Adaptive methods for PDEs can be considered as a problem in managing a dynamic graph on a parallel processor. The properties we want for this graph are that edges in the graph, as much as possible, map to nearest neighbor links in the parallel processor, and that changes to the graph not require a major re-arrangement of the mapping of the nodes of the graph onto the processor. We will discuss a simple restricted set of transformations which are easy to implement on a message passing parallel processor, and discuss in detail the design choices made. These transformations are based on maintaining a graph of bounded node degree at the cost of a small amount of global communication.

In designing adaptive algorithms for parallel processors, the communication speeds of the processors can have a major effect on the design. Different hypercubes can be characterized by three parameters: the floating point speed, the I/O startup time, and the I/O transfer rate. One aspect of algorithm design which is influenced by interprocessor communication is data structure granularity. We will discuss how this affects the choice of algorithm as a function of the three parameters for our algorithm and relate this to our experiments.

Dynamic Grid Manipulation for PDEs on Hypercube Parallel Processors

William D. Gropp

Research Report YALEU/DCS/RR-458
March 1986

This paper will appear in the proceedings of the ARO Workshop on Parallel Processing and Medium Scale Multiprocessors, Stanford University, January 1986.

This work was supported in part by Office of Naval Research Contract #N00014-82-K-0184 and Air Force Office of Scientific Research Contract AFOSR-84-0360

1. Introduction

Parallel computing offers the possibility of greatly increased computing power. However, some problems are so large that even enormous parallel computers will not be able to handle them. Such problems include time dependent partial differential equations (PDEs) which have both a fine scale and a coarse scale structure. Such problems, solved on a uniform grid, may require a mesh of over 1000 points in each direction, for 10^9 mesh points. Further, each point may require many operations, and thousands of time steps may be needed. A conservative estimate for this kind of problem would be 10^{13} floating point operations. However, the solution being computed at this resolution is in areas of coarse scale structure unusably accurate. We can save a tremendous amount of work by adapting our computation to the structure of the problem. However, adaptive algorithms on parallel processors have received little attention. In this paper, we describe work on a major part of the problem of using adaptive methods on a parallel computer—managing the adaptive data structure. The graph management problem in general is hard and does not parallelize very well, so we consider a restricted set of graph transformations or kinds of refinement/derefinement. We performed some simple experiments to show the "correctness" of the algorithm, the difficulty in implementation, and the bottlenecks in the algorithm. Experiments on an Intel Hypercube are presented.

There are many kinds of parallel processors and ways to program them In this paper we will be concerned with loosely-coupled computers using message passing. A loosely-coupled computer is a collection of processors, each of which is connected to some subset of other processors by communication lines. There is no shared memory. All interaction between the processors takes place though the exchange of messages along these communication lines. Message passing is a programming paradigm suitable to both loosely and tightly coupled computers. A algorithm designed using message passing is data synchronized; each processor is responsible for receiving messages and acting on them. In contrast to the shared memory paradigm, no processor can act directly on the memory or data of another processor. This approach removes many of the possibilities for subtle, difficult to debug errors common in shared memory programs. The penalty is in a slightly higher software overhead and a more active roll of the programmer in keeping track of a distributed data structure.

We choose a hypercube or binary n-cube as our model because it has both good local and reasonable (log in the number of processors) global communication. Further, commercial hypercubes are now available. However, we will see that simpler architectures are also adequate (and may even be superior).

2. Background

In order to discuss the reasons for our choice of data structure and algorithm, we need to model our parallel processor. There are two parts to this model. The first is the performance of a single processor. This can be expressed by three parameter: f, the time to do a single floating point operation, β, the time to transfer a single word from this processor to a neighboring processor, and α, the time to start up a transfer to or from a neighboring processor. In this paper we are concerned only with the management of the adaptive method, and so the floating point time f will not enter our calculations. However, β is roughly the same size as f. For many systems, $\alpha \gg \beta$ because of significant software overhead. Typical numbers are $\beta = 10\mu$ seconds and $\alpha = 6500\mu$ seconds.

