Multiprocessor systems offer large gains in performance *if* algorithms for real problems can be found. We show how one algorithm for solving time dependent partial differential equations, Local Uniform Mesh Refinement, can be implemented on a multiprocessor system. Care is taken to insure that communications costs are kept under control, and an estimate of the performance of this algorithm for a range of configurations is presented. Experiments on a multiprocessor system are compared with the theory.

Local Uniform Mesh Refinement
on Loosely-Coupled Parallel Processors

William D. Gropp

Research Report YALEU/DCS/RR-352
December 1984

This work was supported in part by Office of Naval Research Contract #N00014-82-K-0184, National Science Foundation Grant MCS-8106181, and Air Force Office of Scientific Research Contract AFOSR-84-0360

Multiprocessor systems offer large gains in performance *if* algorithms for real problems can be found. We show how one algorithm for solving time dependent partial differential equations, Local Uniform Mesh Refinement, can be implemented on a multiprocessor system. Care is taken to insure that communications costs are kept under control, and an estimate of the performance of this algorithm for a range of configurations is presented. Experiments on a multiprocessor system are compared with the theory.

Local Uniform Mesh Refinement on Loosely-Coupled Parallel Processors

William D. Gropp

Research Report YALEU/DCS/RR-352

December 1984

This work was supported in part by Office of Naval Research Contract #N00014-82-K-0184, National Science Foundation Grant MCS-8106181, and Air Force Office of Scientific Research Contract AFOSR-84-0360

1. Introduction

Local Uniform Mesh Refinement (LUMR) is a powerful technique for the solution of partial differential equations. It is basically a strategy for the placement of uniform grids on a coarsest mesh which reduces the amount of work by attempting to equidistribute the error committed during the calculation. The use of uniform grids allows the computation on each of the refined grids to be done efficiently, particularly on modern vector and parallel computers. In this paper, we discuss issues in implementing LUMR on a loosely coupled system of parallel processors.

There has been much discussion of ways to implement numerical algorithms on parallel computers, but most of these algorithms fail to seriously consider communications costs. Consider a regular mesh of points, with each point separated by a distance of one unit. Assign to each processor a square with sides of length L. If we assume that advancing the solution at each point requires only information from neighboring points, then the communication is proportional to $4L$ while the computation is proportional to L^2. If $L = 1$ (one processor per point), and if communication is as expensive as calculation, then more time is spent communicating with adjacent nodes than in computation. As L increases, the penalty in communication decreases. (See [3] for more discussion of the effects of communication on algorithms.) We will take communication times into account and show that LUMR addresses this problem by essentially keeping L from getting too small. Specifically, we will answer the questions: Where are the bottlenecks? How should the processors be divided up? What is the parallel efficiency of LUMR? We present experiments on one kind of parallel processor and compare them to our experimental results.

2. Assumptions

Communication between processors is not free. Particularly on high performance systems, we should expect communication to be an important cost in any computation. We will consider a parallel processing system which potentially has a large number of processors, with "large" here being hundreds to thousands. Each processor is capable of solving the PDE on a rectangle and of building the necessary data structures (see Section 3). The processors are connected together either by a common bus or local area network (such as the Apollo ring network), or in a linear array (such as the XTRAP [6]).

In the bus architecture, we will assume that only one processor can access the bus at a time. The bus has a constant bandwidth independent of the number of processors, so as the number of processors accessing the bus goes up, so does the amount of time to complete a transaction over the bus. In the linear array, each processor can talk to each of its neighbors simultaneously. The time

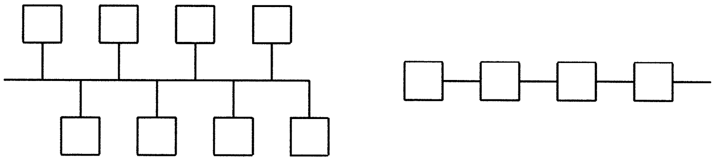

Figure 1: Architectures under consideration.

to complete a transaction over the array is independent of the number of processors (the bandwidth scales with the processors). We will see that there are significant differences between these two kinds of architectures.

3. Serial LUMR Algorithm

We briefly describe the algorithm for LUMR in two space dimensions on a single processor; a more complete discussion of some variants of LUMR may be found in [1, 2]. We will also discuss only a single level of refinement, as multiple levels do not affect the estimates we make here.

The coarse grid is made up of a union of rectangles; each rectangle has a discrete, computationally uniform grid. If two rectangles are connected, they must overlap (this is an implementation requirement) rather than simply abut. This overlap is usually only one space step in width. Every R_g (R_g for ReGridding) steps on the coarse grid, a new refinement also consisting of a union of rectangles is chosen. These new grids are refined by a factor R, that is, the space and time step sizes on the refined grids are reduced by a factor of R from those on the coarse grid. Each grid in the refinement is *overlayed* on top of the coarse grid, rather than inserted into the coarse grid, in order to preserve the uniformity of the coarse grid. While this simplifies the computation on both the fine and coarse grids, it does introduce a number of minor but important complications. First, some points in the domain have both a coarse grid point and a fine grid point. This means that we must inject the values computed on the fine grid into the coarse grid wherever a coarse grid point lies inside the refinement. Second, since the fine grid is not patched into the coarse grid, we must do something special at coarse/fine and fine/fine grid boundaries. At coarse/fine grid boundaries, we can use interpolation from values computed on the coarse grid. At fine/fine grid boundaries, we can take the values needed from the appropriate fine grid (the grids on a level overlap one another to maintain an accurate solution across a fine/fine boundary). Both of these steps are necessary in order to insure an accurate solution. The first is required to prevent the inaccurate solution computed on the coarse grid in the region where refinement is done from affecting the solution

```
$i \leftarrow 0$.
Repeat until done:
  1. Integrate the coarse grid
     Total cost: $C_I n_c$ (data structure cost is negligible)
  2. Regrid if $i$ mod $R_g = 0$
    2.1. Decide where to place the refinements.
    2.2. Initialize the new fine grids with the data in either the old fine grids or in the
         coarse (parent) grid. (Must use the old fine grids if possible.)
         Total cost: negligible
  3. Integrate
    3.1. For $R$ steps: Integrate each fine grid by integrating
      3.1.1. Interior of fine grid
             Total cost: $R^2 n_f C_I$.
      3.1.2. Boundary of fine grid. Fine grids may need to communicate with each other
             (at overlaps between fine grids) and with the coarse grid (where there is a
             coarse-fine grid boundary).
             Total cost: negligible
    3.2. Update the coarse grid with the new solution on the fine grid.
         $i \leftarrow i + 1$
         Total cost: negligible
```

