PARED: a Framework for the Adaptive Solution of PDEs

Jose G. Castaños and John E. Savage
Department of Computer Science
Brown University
{jgc,jes}@cs.brown.edu

Abstract

We describe our experience using PARED*, an object oriented system for the adaptive solution of PDEs in a distributed computing environment.* PARED *handles selective mesh refinement and coarsening, mesh repartitioning for load balancing and interprocessor mesh migration.* PARED *is an object-oriented system that runs on distributed memory parallel computers such as the IBM SP and network of workstations. In this paper, we report on the use of* PARED *to solve two- and three-dimensional PDEs. We show that our object-oriented technology provides great flexibility with a small overhead to support the highly desirable adaptive features of* PARED.

1. Introduction

Adaptive finite element methods are a collection of techniques for the numerical solution of PDEs that are particularly effective on problems with disparate scales or moving physical phenomena [11]. By focusing the available computing resources on regions of high relative error, the use of adaptive meshes has the potential of producing large computational and storage savings but at the price of increasing the sophistication of codes and algorithms. Adaptive computation requires a tight coupling between a mesh generator and a solver. In a parallel environment, the local adaptation of the mesh produces imbalances in the work assigned to the processors. Because of the irregular load requirements of parallel adaptive computation, a mesh must also be dynamically repartitioned and migrated between processors at runtime.

PARED [2] is an integrated system for the parallel adaptive solution of PDEs. It supports the local refinement and coarsening of unstructured two- and three-dimensional meshes, and the dynamic repartitioning and load balancing of the work. PARED is an object-oriented system in which all the support code is written in C++. Our system runs on distributed memory machines in which processing nodes communicate using MPI [9]. Our design supports a dynamically changing environment. Elements and vertices (and associated equations and unknowns) migrate between processors to balance the workload. References to remote elements and vertices are updated as new elements or vertices are created, deleted or moved to a new processor. The complexity of this approach is hidden by the use of a global object space where remote object communication is facilitated by the use of proxies. Cached proxies are also used to reduce latency and communication overhead.

2. The Adaptive FEM Problem

The finite element method (FEM) divides a given domain Ω into a set of non-overlapping simple shapes Ω_i (called *elements*) such as triangles and quadrilaterals in 2D and tetrahedrons and hexahedrons in 3D. The set of elements and their corresponding vertices form a mesh M. To approximate the solution of a continuous function defined over Ω the FEM solves a system of linear equations obtained from M and the corresponding boundary conditions. The rate of convergence and quality of

the solutions provided by the FEM depends heavily on the number, size and shape of the mesh elements. For a given shape, the approximation error increases with element size (h), which is measured by the length of its longest edge.

The goal of adaptive computation is to optimize the computational resources used in the simulation which can be achieved by refining a mesh to increase its resolution on regions of high relative error in static problems or by refining and coarsening the mesh to follow physical anomalies in transient problems [18]. The adaptation of the mesh can be performed by changing the order of the polynomials used in the approximation (p-refinement), by modifying the structure of the mesh (h-refinement), or a combination of both (hp-refinement). Although it is possible to replace an old mesh with a new one containing smaller elements, most h-refinement algorithms divide each element in a selected set of elements from the current mesh into two or more nested subelements.

Error estimates are used to determine regions where adaptation is necessary. These estimates are obtained from previously computed solutions of the system of equations. In a parallel environment mesh adaptation may produce imbalances in the assigned to processors. Thus, efficient use of resources may require that elements and vertices be reassigned to processors at runtime.

Typical static FEM computations produce a partition of the mesh that minimizes the number of interprocessor communications while assigning a similar amount of work to each partition in a preprocessing step before starting the simulation. (This task is related to graph partitioning [5], an NP-complete problem for which many heuristics have been developed [17].) A portion of the mesh is then assigned to each processor. This approach is not sufficient in a dynamic environment that constantly modifies the load of each processor.

Any system [8, 16, 19] for the adaptive solution of PDEs must integrate subsystems for solving equations, adapting a mesh, finding a good assignment of work to processors, migrating portions of a mesh according to a new assignment, and handling interprocessor communication.