The second is the properties of the interconnections between the processors. The hypercube is rich in structure; only a few of its properties interest us here. One property is that a k dimensional grid may be imbedded in a hypercube of dimension $\geq k$ as long as the lengths of the sides of the

grid are powers of 2. The other property we will need is global communication, which is possible in $\log p$ time for a hypercube with p processors.

3. Adaptive methods as graph manipulation

Consider the data as a graph. A vertex of graph represents "point" of computation. The edges of the graph represent the sharing of information between vertices. These edges are determined by the difference stencil for finite difference methods and by the finite element footprint for those methods. For example, the "usual" 5-point difference stencil for an elliptic PDE is a piece out of this graph, and a step of the solution at each vertex consists of using information from the adjacent vertices.

Because a non-adaptive method is often highly ordered, the graph is represented implicitly, for example, through the way in which data is stored. In an adaptive method, the generality of the graph often requires that the graph be represented explicitly. The operations on the graph are refinement and derefinement. Refinement replaces one vertex with 2 or more vertices, with those vertices connected by edges (not necessarily a complete graph). De-refinement replaces 2 of more vertices with a single vertex. Of course, edges must be properly merged if the computational domain which the graph represents is to remain correct.

Thus managing an adaptive grid is the same as managing a dynamic grid.

In practice, it is more efficient to allow each vertex to be a "cubical" subgrid, as this reduces the size of the graph and the amount of "pointer-chasing", as well as providing opportunities for vector processing within a vertex.

Once we view an adaptive method as a graph manipulation, we need to consider how to map it onto a parallel processor. And more than that, we must try to pick a graph which *can* be mapped efficiently onto the parallel processor we are considering. For example, Berger and Bokhari [1] consider one particular method of refinement using "binary decomposition". Other partitionings are possible, and may be more suitable for different adaptive algorithms. The point is that, as with many parallel algorithms, the best choice for a parallel processor is not necessarily the best serial algorithm.

In order to help decide what sort of graph and graph manipulation to use on our parallel processor, we need a criteria to use in selecting among the possibilities. There are several criteria to consider:

- cost of graph, including changes to the graph
- suitability for solving PDEs with the graph—matrix solvers, explicit finite difference schemes
- simplicity of code for simplicity of user interface

The last two items here are very important but hard to quantify. Perhaps most important is that the the graph must be compatible with the solution algorithms. Given the complexity of parallel computing, the structure of the graph must be simple enough to allow easy integration with a wide variety of solution algorithms and with developing linear equation software. In addition, a problem with MIMD parallel computers is that each processor has a copy of the program. This can rapidly chew up a lot of memory if a 1024 processor system is running a program which takes half a megabyte, for a total of half a *gigabyte* for the parallel program. This emphasizes the need for a simple program.

The first point is the easiest to get a handle on. To determine the cost of manipulating the graph, we note that there are 3 sources of cost:

- Cost of communicating information along edges
- Cost of re-arranging the graph. This may include moving vertices to another processor.
- Cost of computing the configuration of the graph

The goal is to minimize the cost

$$\text{cost} = \sum_{\text{steps}} \max_{\text{processor}} (\text{cost of adjacent edges}) + \max_{\text{processor}} (\text{cost of computing graph for each processor}) + \max_{\text{processor}} (\text{cost of re-arranging graph for each processor})$$

Here, "steps" refers to the time or iteration steps taken by the PDE solution algorithm. Since any processor will likely have more than one vertex, we take the maximum over each processor rather than over each vertex.

Fortunately, choosing the graph that minimizes this cost is impossible. If nothing else, the changes in the graph depend on the solution, and can not be predicted. So, we must use heuristics. We will try to make the cost of re-arranging the graph low or zero; to control the cost of edges we will enforce data locality. The cost of computing on the graph will be large, consisting as it does of many floating point operations. However, this term can be made as small as possible by *load balancing*; distributing the vertices among the processors so that each has almost the same amount of floating point work to do.

