nodes are added, and for each node a new basis function is defined as a nodal basis on the current grid, but the existing basis functions remain unchanged. Fig. 11 illustrates construction of the hierarchical basis for piecewise linear functions in one dimension.

A 2-level hierarchical basis for grid T_l can also be defined by starting with the nodal basis for the grid T_{l-1} and defining the higher level basis functions as above. It can be shown that conversion between the nodal basis and the 2-level hierarchical basis forms of coefficient vectors and finite element stiffness matrices is easily performed with a small number of operations per node [7].

The sequential hierarchical multigrid algorithm is based on performing a sequence of transformations between nodal and 2-level hierarchical bases. The l^{th} grid in $\{T_i\}$ is obtained by using the first l levels of refinement in the adaptive grid. When working on level l, the points that are in grid $l-1$ are referred to as black points, and those that are in grid l but not in grid $l-1$ are referred to as red points. With the rows and columns ordered such that the black points come before the red points, the equations formed by using a 2-level hierarchical basis can be written

$$\begin{bmatrix} A_{bb} & A_{br}^T \\ A_{br} & A_{rr} \end{bmatrix} \begin{bmatrix} x_b \\ x_r \end{bmatrix} = \begin{bmatrix} b_b \\ b_r \end{bmatrix} \tag{2}$$

where the subscripts b and r correspond to black and red points. (For simplicity of notation, the designation of the associated grid level is omitted.) Note that by definition of the hierarchical basis, the coarse-coarse block, A_{bb}, of the 2-level hierarchical matrix is the nodal matrix of the coarse grid. This suggests the use of

$$A_{bb}x_b \;=\; b_b \;-\; A_{br}^T x_r \tag{3}$$

as the coarse grid problem. Such a formulation can be shown to be equivalent to the standard formulation of the multigrid method in the case of piecewise linear elements. However, it is more easily generalized to other finite element spaces. With this formulation, one cycle of the multigrid algorithm can be described as in Fig. 12.

```
for l = L downto 2
  perform one or more Gauss-Seidel sweeps
  convert to 2-level hierarchical basis of level l
  b_b  <-  b_b  -  A_br^T x_r
end for
solve coarse grid problem on level 1
for l = 2 to L
  b_b  <-  b_b  +  A_br^T x_r
  convert to nodal basis of level l
  perform one or more Gauss-Seidel sweeps
end for
```

Fig. 12 Algorithm for one cycle of the hierarchical multigrid method.

The full domain partition provides a means of adapting this algorithm for parallel computation. Since each processor has a compatible, hierarchical adaptive grid that covers the whole domain, the multigrid algorithm can be run on each processor in parallel. Obviously, some level of communication is necessary, since the individual grids do not contain sufficient resolution over the entire domain. These communication steps occur only twice per multigrid cycle. The first communication step occurs when we reach the coarsest grid, i.e., just before the line “solve coarse grid problem on level 1” in Fig. 12. The second communication step occurs at the end of the cycle, i.e., after the last line in Fig. 12. Thus, again we have a large block of computation between relatively few communication steps.

The key difference between the sequential and parallel versions is the computation of the $A_{br}^T x_r$ term in Eq. 3. None of the processors contain all of the data to compute this term, so the computation is distributed over the processors. The contributions of each processor to this term are communicated in the first communication step, along with the current solution values at the nodes owned by each processor. After this communication step, all processors have the same value for the solution and the modified right hand side at all the owned and shadow nodes. In the second communication step, only the solution at owned nodes is sent to other processors to update the solution at shadow nodes.

Processors						
1	2	4	8	16	32	64
.093	.093	.094	.099	.098	.101	.098

Table 4 Multigrid contraction factors for an adaptive grid with about 60000 vertices.

The computational results of the parallel algorithm are not exactly the same as those of the sequential algorithm, because the contribution of other processors to the $A_{br}^T x_r$ term do not arrive until the middle of the cycle. During the first half of the cycle, the values from other processors are generally from the previous cycle, and thus not quite correct. However, only a small part of $A_{br}^T x_r$ comes from other processors, and it is only off by the amount that x_r changed, so this has only a small effect on the rate of convergence of the multigrid method. An experiment was performed [9] with Laplace's equation to examine the effect of this lag on the rate of convergence. Table 4 shows the observed contraction factors (i.e., the factor by which the error is reduced by one cycle) when using between 1 and 64 processors. It is observed that the error is reduced by approximately a factor of 10 independent of the number of processors.

computer (vertices)	nproc	time (s)	scaled speedup	scaled efficiency
PPro	1	3.16	–	–
(16K	2	3.79	1.67	.83
per proc)	4	3.13	4.04	1.01
	8	3.76	6.72	.84
SP2	1	14.09	–	–
(64K	2	15.71	1.79	.90
per proc)	4	16.62	3.39	.85
	8	17.89	6.30	.79
	16	20.83	10.82	.68
	32	25.31	17.81	.56

Table 5 Multigrid speedup results for scaled problem size.