3. PARED: an Overview

PARED is a system of the kind described in the last paragraph. It provides a number of standard iterative solvers such as Conjugate Gradient and GMRES and preconditioned versions thereof. It also provides refinement and coarsening of meshes, algorithms for adaptation, graph repartitioning using standard techniques and our own *Parallel Nested Repartitioning* (PNR) [2], and work migration. PNR is a hierarchical procedure for rebalancing the work that uses the refinement history to repartition the mesh. It gives partitions with a quality comparable to those provided by standard methods such as Recursive Spectral Bisection (RSB) [17] but with a much lower migration cost.

3.1. Initial Partition of the Mesh

PARED partitions the mesh *by elements*. Every element is assigned to one processor and mesh vertices are shared if they are adjacent to elements located on different processors. By using element partitioning, the local element matrices can be computed in processors with no communication.

To start a numerical simulation PARED loads the initial mesh M^0 into a distinguished processor called the *coordinator*. The coordinator creates the dual graph $G(V, E)$ of the mesh, that is, a graph with one vertex $v_i \in V$ for every element $\Omega_i \in M^0$ and an edge $e_{i,j} \in E$ if two elements $\Omega_i, \Omega_j \in M^0$ are adjacent. The graph G is assumed to be small enough to be partitioned using a variety of serial graph partitioning algorithms on G drawn from the Chaco library [7] including Multilevel-KL.

After partitioning G, the coordinator distributes M^0 to all the processors to start the simulation. The coordinator maintains a copy of G that is used later to repartition the mesh.

3.2. Mesh Refinement

Based on an adaptation criterion, each processor adapts the mesh using a local h-refinement algorithm such as Rivara's longest edge bisection of (triangular or tetrahedral) unstructured meshes [13, 14]. This is a recursive procedure that in two

dimensions splits each triangle Ω_i from a set of triangles selected for refinement by adding an edge between the midpoint of its longest side and the opposite vertex. The refinement propagates to adjacent triangles to maintain the conformality of the mesh. In three dimensions, a tetrahedron is bisected by inserting a triangle between the midpoint of its longest edge and the two vertices not included in that edge.

Starting from the initial mesh M^0 the refinement procedure creates a sequence of nested meshes $M^0, M^1, \ldots M^t$ where every mesh $M^l, 0 < l \leq t$ is obtained from M^{l-1} by refining and coarsening some of its elements. For every element $\Omega_i \in M^0$ the adaptation algorithm creates a forest of trees τ_i of elements rooted at Ω_i. In PARED all the elements that result from refining Ω_i are assigned to the same processor as Ω_i. Because an element does not get destroyed when refined, the mesh is easily coarsened by replacing all the children of a refined element by their parent.

The refinement of the mesh can require synchronization between neighboring processors to maintain the conformality of the mesh across processor boundaries. PARED incorporates a new parallel h-refinement algorithm that insures that the mesh is conformant and generates the same refined meshes as in the serial algorithm. It also insures that each mesh element on a processor boundary has access to neighboring elements on other processors via their proxies. The details of our parallel refinement algorithm are discussed in [3].

3.3. Load Balancing

After the adaptation phase, PARED determines if a workload imbalance exists due to increases and decreases in the number of mesh elements on individual processors. If so, it invokes a procedure to decide how to repartition mesh elements between processors. In many physical problems in which the adaptive process is used to adjust the resolution of the mesh as the simulation evolves, the number of elements in the refined mesh M^t is much larger than the number of elements in the initial mesh M^0. Thus, although it is possible to use a serial graph partitioning algorithm to partition and distribute M^0, it is not always feasible to use the same serial partitioning algorithm to rebalance the work of refined meshes.

To produce high quality parallel partitions of large graphs is very difficult. For example, geometric graph partitioning methods [10] that use coordinate information are scalable but do not always generate good partitions. Spectral methods [12] are not practical for large graphs because they do not provide good speedups. The Kernighan-Lin heuristic that is used in multilevel algorithms is P-complete [15] and does not parallelize.