Also, we want to keep in mind that what we want is the best way to solve PDEs, not a way to sell a particular style of parallel computer. Thus, we would also like to consider the question of what kind of parallel computer is best (shared memory or message passing), and what kind of interconnection (ring, hypercube, etc.). How do the parameters of the machine affect the choice of algorithm? We won't answer these questions here (several years of experience at least will be required), but the work we are doing is aimed at finding answers to these problems.

4. 1-irregular grids

A 1-irregular grid is a special kind of graph which we will use as the basis of our mesh refinement algorithm. We can define it in several ways. One is by the operations which create it:

- Start with a uniform cartesian mesh (of dimension d).
- Replace a node with 2^d nodes. This replacement is itself a uniform cartesian mesh, and the edges are the "natural" edges.
- Allow only 2^{d-1} edges on each *face*. In 3-d, this is in the directions East, North, South, West, Up, and Down.

Such a 2-d graph is shown in Figure 1). Another definition is presented in [2]: In 2 dimensions, given a graph, force additional refinement by applying, as many times as possible, the rule "refine any unrefined element with more than one irregular vertex on one of its sides".

Note that this precludes too rapid refinement refinement

Why do we chose this particular type of grid? One reason is because of the simplicity of the data structure. In particular, the structure can be stored without lists of neighbors because the number of neighbors is limited, and the wasted space is roughly the same as the space used to store the list pointers. For example, in 3-d, the average number of edges is > 6; for a list that would be $6 \times 2 = 12$ words per vertex. If stored as $6 \times 4 = 24$, or 4 words per face, there is only a factor of two space penalty, and no list is necessary. In 2-d, there are other simplifications [2].

Another reason is that this method automatically excludes overly rapid refinement.In addition, for PDEs requiring the solution of linear or nonlinear equations, the assembly of the matrix of a 1-irregular grid is easier than for a general grid [2].

The disadvantages of this structure are that additional refinement, not necessary for the solution but necessary for the graph, may be required. Also, and this will turn out to be a major

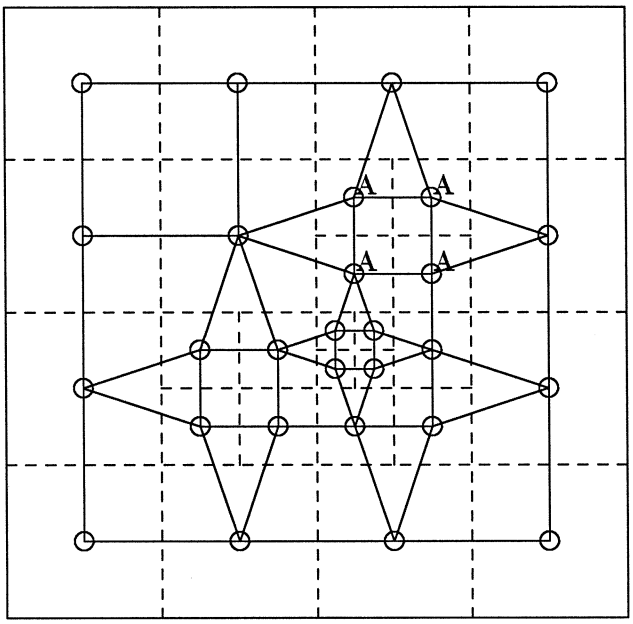

Figure 1: A 1-irregular grid and the associated mesh. The circles represent vertices in the graph; solid lines are edges in the graph. The dashed lines show the associated mesh. The vertices labeled "A" were added to make this graph 1-irregular.