Algorithm 1: Algorithm for serial LUMR.

away from the refinement. The second is necessary to preserve the accuracy on the fine grid (if the values were taken from the coarse grid, the accuracy would be reduced to that of the original coarse grid computation). Thus, the fine and coarse grids must communicate with each other frequently. The algorithm is shown in Algorithm 1.

The potential bottlenecks in a parallel version of Algorithm 1 lie in the steps which need to communicate between processors. These are the regridding step 2, the handling of the boundaries 3.1.2, and the update step 3.2. We can reduce each of these bottlenecks by taking advantage of the local nature of solutions to hyperbolic PDEs and the local (and uniform) nature of the refinements to reduce the amount of interprocessor communication.

Note that the feedback between the grids means that we can't do the fine and coarse grids separately. This suggests that, for explicit methods at least, we partition the algorithm among many processors by keeping the fine grids as close to the coarse (parent) grids as possible, with closeness determined by the communication network.

4. Costs

We will compare the serial and parallel algorithms, and motivate the choices in the parallel algorithm, by estimating the cost of various choices. In order to simplify the discussion, we will make certain assumptions about the costs in the algorithm:

1. The time to do regridding (step 2.1) and the time to do updates (as in step 2.2 and 3.2) are negligible compared to the time to do the integration. This includes the time to set up and move the data structures.
2. The time to transfer data from one processor to another is non-negligible.
3. The size of the data structure is negligible compared to the size to the grids.

We use the following parameters:

C_I Time per point in integration.

C_T Time per point to transfer from one processor to another.

R_g Steps (on the coarse grid) per regridding.

R Amount of refinement.

n_c Number of points in the coarse grid. We assume that the coarse grid is (roughly) square.

n_f Amount of grid which is refined. points. We assume that a fine grid is (roughly) square. $n_f = fn_c$, where f is the fraction of the grid which is refined. The actual number of fine grid points is roughly $R^2 n_f$.

p Number of processors.

We will assume that $C_T = \alpha(p)C_I$, where α is typically of order one. For the ring architecture, $\alpha(p)$ is independent of p; for the bus, $\alpha(p) \propto p$. This is an assumption that communication is (possibly much) slower than a single floating point operation.

R_g is independent of the amount of refinement, and depends only on the problem. It is roughly the minimum size of a refined grid (in problem coordinates) divided by the maximum velocity of solution. This minimum size is basically the amount of *buffer* put around the refined grids to keep the part of the solution which needs to be refined from escaping from the fine grid before the next regridding step. This minimum size is usually picked as a certain number of steps (Δx or Δy) in the fine grid. Since the time step Δt is refined with the space steps, the number of steps that can be taken before a feature could escape from the fine grid it is in is independent of the amount of refinement R. A typical value for R_g is between 4 and 10. The amount of refinement, R, is determined by accuracy and work constraints; typical values are 2 and 4.

The actual number of points in the fine grids is $R^2 n_f$; in other words, n_f is (almost) the number of coarse grid points covered by the refinement. We choose to use this form rather than letting n_f be the number of fine grid points because the ratio $f = n_f/n_c$ is the fraction of the area of the entire domain which needs to be refined. This is a constant (independent of the refinement parameters) determined by the behavior of the problem and the desired accuracy in the solution.

In our estimates, we will assume that the amount of work done is roughly the same on each processor. We will discuss ways of achieving this in Section 8.

5. A Parallel LUMR algorithm

One possible parallel LUMR algorithm is the following: For each rectangle in the coarse grid, divide the grid into p (nearly) disjoint regions. These regions correspond to separate regions in the physical domain; they overlap by only one space step size in the y-direction. (This is the overlap for "connected" grids mentioned in the description of serial LUMR). Assign each region to a separate processor. The exact choice of division will depend on the architecture available. The rule to follow is that adjacent regions should be able to communicate quickly with each other. For the linear array and a 2-D problem, this suggests strips (see Figure 2); for a bus a different partitioning may be better (see Section 7). In a 3-D problem, slabs would be used instead of strips. We will concentrate on the strip partitioning.

Each processor is responsible for its assigned region in the domain. That is, a processor integrates the solution on its domain, and creates and integrates the solution on any refined grids it finds that it needs in its domain. The refined grids are *not* sent to other processors; by keeping them local to the processor which their parent grid lives on, we can significantly reduce data motion. Instead, each processor shares information along the overlap in the domain with it neighboring processor. Thus, the amount of data which must be moved is roughly the square root of the total

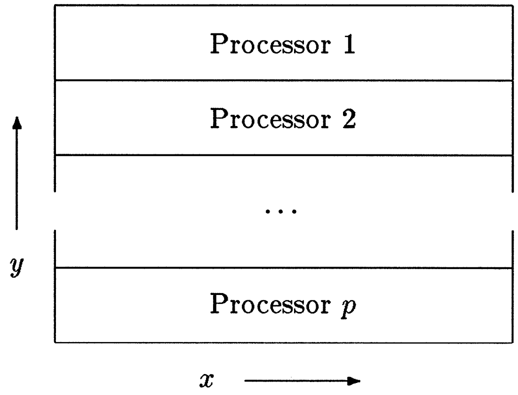

Figure 2: Partitioning of domain for a linear array of p processors. Not shown is an overlap of width Δy for each processor and its neighbor.

data, or equivalently, the data to be moved is "one-dimensional" while the data being computed is "two-dimensional". This partitioning may change during the course of the solution process in order to distribute the load among the p processors, but we will see in Section 6 that it should not change too fast. The algorithm is shown in Algorithm 2.