PARED uses an alternative repartitioning procedure [1] to avoid these problems that operates on the graph G described above associated with the initial coarse mesh M^0. Each vertex v_i in G is assigned a weight equal to the number of unrefined elements in its associated tree τ_i. Each edge (v_i, v_j) in G is assigned a weight equal to the number of edges between unrefined elements in τ_i and τ_j. Dual graph vertex and edge weights represent computational intensity and communication cost respectively. Each processor sends to the coordinator the new weights associated with G. The coordinator uses these weights to compute a new partition of the mesh. Because G is assumed to be a relatively small graph, it is partitioned with a serial algorithm.

Many of the heuristics designed for graph partitioning can also be used in graph repartitioning. Unfortunately, when these heuristics are applied to slightly different problems they can generate very different results. For example, standard graph partitioning algorithms such as Multilevel-KL or RSB usually compute a new distribution of the adapted mesh that is very different from the current one and require a large movement of elements and vertices between processors. We have developed new techniques that greatly reduce the cost of migration [4].

Because there are many more ways to partition the adapted mesh M^t than to partition G, a good partition of G does not necessarily imply a good partition of M^t. However, our many experiments comparing partitions of M^t and those obtained by partitioning G and then projecting these partitions to M^t show that the partitions obtained in the two cases have similar quality [4].

In PARED G is a dynamic graph that, although initially created from the mesh M^0, can also evolve over time. The local refinement of the mesh can cre-

ate vertices in G that have very high weight relative to other vertices which might lead to the impossibility of creating balanced partitions. PARED can detect this condition and allow the graph G to be expanded by replacing each such vertex by a subgraph. These dynamic graphs also allow PARED to handle problems such as the study of fractures in materials that require the modification of the structure of a mesh.

After the coordinator obtains a partition of the weighted graph G it informs the processors of the elements that need to move. These elements and the corresponding vertices are migrated between the processors. PARED is then ready to resume another round of equation solving, error estimation, mesh adaptation, mesh repartitioning, and work migration.

3.4. Object-Oriented Mesh Representation

PARED uses remote references and smart pointers, two ideas commonly found in object oriented programming, to provide a simple replication mechanism that is tightly integrated with our mesh data structures. In adaptive computation, the structure of the mesh evolves during the computation as elements and vertices are created, destroyed or assigned to different processors. The use of remote references and smart pointers have greatly simplified the creation of dynamic meshes.

Because PARED uses element partitioning, proxies for vertices that are common to mesh elements on different processors are held in each of them. Proxies of a common vertex refer to each other using remote references which are functionally similar to standard C pointers but address objects in different address spaces. When implemented in C++, a remote reference is just an object that consists of a processor number and memory address.

A processor can also use a remote reference to invoke methods on objects located in a remote processor. Method invocations and arguments destined for remote processors are marshaled into a few messages that contain memory addresses of the remote objects. In the destination processor(s), each address is converted to a pointer to an object of the corresponding type through which the method is invoked. Because the different processors are inherently trusted and MPI guarantees reliable communication, PARED does not incur the overhead traditionally associated with distributed object systems.

Smart pointers are used so that proxy objects can be destroyed when there are no more references to them. For example, in PARED vertices are associated with multiple elements. When the reference count of a vertex proxy reaches zero, the proxy is no longer attached to any element located in the processor and can be destroyed. If a vertex proxy is located in an internal boundary between processors, then some processor might have a remote reference to it. In that case, before a proxy is destroyed, it informs the copies in other processors to delete their references to it. This procedure insures that the shared vertex can then be safely destroyed without leaving dangerous dangling pointers referring to it in other processors.

Finally, PARED uses streamed non-blocking communication to hide the complexity and overhead of message passing where each object marshals and unmarshals itself onto a stream. The refinement and migration algorithm have a communication pattern that is different from most scientific code such as a parallel matrix-vector product in which the same set of memory locations are repeatedly exchanged between processors. In the refinement and migration algorithms, it is also difficult for the destination processors to estimate the size of the receiving buffers and the messages can become very large in the migration algorithm if a lot of data movement is required. To overcome these problems, our system uses automatic buffering that divides very large messages into smaller ones.