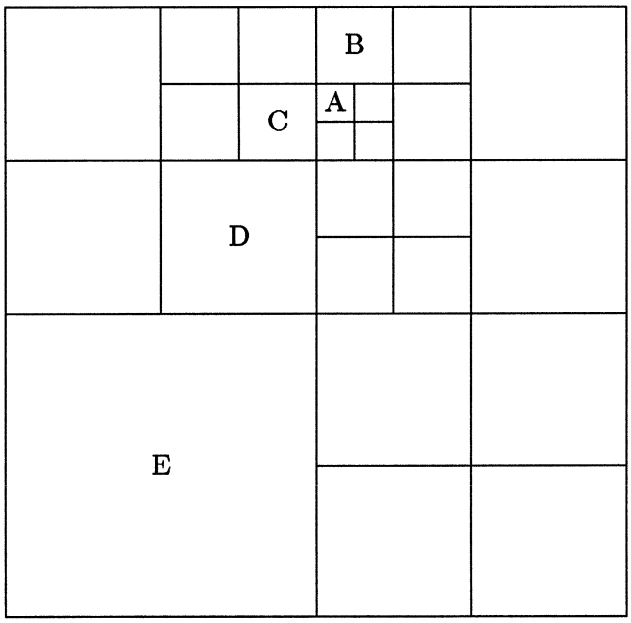

Figure 2: To refine mesh cell "A", cells "B" and "C" must first be refined. To refine cell "C", cell "D" must be refined. Similarly for "E".

problem on parallel computers, a refinement at one point may require massive refinement over a large part of the graph. An example is shown in Figure 2.

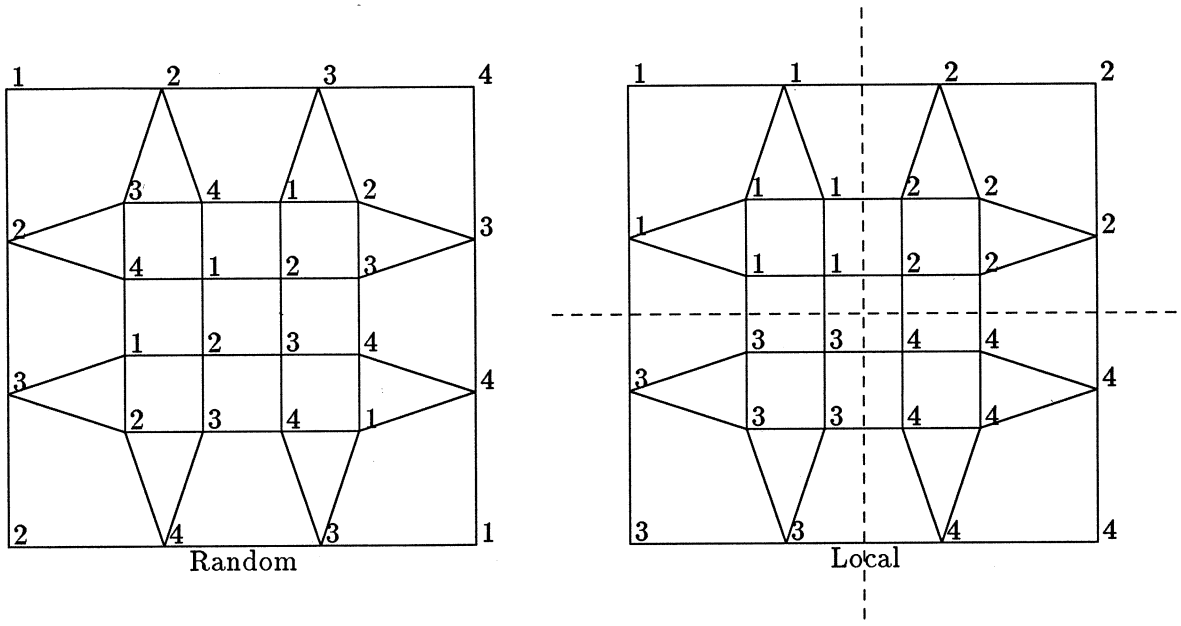

Figure 3: Random and local mapping of vertices onto processors. Mappings for a 4 processor system are shown; dashes in the local mapping show the "clumps".

5. Mapping graph onto processors

There are many ways to map a graph onto a set of processors. We will be interested only in the two most obvious ways, a random mapping and a local mapping. A random mapping assigns vertices in the graph randomly to processors. The idea is to insure that each processor has the same amount of work to do. The disadvantage is that communication between vertices which are neighbors in the graph is now expensive, since those vertices may now be at opposite ends of the parallel processor.