6. Initializing the fine grids

In this section we motivate the choice of partitioning by considering the other obvious alternative, partitioning by grid. Initializing the fine grids can be expensive if the fine grids are not restricted to lie in disjoint, fixed domains. If the fine grids don't lie in a single domain, then there is the additional cost of $R^2 n_f C_T a$, where a is the fraction of the fine grid which must be initialized with data from another processor. We assume here that any algorithm will attempt to cache data so that a will normally be less than 1.

The ratio of the time to initialize the grid over the time spent integrating the grids is

$$\frac{R^2 n_f C_T a}{\left(\frac{R_g(1+R^3 n_f)C_I}{p}\right)} \approx \frac{pC_T a}{RR_g C_I}.$$

With $C_T/C_I = \alpha(p)$, this is $p\alpha(p)a/RR_g$. α will typically be of order 1 or larger. Typically, p will be larger than RR_g (possibly much larger), so the time spent in communicating the data for the fine grids can be a significant fraction of the total time. As α gets larger, this may become a dominant term. This is likely for bus oriented parallel processors, as $\alpha(p) \propto p$.

This argues for an arrangement where a is small or zero, that is, an algorithm where the new fine grids can be initialized from local or mostly local data. Algorithm 2 uses only local data, so

```
$i \leftarrow 0$.
Repeat until done:
  1. Integrate the coarse grid
     Total cost: $\frac{1}{p}C_I n_c + 2\sqrt{n_c}C_T$
  2. *Regrid* if $i$ mod $R_g = 0$
    2.1. Decide where to place the refinements. Each processor computes the data structure on its region; in doing so, it must receive from its neighbors the data structure for the fine grids at the region boundary.
         Total cost: negligible (data structure cost is $\propto 1/p$, considered negligible)
    2.2. Initialize the new fine grids with the data in either the old fine grids or in the coarse (parent) grid. (Must use the old fine grids if possible.)
         Total cost: $\propto 1/p$, considered negligible
  3. *Integrate*
    3.1. For $R$ steps: Integrate each fine grid by integrating
      3.1.1. Interior of fine grid. Each processor does its own region.
             Total cost: $\frac{1}{p}R^2 n_f C_I$
      3.1.2. Boundary of fine grid. Each processor does its own region. In addition, it must get the fine/fine boundaries from the adjacent processors.
             Total cost: $2R\sqrt{n_f}C_T$. Consider the computational effort at the boundaries negligible by comparison ($\propto 1/p$).
    3.2. Update the coarse grid with the new solution on the fine grid.
         $i \leftarrow i + 1$
         Total cost: $\propto 1/p$, considered negligible
```

Algorithm 2: Algorithm for Parallel LUMR. Costs are per processor.

$a = 0$. Another approach would be to choose a regridding algorithm which has the property that small changes in the solution will cause small changes in the choice of fine grids. In this case, a will be small, because the solution of the PDE is continuous in time. It should be noted that this is very hard to do. None of the approaches that have been suggested have this property.

If the partitioning of the domain among the processors needs to change as a result of the evolution of the solution, then there will be some overhead associated with that. This can be handled by *dynamic load balancing* (see Section 8). The computation effort involved in implementing load balancing is small and can safely be ignored.

7. Estimate of Performance

In this section we estimate the performance of parallel LUMR against serial LUMR. The cost per regridding cycle of serial LUMR is

$$T_s = R_g(1 + R^3 f) n_c C_I$$

while the cost of parallel LUMR is

$$T_p = \frac{1}{p} T_s + 2 C_T R_g \sqrt{n_c}(1 + R^2 \sqrt{f}).$$

The speedup s is given by T_s/T_p, and is

$$\frac{1}{s} = \frac{1}{p} + \frac{2\alpha(p)(1 + R^2\sqrt{f})}{\sqrt{n_c}(1 + R^3 f)},$$

where $\alpha(p) = C_T/C_I$, and $\alpha(p) = \alpha$ for a linear array and $\alpha(p) = \alpha p$ for a bus. We define

$$\beta(p) = \frac{2\alpha(p)(1 + R^2\sqrt{f})}{\sqrt{n_c}(1 + R^3 f)}.$$

This gives

$$\frac{1}{s} = \frac{1}{p} + \beta(p). \tag{7.1}$$

We will use (7.1) in our estimates. See Figure 3 for a graph of β as a function of α and f, using the values in (9.1) for n_c and f.

There are a couple of interesting values for s and p. One is the value of $p = p_b$ where $s = 1$, i.e., the parallel algorithm is just as efficient as the non-parallel algorithm.