4. Results

A good static test for our framework is the two-dimensional problem defined by Laplace's equation $\Delta u = 0$ in the square $\Omega = (-1, 1)^2$ with the following Dirichlet boundary condition:

$$g(x, y) = \cos(2\pi(x - y)) \frac{\sinh(2\pi(x + y + 2))}{\sinh(8\pi)}$$

The analytical solution to this problem is known to be $u(x, y) = g(x, y)$ at every point of the domain Ω. This solution is smooth but changes rapidly

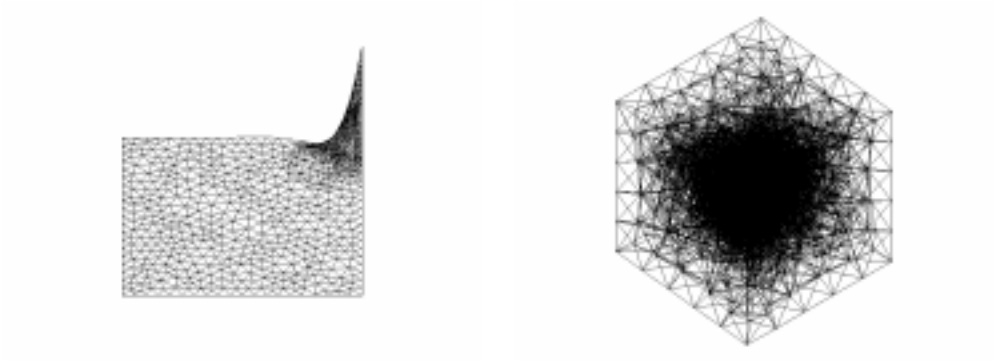

Figure 1. Locally adapted two- and three-dimensional meshes have small elements in the region of high activity.

close to the corner $(1, 1)$. To solve this problem adaptively we generated an unstructured initial mesh with 6394 vertices and 12498 triangles of similar size. We defined a similar problem in 3D and generated an unstructured mesh that contains 2013 vertices and 9540 tetrahedrons. Because the analytical solution of these problems is known, it was possible to select the elements to refine using the L_∞ norm between the computed solution $\hat{u}$ and the real solution u. In this example of a static problem, only refinement was used. The local refinement of these meshes creates a large number of small elements in the high activity corner, as shown in Figure 1.

Figure 2 shows the number of elements and vertices in M^t as a function of successive local adaptations for the 2D and 3D problems. In each level the mesh is partitioned and migrated using the algorithm outlined in Section 3.3, after which apply a Conjugate Gradient solver with a Jacobi preconditioner. Using the L_∞ norm we select a new set of elements that we refine by their longest edge, as explained in Section 3.2. The error criterion was set so that 17 refinement levels were needed for the 2D case and 18 for the 3D case. Note that the final meshes contain more than 2 million elements in the 2D case and about 1.7 million elements in the 3D case.

4.1. Experiments on an IBM SP

We compare the times spent by PARED in each of its four phases of partition, migration, solution and refinement on the 2D and 3D versions of Laplace's equation described above on an IBM SP parallel computer containing four to 32 processors and using MPI. These times for the last 5 adaptation levels are shown in Figures 3 and 4 for the 2D and 3D problems, respectively.

The 2D and 3D problems have very different relative costs. The 2D mesh is an example of a problem that is dominated by its solution time. It requires more than 2,300 CG iterations to reduce the residual below 10^{-10} on the larger meshes. On the other hand, the 3D problem requires little more than 100 iterations to achieve the same error. However, while the solution time dominates that for partitioning and refinement, it no longer dominates the migration time. The solution time will dominate the migration time if either the latter is reduced, a possibility discussed below, or if more time is spent in solution, which would be the case if we used nonlinear polynomial basis functions instead of linear ones.