A local mapping reduces the problem of communication by dividing the graph into "clumps", each of which contains most of the neighbors of the vertices in the clump. This approach however suffers from problems in load balancing. However, it can make better use of the local memory, and in particular can make good use of vector or pipelined processors which may be present. These two approaches are shown in Figure 3.

To understand the magnitude of the tradeoffs involved here, we need to more closely analyze the sources of communication cost in a loosely coupled parallel processor.

5.1. Model of communication cost

We assume that each vertex must trade one item of information with each of its neighbors. For simplicity, we will assume that each vertex has six neighbors, and that there are N vertices in the graph. Recall that α represents the IO startup time and β the time to transfer a single word (not byte).

For the non-local or random mapping, each vertex must send data to its neighbors, over an average distance of $d/2$, where d is the diameter of the *parallel processor*.[1] For example, if there are p processors, then a 2-d grid has diameter $d = \sqrt{p}$ and a hypercube diameter $d = \log p$. The total cost per vertex is then

$$6 \text{ neighbors } \times (\alpha + \beta)\frac{d}{2}$$

[1] The average distance is $d/2$ for any parallel processor which has the same number of nodes at a distance i and $d - i$.

Since there are N/p vertices per processor, each processor spends time $3(N/p)(\alpha+\beta)d$ just doing communication.

In the case of a local mapping, only the vertices on the surface of the clump actually communicate to neighbors on other processors. If the clump is roughly cubical, there will be only $6(N/p)^{2/3}$ such vertices. Further, we assume that the clumps are arranged so that the communication distance is one; that is, the clumps fit the architecture of the parallel processor. In this case, the total time per processor is $6(N/p)^{2/3}(\alpha+\beta)$ since the distance is one.

From this, we can see that the communication time for the random mapping is $\frac{1}{2}d(N/p)^{1/3}$ times the time for the local mapping. Since $N/p > 1$ and $d > 2$ for any interesting parallel processor, the random mapping always spends more time communicating than the local mapping. For example, in a 128 node hypercube, with 24 vertices per processor, the random time is ten times the local time. However, in practice, this cost must not be considered without also looking at the effect of load balancing and synchronization.

There are a number of optimizations which we can introduce to reduce the overhead in communication. The most important of these is buffering of the data so that the largest possible messages are sent. Let the buffer size be B, then the times for the non-local and local become

$$\text{non-local} \qquad \frac{N}{pB}6(\alpha+\beta B)d$$

$$\text{local} \qquad 6\left(\frac{N}{p}\right)^{2/3}\frac{1}{B}(\alpha+\beta B)$$

However, this is misleading unless B is small, since in the non-local case the messages are sent to $p-1$ other processors, while in the local case message are send to only ≈ 6 processors. This can severly limit the amount of the buffer which can actually be used. In the local case, we can make $B=(N/p)^{2/3}$; the cost is then

$$\text{local} \qquad 6(\alpha+\beta(N/p)^{2/3}).$$

In the non-local case, the amount of data for each processor is $6N/p$ which is evenly distributed among $p-1$ processors, or $B < 6N/p^2$ so

$$\text{non-local} \qquad p(\alpha+\beta 6N/p^2)d$$

The non-local has $dp/6$ more IO starts (α term) and $(N/p)^{1/3}d$ times as much data to send.

6. Parallel Algorithm

In this section we discuss a parallel algorithm for a local division of the graph among the processors. This algorithm is data driven in that the actions and timing of each processor are determined by the arrival of information for neighboring processors. This ensures that the algorithm is always properly synchronized, and is a major advantage of the message passing paradigm.

There are a number of optimizations involved here. The first is the use of local, read-only copies of vertices on neighboring processors. If a vertex A in processor 3 has a neighboring vertex (determined by the graph) B in processor 7, a read-only copy of vertex B may be kept in processor 3. This copy can be used for references to pointers and data instead of forcing the processor A to send an expensive message to the neighbor processor B. Second, the messages are buffered in the method described above. Third, communication is *predictive*. That is, instead of a processor having to request data, each processor sends the data that will be needed. This can be done because the structure of the algorithm is so simple, and because the local nature of the mapping of the