Experiments were also performed [9] to examine the speedup obtained by parallel computing. Two parallel environments were examined: a cluster of Pentium Pro-based computers connected by fast ethernet, and an IBM SP2. On the SP2 we used the ethernet connection between processors, not the high speed switch, to study the performance in a high latency environment. Computations were performed with a scaled grid size, i.e., the size of the grid grows proportional to the number of processors. In the ideal situation, the computation time would remain constant as the number of processors (and size of the grid) increases. On the PPro cluster, a grid with 16,000 nodes per processor was used, and on the SP2 the grid contained 64,000 nodes per processor. Table 5 shows the results of these computations. We observe parallel efficiencies above 50% throughout the range of number of processors considered, with the efficiency usually above 75%.

6 Conclusion

This paper presented an overview of a parallel adaptive multilevel method based on the full domain partition. It gave a general description of each aspect of the algorithm, while leaving the detailed description to the references. Numerical results were presented to demonstrate the effectiveness of the algorithms. These results show that the parallel algorithms presented in this paper can achieve high parallel efficiency on small cluster computers.

The methods described in this paper have been implemented in a public domain Fortran 90 code, PHAML, which can be downloaded from [12]. PHAML includes several example problems to demonstrate its use. It has also been demonstrated that existing codes can be parallelized relatively easily by using the full domain partition. The program PLTMG, which has been around for nearly two decades, was recently parallelized [2] by techniques nearly identical to those described in this paper.

Acknowledgements The mention of specific products, trademarks, or brand names is for purposes of identification only. Such mention is not to be interpreted in any way as an endorsement or certification of such products or brands by the National Institute of Standards and Technology. All trademarks mentioned herein belong to their respective owners.

References

[1] R. E. Bank and A. H. Sherman, An Adaptive Multilevel Method for Elliptic Boundary Value Problems, Computing **26**, 91–105 (1981).
[2] R. E. Bank and M. Holst, A New Paradigm for Parallel Adaptive Meshing Algorithms, SIAM J. Sci. Comput. **22**, 1411–1443 (2000).

[3] K. Devine, B. Hendrickson, E. Boman, M. St. John, C. Vaughan and W. F. Mitchell, Zoltan: A Dynamic Load-Balancing Library for Parallel Applications, User's Guide, Tech. Rep. SAND99-1377, Sandia National Laboratories, Albuquerque, NM (2000).

[4] G. Karypis, K. Schloegel, and V. Kumar, ParMETIS: Parallel Graph Partitioning and Sparse Matrix Ordering Library, Version 2.0, http://www.cs.umn.edu/karypis/metis/metis.html.

[5] S. F. McCormick, Multilevel Adaptive Methods for Partial Differential Equations, vol. 6 of Frontiers in Applied Mathematics (SIAM, Philadelphia, 1989).

[6] W. F. Mitchell, Adaptive Refinement for Arbitrary Finite-element Spaces with Hierarchical Bases, J. Comp. Appl. Math. **36**, 65–78 (1991).

[7] W. F. Mitchell, Optimal Multilevel Iterative Methods for Adaptive Grids, SIAM J. Sci. Stat. Comput. **13**, 146–167 (1992).

[8] W. F. Mitchell, The Full Domain Partition Approach to Distributing Adaptive Grids, Applied Numerical Mathematics **26**, 265–275 (1998).

[9] W. F. Mitchell, A Parallel Multigrid Method Using the Full Domain Partition, Elec. Trans. Num. Anal. **6**, 224–233 (1998).

[10] W. F. Mitchell, The Full Domain Partition Approach to Parallel Adaptive Refinement, in: Grid Generation and Adaptive Algorithms, IMA Volumes in Mathematics and its Applications Vol. 113 (Springer-Verlag, 1998), pp. 151–162.

[11] W. F. Mitchell, A Refinement-Tree Based Partitioning Method for Adaptively Refined Grids, in: Proceedings of the Tenth SIAM Conference on Parallel Processing for Scientific Computing (SIAM, Philadelphia, 2001).

[12] W. F. Mitchell, PHAML, http://math.nist.gov/phaml.

[13] U. Rüde, Mathematical and Computational Techniques for Multilevel Adaptive Methods, vol. 13 of Frontiers in Applied Mathematics (SIAM, Philadelphia, 1993).

[14] E. G. Sewell, Automatic Generation of Triangulations for Piecewise Polynomial Approximations, Ph. D. thesis, Purdue University, West Lafayette, IN (1972).

[15] R. Verfürth, A Review of a posteriori Error Estimation and Adaptive Mesh-Refinement Techniques (Wiley-Teubner, New York, 1996).