$\beta(p) = \beta$ (linear array)	$\beta(p) = \beta p$ (bus)
$\frac{1}{s} = \frac{1}{p} + \beta$	$\frac{1}{s} = \frac{1}{p} + \beta p$
$p_b = \frac{1}{1-\beta}$	$\beta p_b^2 - p_b + 1 = 0$
	$p_b = \frac{1 - \sqrt{1 - 4\beta}}{2\beta}$

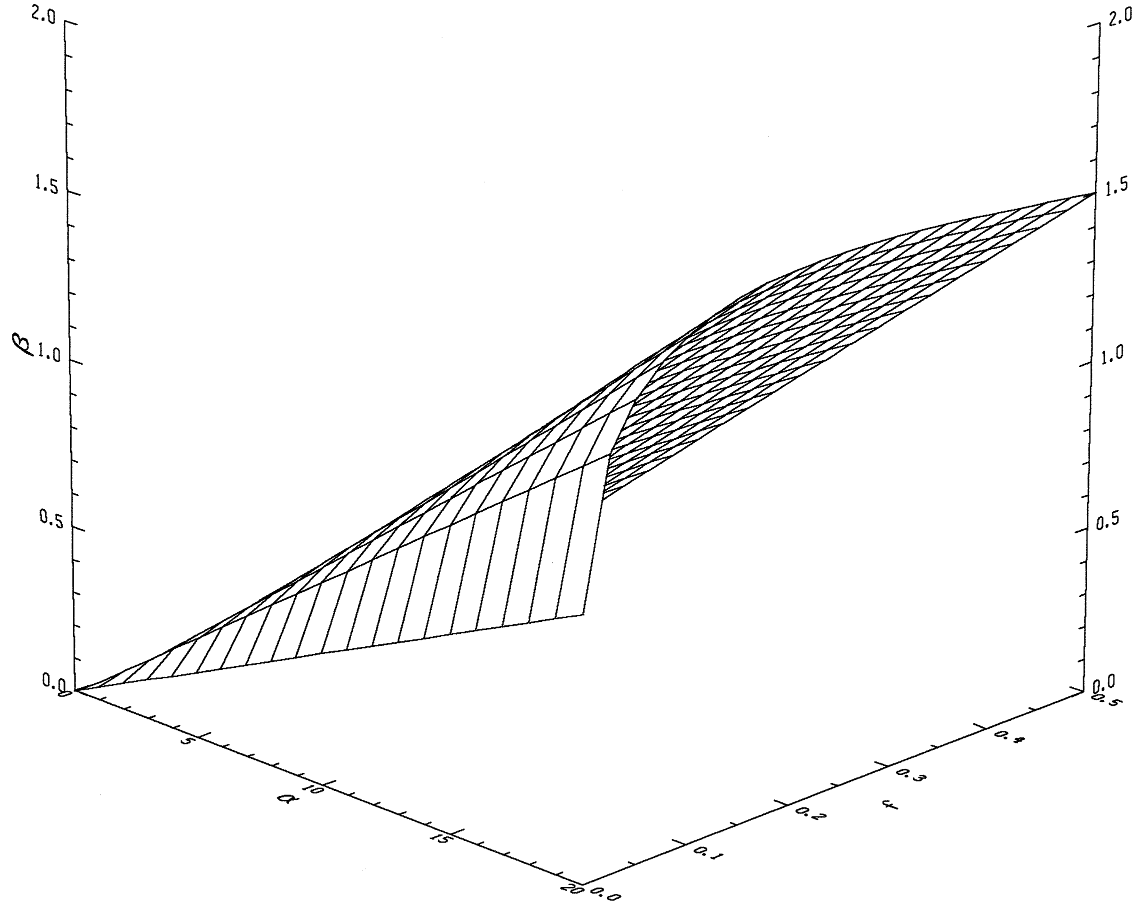

Figure 3: β as a function of α and f.

Note that for the bus, when $\beta > \frac{1}{4}$, the algorithm is *always* less efficient than the non-parallel version. For the linear array, β must be less than 1 for the algorithm to be more efficient than the non-parallel version.

Also interesting is the speedup and the maximum speedup.

$\beta(p) = \beta$ (linear array)	$\beta(p) = \beta p$ (bus)
$\frac{1}{s} = \frac{1}{p} + \beta$	$\frac{1}{s} = \frac{1}{p} + \beta p$
$s = \frac{p}{1+\beta p}$	$s = \frac{p}{1+\beta p^2}$
$\frac{ds}{dp} = \frac{1}{(1+\beta p)^2}$	$\frac{ds}{dp} = \frac{1-\beta p^2}{(1+\beta p^2)^2}$
$\frac{ds}{dp} = 0 \text{ at } p = \infty$	$\frac{ds}{dp} = 0 \text{ at } p = \frac{1}{\sqrt{\beta}}$
$s_{\max} = \frac{1}{\beta}$	$s_{\max} = \frac{1}{2\sqrt{\beta}}$

Note that β must be less than 1 for the parallel algorithm to break even, thus the square root makes the bus architecture far less effective than the linear array architecture.

In these cases, the major bottleneck is in step 3.1.2, where the neighboring fine grid values must be sent to adjacent processors. By chosing a different partitioning, we can reduce this, though at a cost in added complexity for the communication links. For example, if we had a 2-D mesh connected array, we could cut up the domain into squares of side $1/\sqrt{p}$. This would essentially divide the term in step 3.1.2 by $\sqrt{p}$, which will improve these estimates. For example, for a mesh-connected (rather than linear) array, the speedup $s = p/(1 + \beta\sqrt{p})$, which is unbounded. However, the efficiency of processor usage is roughly $1/\beta\sqrt{p}$, and thus goes to 0 as the number of processors goes to infinity. For the bus architecture, the speedup would still be bounded, with $s = p/(1 + \beta\sqrt{p^3})$ and a maximum speed of $s = \frac{1}{3}\sqrt[2/3]{2/\beta}$. This speedup comes from reducing the absolute amount of data to be communicated by reducing the ratio of circumference to area for each of the domains to the minimum possible. Figures 4 and 5 show the speedup s as a function of β and p.