In these examples most of the refined elements are located on one or a few processors. Most of the refinement time is spent refining elements that are local to a processor; there is very little communication overhead. Thus, the refinement time does not necessarily increase with increasing refinement

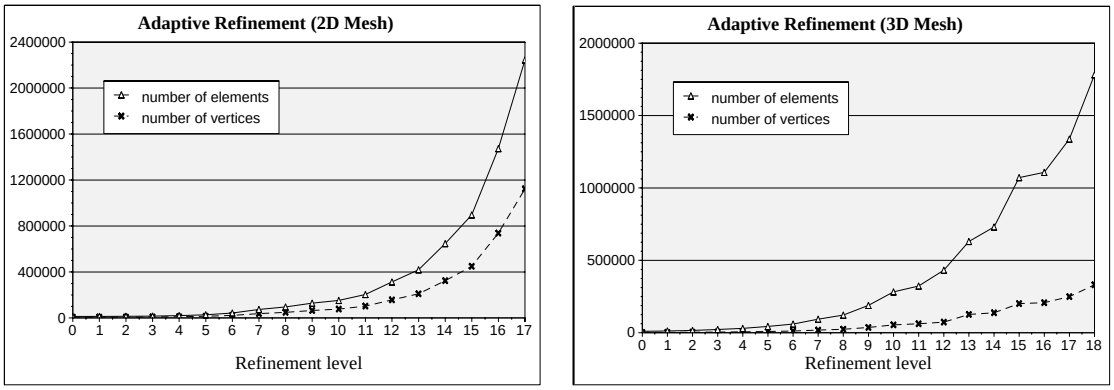

Figure 2. Number of elements and vertices for each refinement level.

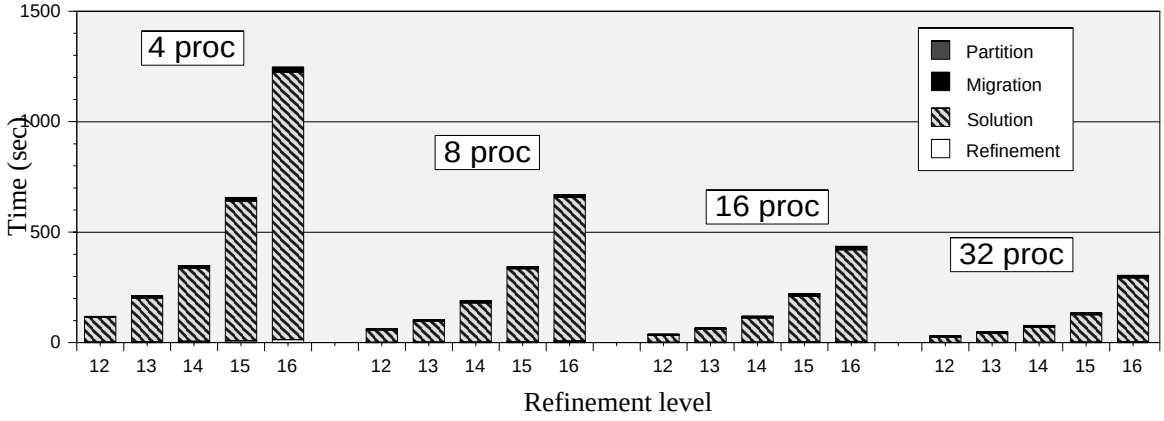

Figure 3. Times for the last five refinement phases of the locally adapted two-dimensional problem on 4, 8, 16, and 32 processors.

levels. Also, the refinement times are comparable in the 2D and 3D cases for meshes of similar sizes. For example, to create 576,324 new triangles at level 16 requires 8 sec. on 4 processors and 2.63 sec. on 32 processors. To create 444,150 new tetrahedrons at level 17 requires 6.4 sec. on 4 processors and 2.24 sec. on 32 processors.

The repartitioning time remains almost constant in both problems as the number of elements increases because a partition is computed from the small weighted graph G obtained from the initial mesh. The partition time slowly increases with the number of processors and varies between 0.28 sec. and 1.68 sec.

As explained in Section 3.3 the use of standard partition algorithms to repartion G generally causes half of the elements to move to a new processor. Our new techniques, mentioned above, cause a dramatic reduction in migration time.