- determine where to refine
- while cells remain to be refined *on any processor*
  - for each cell to be refined, either refine it or add the neighbors which must be refined first to the list. If one of the neighbors is a remote cell, send that neighbor a message.
  - send EOD (end-of-data) to all neighbors
  - read data from all neighbors. This includes requests to refine, refinement along borders
- determine where to derefine
- while cells remain and the graph changed last time in this loop
  - for each cell to be derefined, try to derefine it.
  - send EOD to all neighbors
  - read data from all neighbors. This consists of updating vertices along borders

Algorithm 1: Algorithm for managing a distributed, 1-irregular graph

graph across processors makes it easy to determine who needs what information. This optimization reduces the communication by a factor of two by eliminating the request part of a request-reply communication protocol.

There is another consideration in designing the program which has not received enough attention. That is the limited amount of memory available for the program. Despite the fact that a parallel computer may have vast amounts of memory (e.g., 64 MBytes), this is spread over many processors. In addition, this memory must be shared with p copies of the operating system, further reducing the available memory. In the case of the Intel Hypercube, only about 0.25 Mbytes are available for user program and data on each node.

Another concern is for technology transfer. If the program is too complicated, it will not be possible to adapt it easily to "real-world" problems, and hence will have little impact on problem solving.

To keep the program simple, all operations are designed to be as self-contained as possible, and to always leave the data structures in a consistent state. The operations are: `refine-cell`, `deRefine-cell`, `Update-value`, `request-refinement`, `get-value`, and `set/read-global-flag`. All of these operations can be implemented in a simple and functional style, making the code robust and small.

Note that the algorithm terminates because at least one cell in some processor is refined each iteration, and in the worst case we will stop when we have a uniform grid.

7. Experiments

The purpose of the experiments described in this section is to show that the algorithm works and to gain an understanding of the performance in the absence of any load balancing. We also investigated the effect of buffering of data on the communication times.

The two test problems used consisted of (1) refineing everywhere, then de-refining everywhere and (2) refineing in a moving sphere. In case (2), the sphere moves in a circle whose origin is at one corner of the domain and whose radius is half of the size of the domain. The radius of the sphere is half the radius of the circle the sphere moves in. Case (1) is a very basic test and, in the absence of load balancing, is the case to which the theory developed in Section 5.1 is applicable. Case (2) is more realistic and give an idea of how important load balancing is.

These tests are quite limited and leave many questions unanswered. Among the items not tested are multiple levels of refinement. In addition, the code has not been optimized and no floating point operations are included. Only the effort needed to manipulate the graph has been included. The effect of this is to accentuate the communication cost of the graph manipulation.

In addition, we tested various decompositions of the domain. We can divide the domain into slabs (a 1 dimensional partition), into bars (a 2 dimensional partition), or cubes (a 3 dimensional partition). The major effect of these different partitions is to reduce the coefficient of α at the expense of increasing the coefficient of β. Since software overheads make $\alpha \gg \beta$, this can be a useful tradeoff. In particular, the basic estimates for the time spent communicating in these three cases are (in the local, buffered case)

$$\begin{aligned} &\text{1-d:} && 2(\alpha + \beta N^{2/3}) \\ &\text{2-d:} && 4(\alpha + \beta N^{2/3}/p^{1/2}) \\ &\text{3-d:} && 6(\alpha + \beta (N/p)^{2/3}) \end{aligned}$$

From these formulas we can construct a graph which predicts where each kind of decomposition will be better. Let $\alpha' = \alpha/(\beta N^{2/3})$. We solve the equation

$$6(\alpha' + p^{-2/3}) = 4(\alpha' + p^{-1/2})$$

for α' as a function of p. This curve separates the region where the cube decomposition is more expensive than the bar decomposition. The solution of the equation

$$4(\alpha' + p^{-1/2}) = 2(\alpha' + 1)$$

gives the curve separating the region where the bar decomposition is more expensive than the slab decomposition. The solutions of these equations are shown graphically in Figure 4. Note that only for very large problems or very small ratios α/β is the cube decomposition superior.

Each experiment was run on each of these divisions of the domain. The results confirm the predictions in the graph.