8. Load balancing

One assumption we have made is that the load on each processor is the same as the load on any other processor. In practice, this will rarely happen. Our approach to *dynamic load balancing* is to adjust the partitioning of the physical domain among the processors. We consider the processors to be connected in a linear array from left to right, even if the actual hardware is a bus. "Logically right" means the processor to the right in this ordering. The algorithm used takes the following steps:

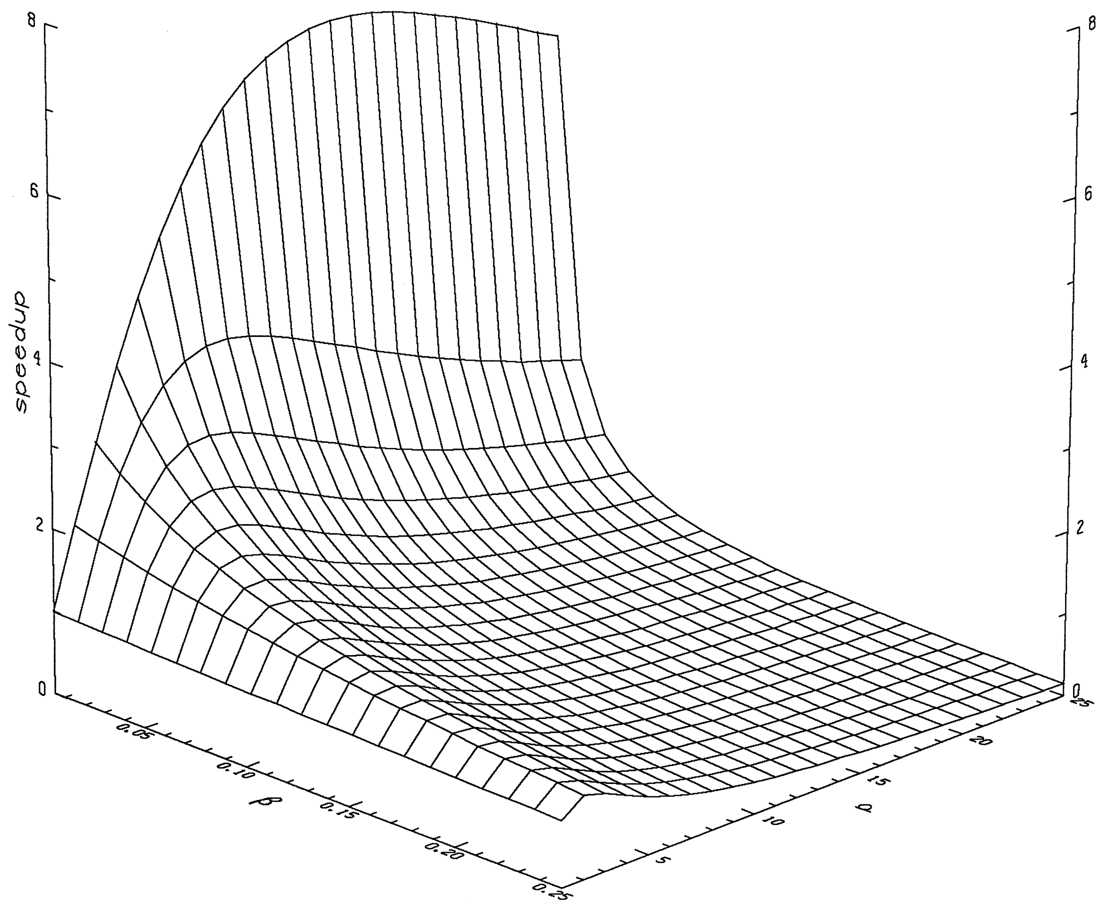

Figure 4: Speedup s as a function of β and p for a bus architecture. The minimum value of β is 0.005