4.2. Experiments on Sun Workstations

We conducted the same experiments using the MPICH [6] communications library on a network of four to 32 Sun Unltra-1 workstations, each having 128MB of memory and connected via a 100Mbps ethernet network. Although, the network of workstations (NOW) is not a controlled environment and

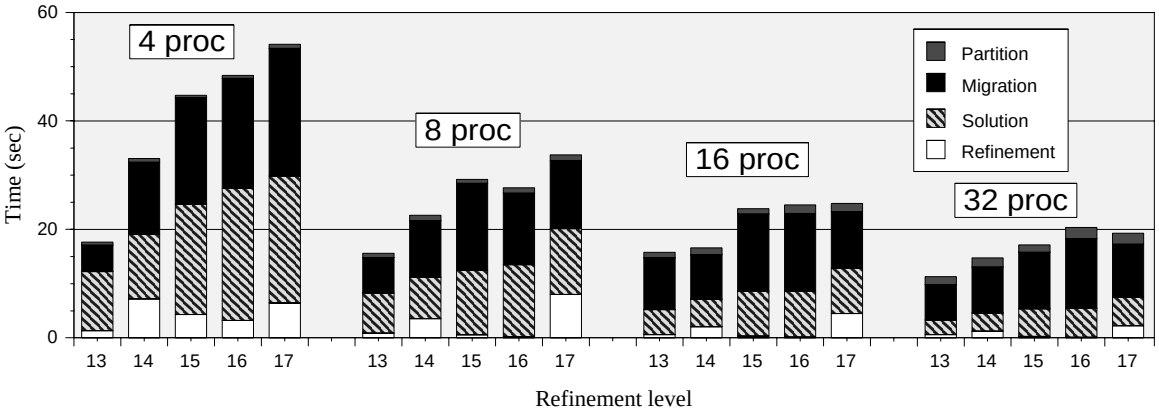

Figure 4. Times for the last five refinement phases of the locally adapted three-dimensional problem on 4, 8, 16, and 32 processors.

does not have the benefit of a fast switch, both of which are characteristic of the SP, the performance achieved on our test problems is not very different from that obtained on the SP. The situation may change if more processors are used.

The NOW has a higher latency which mainly affects smaller messages, such as the the global sums for the Conjugate Gradient. For that reason it is more difficult to obtain speedups in the NOW than on the SP. Also, on the NOW there is a larger potential for network congestion because all the processors communicate through the ethernet.

Figure 5 shows the solution and total times for the 32-processor Sun NOW when normalized by dividing each time by the corresponding times for the same problems on a IBM SP. These results apply to the problem described in Section 4. The normalized times for partition, migration and refinement, which are not shown, are almost constant and range from 1 to 3. The relative solution time for small problems on the NOW is much larger than it is for large problems. This is due to the higher latency of the NOW.

5 Conclusions

We have described computational experiments performed with PARED, an integrated object-oriented system for the adaptive solution of PDEs using the FEM in a distributed computing environment. PARED solves a system of equations, adapts a mesh to regions of high local error, repartitions the mesh in order to rebalance the workload, and migrates the mesh elements and vertices needed to achieve balance. We have examined the time required for each of these four phases on two representative static 2D and 3D problems of large size and have shown that our algorithmic design results in times for the computational overhead that are at worst comparable to the times required to solve the associated system of equations.

Because of its dynamic load balancing properties, PARED is an ideal framework for an environment in which the available computational resources change during a simulation.

References

[1] J. G. Castaños and J. E. Savage. The dynamic adaptation of parallel mesh-based computation. Technical Report CS-96-31, Department of Computer Science, Brown University, October 1996.

[2] J. G. Castaños and J. E. Savage. The dynamic adaptation of parallel mesh-based computation. In *SIAM 7th Symposium on Parallel and Scientific Computation*, 1997.

[3] J. G. Castaños and J. E. Savage. Parallel refinement of unstructured meshes. Technical Report CS-99-10, Department of Computer Science, Brown University, June 1999.