In tables 1–3 we present the main results or our experiments. This is additional data which does not change this picture. Also, the problems considered are quite small, with the largest containing only 32768 vertices over the entire parallel processor (64 nodes).

As can be seen from Table 3, buffering of the data is not a significant effect, and is probably not worth implementing on this particular hypercube.

From Table 1, we see that the algorithm achieves about 30% efficiency under near optimal conditions of uniform refinement. A close look at the timing data returned by the program showed that much of the "lost" time was in IO waits, i.e., synchronization.

From Table 2, we see that for the small problems considered, the 1-d (slab) decomposition is the best.

Table 1 shows, not unexpectedly, that without load balancing, the performance of the algorithm is poor. The main purpose of this test was to show that it worked, and some speedup was achieved.

The missing values in Table 2 could not be run on the Intel Hypercube because of problems with the system communication software.

8. Some comments on load balancing

In order to load balance, we must move vertices between processors, since in an actual computation, the floating point work in solving a PDE on the mesh will dominate the cost of the

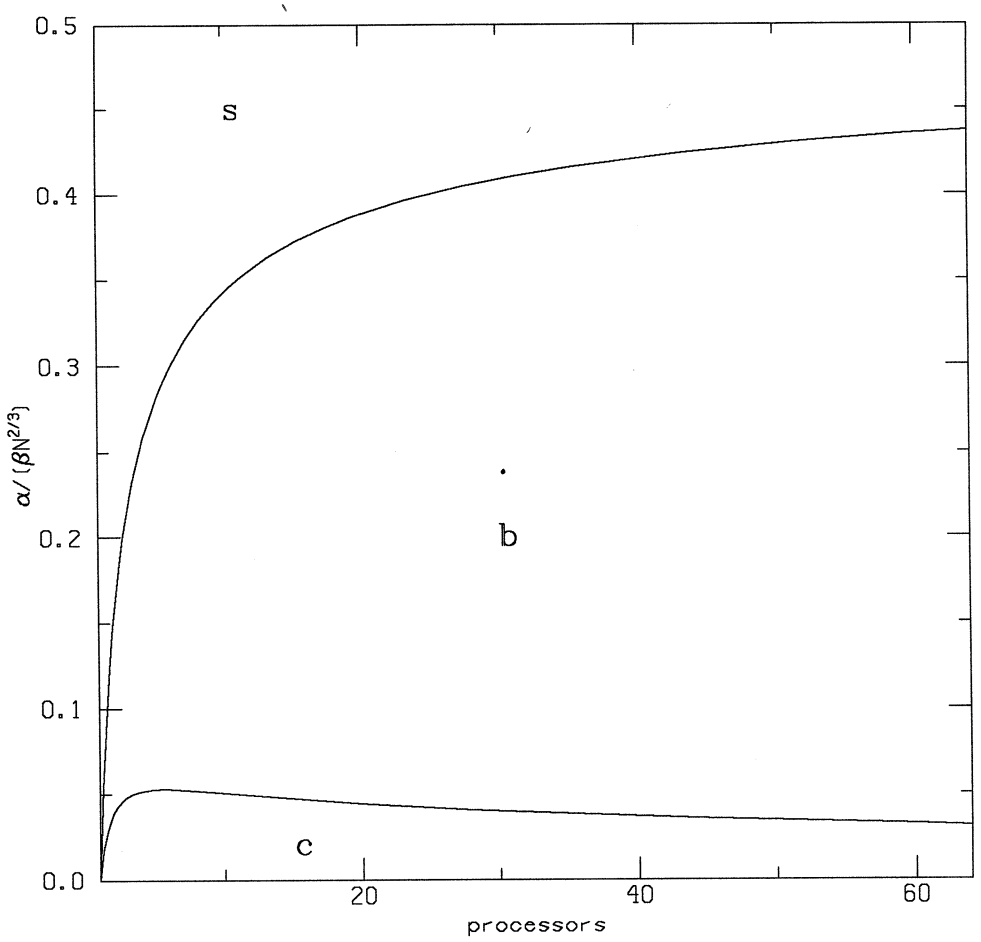

Figure 4: Graph showing where each kind of decomposition of the domain is superior. The region labeled **s** is best for slabs, the region labeled **b** is best for bars and the region labeled **c** is best for cubes.