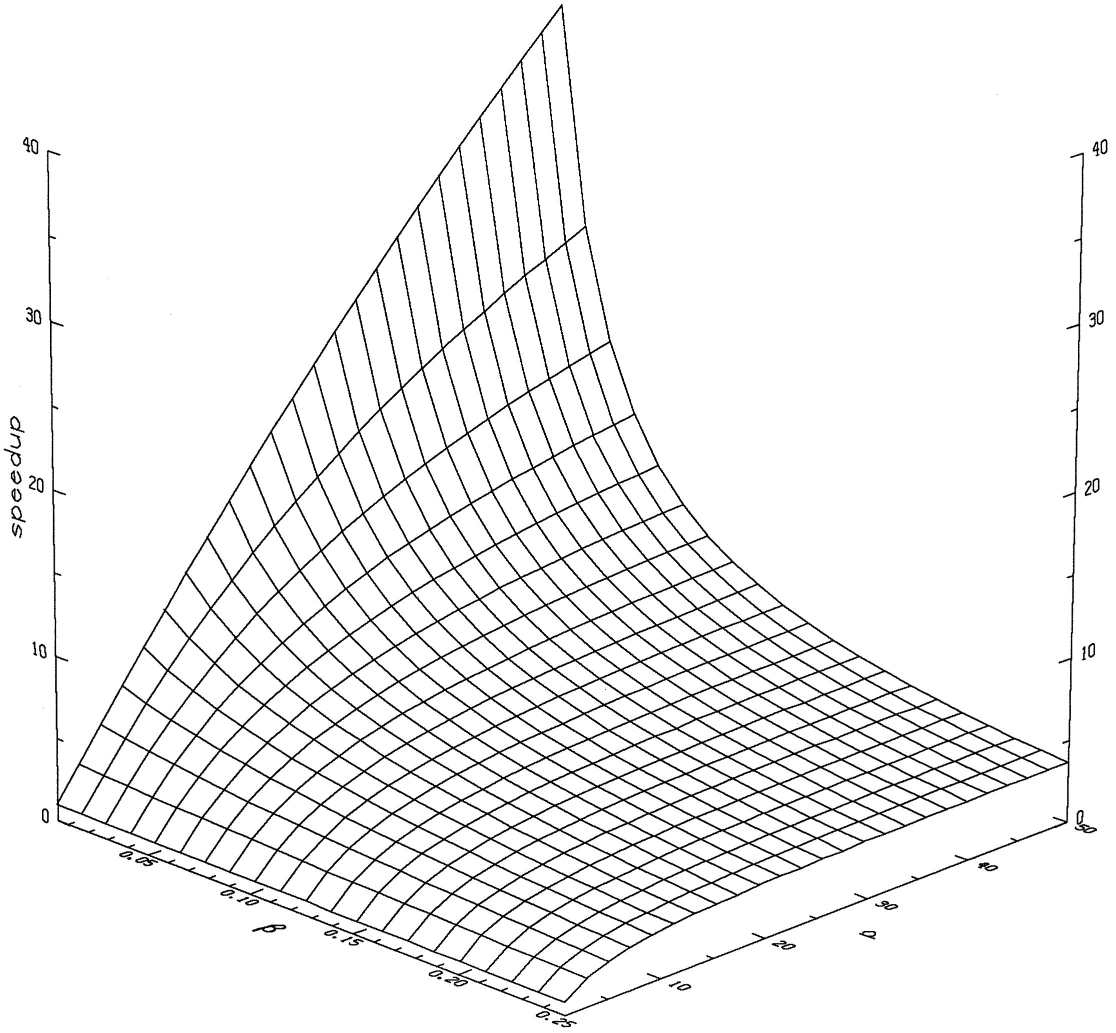

Figure 5: Speedup s as a function of β and p for a linear array architecture. The minimum value of β is 0.005.

w_i The work on the i^{th} processor.

wp_i The partial sum of work on processors $i = 0, \ldots, i-1$.

w_e Average amount of work for each processor.

For each processor $(i = 0, \ldots, p-1)$

1. Estimate the work being done on this processor (w_i).
2. Get the partial sum wp_i from the logically left processor. $wp_0 = 0$ for the leftmost processor. Form $wp_{i+1} = wp_i + w_i$ and pass wp_{i+1} to the logically right processor. If at the rightmost processor $(i = p-1)$, form $w_a = wp_p/p$, where p is the number of processors. Wait for w_a from the logical right and pass it on to the logical left.
3. Compare the estimated work to the logical right (for all the processors) to the desired work to the logical right. The actual work is just $w_r = w_a p - wp_{i+1}$ and the desired work is $w_d = w_a(p - (i+1))$. If $w_r < w_d$, then send $\min(w_i, w_d - w_r)$ to the logical right processor. If $w_r > w_d$, then request $w_r - w_d$ work from the logical right. The actual work received may be less than requested (i.e., the logical right neighbor does not have that much work to give).

Thus this algorithm proceeds in two passes up and down the line of processors. The pass from processor 0 to $p-1$ gathers up the wp_i values. The pass from processor $p-1$ to 0 passes w_a to processor $i-1$, and redistributes the load among processors i and $i+1$, in a way that insures that after the load balancing step, the sum $\sum_{j=i}^{p-1} w_j$ is closer to $\sum_{j=i}^{p-1} w_a$. Since this is true for all i, the work on each processor w_i after the load balancing step will be closer to the desired work w_a. However, since there is a definite direction to the algorithm, and only neighbor transactions are allowed, it is possible that up to $p-1$ steps could be required to distribute the load, assuming that the work on each processor doesn't change between load balancing steps. Such an unbalanced load is very unlikely, however, and in practice only a few steps are needed to balance the load. As the solution develops, the load will change slightly as important solution features move from place to place. Since the solution changes slowly, the partitioning of processors among the domain will change slowly, and this algorithm can easily keep the load well distributed.

In practice, this algorithm worked much better than a purely local algorithm which compared the work on a processor to its left and right neighbors. However, it does have its disadvantages. Beacuse it requires some global data (the work estimate w_a), the processors must wait until every processor is ready. On a linear array this does not impose much of a penalty, as the load balancing algorithm will insure that most of the processors finish at the same time. For the bus architecture, the situation is more complicated. If the loads are *not* well balanced, then some processors will be waiting while others are computing. Those that are waiting can use the bus to send data to other processors. This effectively increases the bandwidth of the bus by overlapping computation (on some processors) with data transfers (on others). Load balancing may actually make things worse by reducing or eliminating the amount of data transfer which can be overlapped with computation on a bus.

9. Experimental results

To test the above theory, we ran a parallel mesh refinement code on a 14 node Apollo ring. The ring consisted of 10 DN300s and 4 DN420s. The Apollo DN300 is a 68010 based workstation with no hardware floating point. The Apollo DN420 is a 68000 based workstation with hardware floating point and a hard disk. The Apollos are connected by a token passing network; at the user level, processes on different processors communicate through mailboxes which live on a disk. The DN300s were used for the actual computations, with the DN420s being used to hold the actual mailboxes, monitor the computation, and act as partner to the DN300s. There is a fixed I/O bandwidth which is independent of the number of processors, so this configuration is similar to a bus architecture and will be compared with our results for a bus.

The sample problem is a variant of the revolving cone problem used in [4] and [1]. The problem is:

$$u_t - yu_x + xu_y = 0,$$

with the initial condition

$$u(x,y,0) = \begin{cases} 1 - 16r, & \text{if } r < \frac{1}{16}, \\ 0, & \text{otherwise,} \end{cases}$$

where

$$r = \left(\left(x - \frac{1}{2} \right)^2 + \frac{3}{2} y^2 \right).$$

The boundary conditions at the inflow boundaries are zero. The solution to this variable coefficient problem is given by rigid counter-clockwise rotation of the initial data about the origin with angular frequency $\omega = 1$. The region of refinement is concentrated around the cone; hence no static partitioning will make effective use of the available processors. In our test, we use Lax-Wendroff as the difference approximation, with inflow boundaries specified as $u = 0$ and outflow boundaries specified with first order extrapolation.