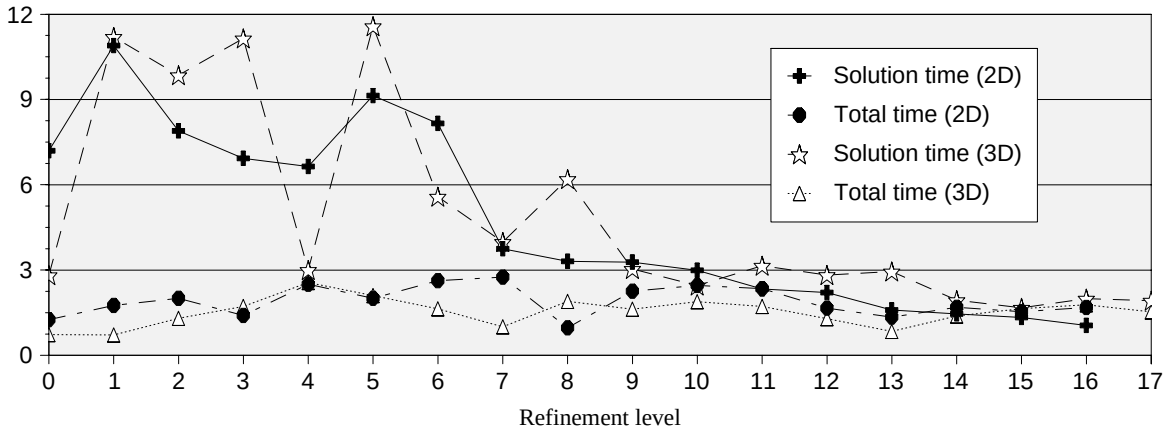

Figure 5. The solution and total times on a Sun NOW normalized by equivalent times on a IBM SP as a function of refinement level.

[4] J. G. Castaños and J. E. Savage. Partitioning unstructured adaptive meshes. Technical Report in preparation, Department of Computer Science, Brown University, 1999.

[5] M. Garey, D. Johnson, and L. Stockmeyer. Some simplified NP-complete graph problems. *Theoretical Computer Science*, 1:237–267, 1976.

[6] W. Gropp and E. Lusk. *User's Guide for MPICH: A portable Implementation of MPI.* Argonne National Lab and Missisipi State University.

[7] B. Hendrickson and R. Leland. The Chaco user's guide, version 2.0. Technical Report SAND94-2692, Sandia National Laboratories, 1995.

[8] M. T. Jones and P. E. Plassmann. Parallel algorithms for the adaptive refinement and partitioning of unstructured meshes. In *Proceedings of the Scalable High-Performance Computing Conference*, 1994.

[9] Message Passing Interface Forum. MPI: A message passing interface standard, 1994.

[10] G. L. Miller, S.-H. Teng, and S. A. Vavasis. A unified geometric approach to graph separators. In *Proceedings of the 31st Annual Symposium on Foundations of Computer Science*, pages 538–547, 1991.

[11] W. F. Mitchell. A comparison of adaptive refinement techniques for elliptic problems. *ACM Transactions on Mathematical Software*, pages 326–347, 1989.

[12] A. Pothen, H. D. Simon, and K.-P. Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM Journal of Matrix Analysis*, 11(3):430–452, 1990.

[13] M. C. Rivara. Selective refinement/derefinement algorithms for sequences of nested triangulations. *International Journal for Numerical Methods in Engineering*, 28:2889–2906, 1989.

[14] M. C. Rivara. A 3-D refinement algorithm suitable for adaptive and multi-grid techniques. *Communications in Applied Numerical Methods*, 8:281–290, 1992.

[15] J. E. Savage and M. Wloka. Parallelism in graph partitioning. *Journal of Parallel and Distributed Computing*, 13:257–272, 1991.

[16] M. S. Shephard, J. E. Flaherty, H. L. DeCougny, C. Ozturan, C. L. Bottasso, and M. Beall. Parallel automated adaptive procedures for unstructured meshes. In *Parallel Computing in CFD*. AGARD, 1995.

[17] H. D. Simon. Partitioning of unstructured meshes for parallel processing. *Computing Systems Eng.*, 1991.

[18] R. Williams. Adaptive parallel meshes with complex geometry. *Numerical Grid Generation in Computational Fluid Dynamics and Related Fields*, 1991.

[19] R. D. Williams. DIME: A user's manual. Technical Report Report C3P 861, Caltech Concurrent Computation, 1990.