Nodes	Total Time (ms)	
	All	Sphere
1	12300	1904
64	670	1392

Table 1: Speedup for refinement everywhere (All) and for refinement in sphere (Sphere) for 64 initial vertices.

calculation. However, using an arbitrary scheme for distributing vertices to other processors has all of the problems of the non-local mesh point distribution. Thus we would like to maintain a local distribution of vertices even in the presence of load balancing. There are several possible solutions to load balancing which maintain the local nature of the mapping of vertices onto processors. One method is to nest grids. View each processor as holding a clump of vertices, arranged as an onion. If a processor has too many vertices, take a small clump from the center of the vertices in that processor, and move those vertices to an adjacent processor. This method can be continued by

type	All		Sphere	
	64	512	64	512
slabs	—	12800	—	928
bars	1120	16000	1392	1744
cubes	1280	15300	1392	1872

Table 2: Time as a function of decomposition of domain. 2nd column is for 64 vertices/node, 3rd column is for 512 vertices/node. Times in milliseconds.

type	All		Sphere	
	buffered	unbuffered	buffered	unbuffered
bars	1120	1360	1392	1408
cubes	1280	1470	1392	1392

Table 3: Effect of buffering on the total time. There were 64 vertices/node. Times in milliseconds.

taking an inner clump from the moved clump and giving that to an adjacent processor. Eventually, the original clump will have shells (onion layers) distributed throughout the processor. This method requires very little interconnectivity; even a ring (linear array) will do. A disadvantage is that though load balancing may be achieved by this method, it makes it very difficult to continue to do load balancing as the computation proceeds, as each processor ends up with a number of shells rather than clumps. Further, these shells have a poor surface area to volume ratio, so that communication costs increase as load balancing progresses.

Another approach is to find a restricted set of vertex movements which maintain the local character of the graph to processor mapping (no links longer than 1). The problem here is that there are some configurations which can't be load balanced, or which can be load balanced quickly.

9. Conclusions

Global communication is a serious problem. Most of the synchronization delay and a large fraction of the communication time in the experiments was due to the global flag check step. There are a number of ways to deal with this problem. One method is to observe that the volume of global data is very low. A hardware solution might use a global bus of bandwidth adequate for the amount of data, and perhaps a 2 or 3-d mesh instead of a full hypercube (since we don't need the $\log p$ diameter of the hypercube if we have the global bus). This could be a big advantage for PDE solution techniques which also require a small amount of global communication, such as Conjugate Gradient methods for solving linear equations.

Alternately, we could eliminate or reduce global communication be either allowing a more general graph since it is the 1-irregular property which introduces the global changes, or by finding conditions is which no global communication will be needed. This later should be the most common case, as in practice the graph will be changing slowly.

Another conclusion has to do with the organization of the software on a parallel computer. The basic communication operation in the algorithm consists of sending or receiving the next message along a particular interprocessor connection. There is *no* communication with non-neighbors in the algorithm. Even the global communication is handled most efficiently by nearest neighbor exchanges. However, the software on the machine we used provides a more general abstract model which presents the user with a completely connected model. This model not only slows down the communication, it actually gets in the way of the algorithm. A more effective approach would

be to provide the user with a layered set of software. The lowest level could provide the simple point-to-point communication needed by this algorithm. On top of these routines could be built routines providing a variety of communication models, including the complete connection one Intel provides as the sole model. This is not a new idea; most networking software is designed on these principles. It is time for this same approach to be used in message passing parallel computers.

References

[1] Marsha J. Berger and Shahid Bokhari, *A partitioning strategy for non-uniform problems on multiprocessors,* Technical Report 85–55, ICASE, November 1985.

[2] Alan Weiser, *Local-mesh, local-order, adaptive finite element methods with a posteriori error estimators for elliptic partial differential equations,* Technical Report 213, YALECS, December 1981.