The processors were divided up as equal sized strips in y; for the two processor case, the initial domains were $-1 \leq y \leq h$ and $0 \leq y \leq 1$, where h is the space step size. The values of the various parameters are:

$$\begin{aligned} C_T/C_I &\approx 0.16p \\ R_g &= 10 \\ R &= 2 \\ n_c &= 51 \times 51 \\ n_f &\approx .2n_c \end{aligned} \tag{9.1}$$

The value for $C_T/C_I = \alpha p$ was determined by comparing the time for the unrefined grid calculations given in Table 1 to

$$T_p = \gamma \left(\frac{1}{p} \sqrt{n_c} + 2\alpha p \right) = a_1 \frac{1}{p} + a_2 p.$$

A least squares fit to the data for the unrefined calculations (for $p = 2, 4, 6, 8, 10$) determined $a_1 = 1814$ and $a_2 = 11.5$. As can be seen from Figure 8, the fit is quite good. The value of β for these parameters is $\beta = 0.0067p$.

Three sets of runs were made. The first run used no refinement and is a simple test of the parallel integration algorithm. The second test used fixed partitions. The third test used dynamic load balancing to modify the partitioning in an attempt to more equally distribute the load. The results are presented in Table 1 and graphically in Figure 7.

In Figure 8, we compare our experimental results with our predictions. We see that the estimated elapsed time is greater than we observed, but only by about 20%. The discrepancy is probably due to a combination of lack of load balancing and to items we have neglected, such as the computational time in handling the refinements (regriddings, updatings, data structure transfers). They clearly show that 10 Apollo DN300s are about all that could be used efficiently with this algorithm, because the interprocessor communication mechanism is too slow, and that our theory is an accurate model of the algorithm.

10. Conclusion

The partitioning of the problem by domain rather than by grid was the key in keeping this algorithm efficient. This same approach may also be applied to less regular grids, as long as there is some way to solve the problem on a partial domain such as using an explicit method or an

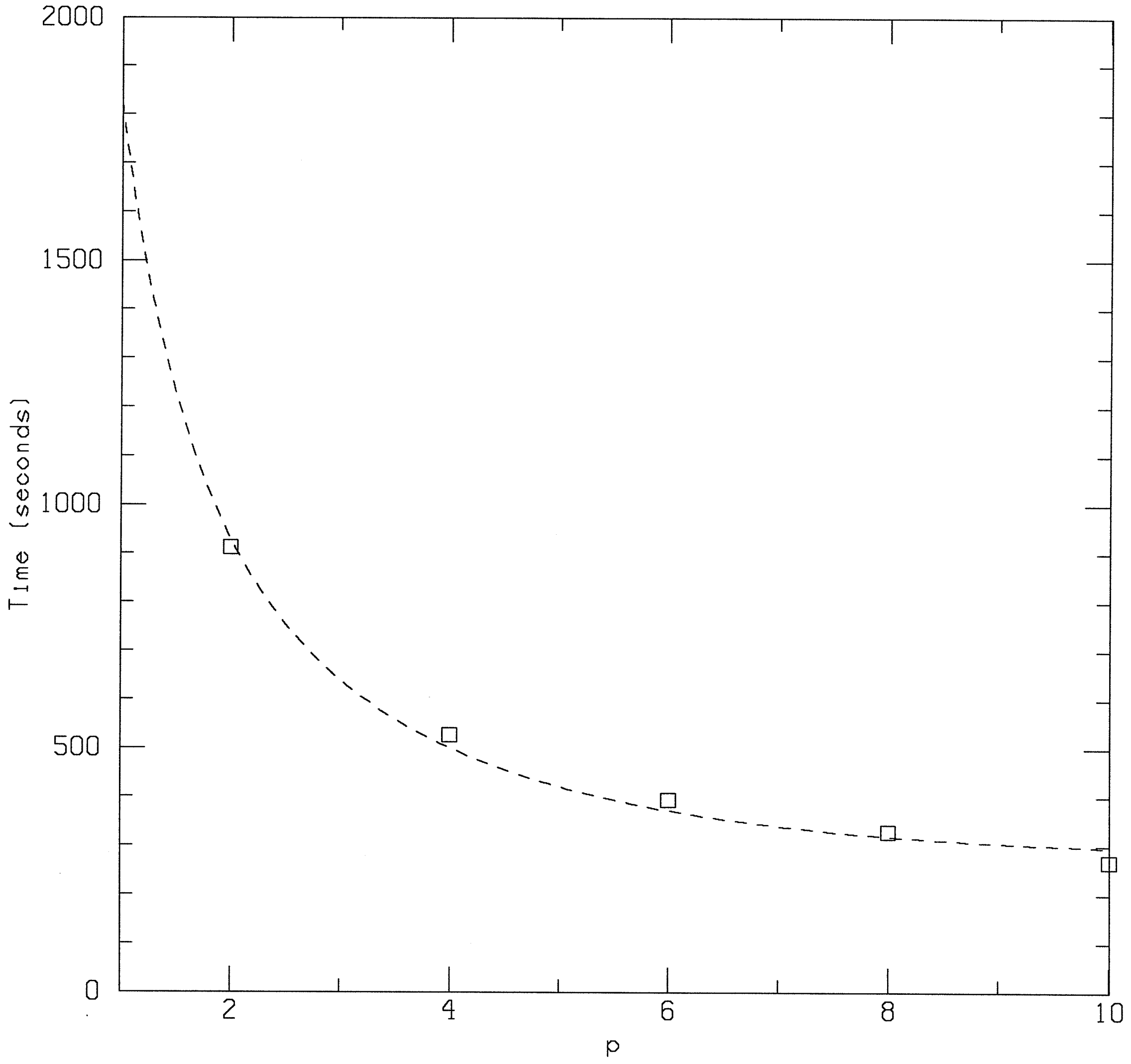

Figure 6: Comparison between the predicted (dashed line) and observed (squares) times for a no-refinement calculation.

Processors	Elapsed Time			Max CPU Time			Average CPU Time		
	a	b	c	a	b	c	a	b	c
1	1714	3482	NA	1699	3450	NA	1699	3450	NA
2	912	3025	2468	887	2940	2199	872	2089	1820
4	527	2149	1625	495	2026	1379	462	1152	1092
6	393	1770	1385	359	1491	1133	325	815	814
8	329	1521	1128	290	1328	858	256	656	668
10	267	1259	1048	222	1083	723	215	555	582

Table 1: Times for parallel computation. a is for no refinement, b is for refinement without load balancing, and c is for refinement with load balancing.

iterative technique such as the Schwartz alternating procedure. Such a technique may work well with implicit integrators and with time independent problems.

In three dimensions, the analysis is much the same, using slabs instead of slices. Because the size of the interface between slabs is proportional to the two-thirds power of the number of points (one-half in the 2-D case), the overhead will be more significant. In this case then it is probably better to divide the domain into cubes rather than slabs to reduce the overhead.

It is clear from our results that bus oriented architectures will have trouble once the number of processors gets very large. The algorithm we have described was designed to significantly reduce the amount of data being moved between processors, yet even at 10 processors, the we are almost at the minimum of the elapsed time curve (12 is the predicted minimum). Note that these processors are very slow; β would be larger for a system using state-of-the-art processors and communication. On the other hand, our analysis shows that a linear or mesh-connected array of processors offers far more potential speed up, even for much larger values of β. Such systems are starting to appear on the market and should allow effective parallel programs to be constructed.

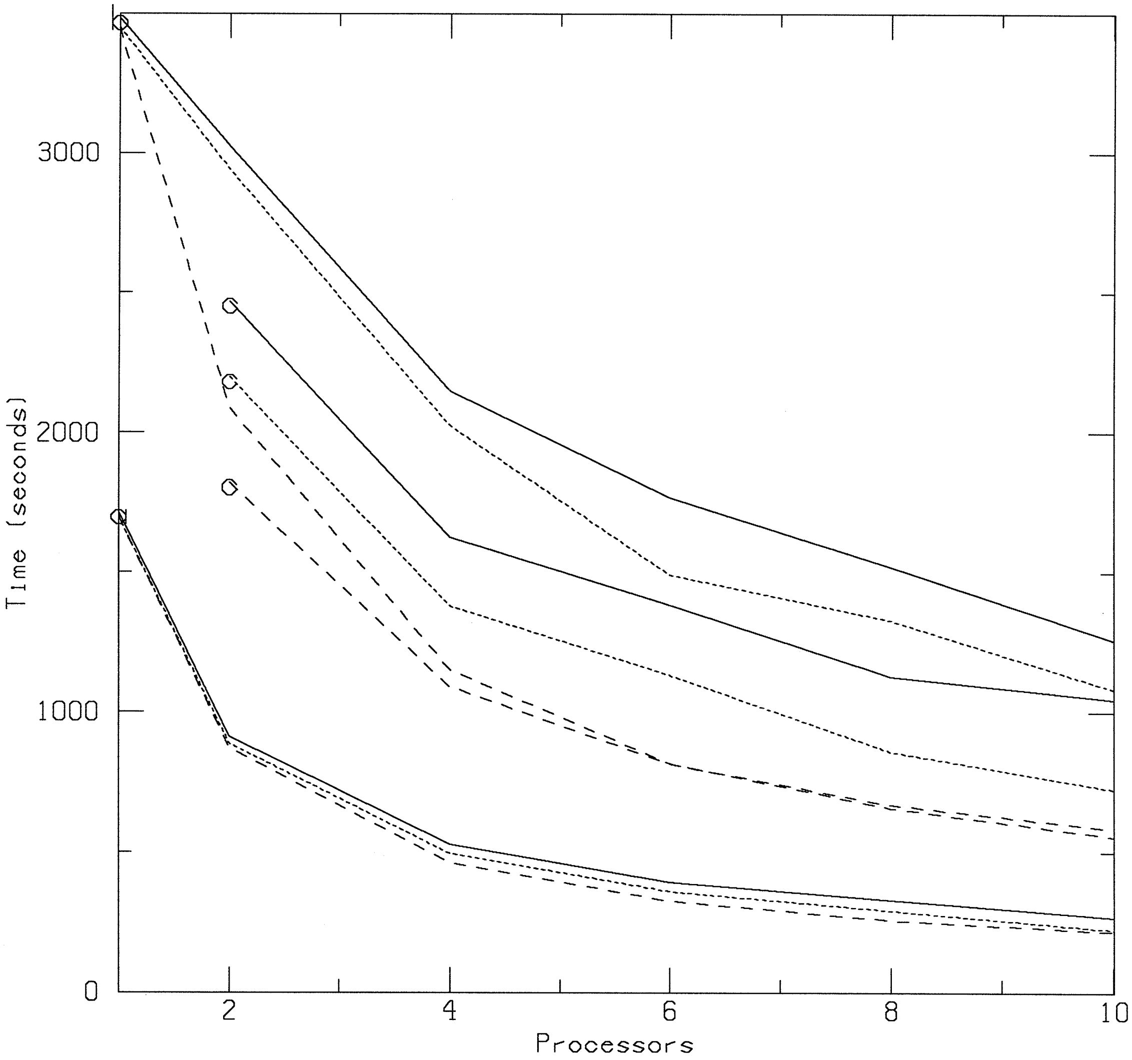

Figure 7: Timing results for experiments. The data is from Table 1; a, b, and c have the same meanings. The solid lines are the total elapsed times, the dotted lines are the maximum cpu times and the dashed lines are the average cpu times.

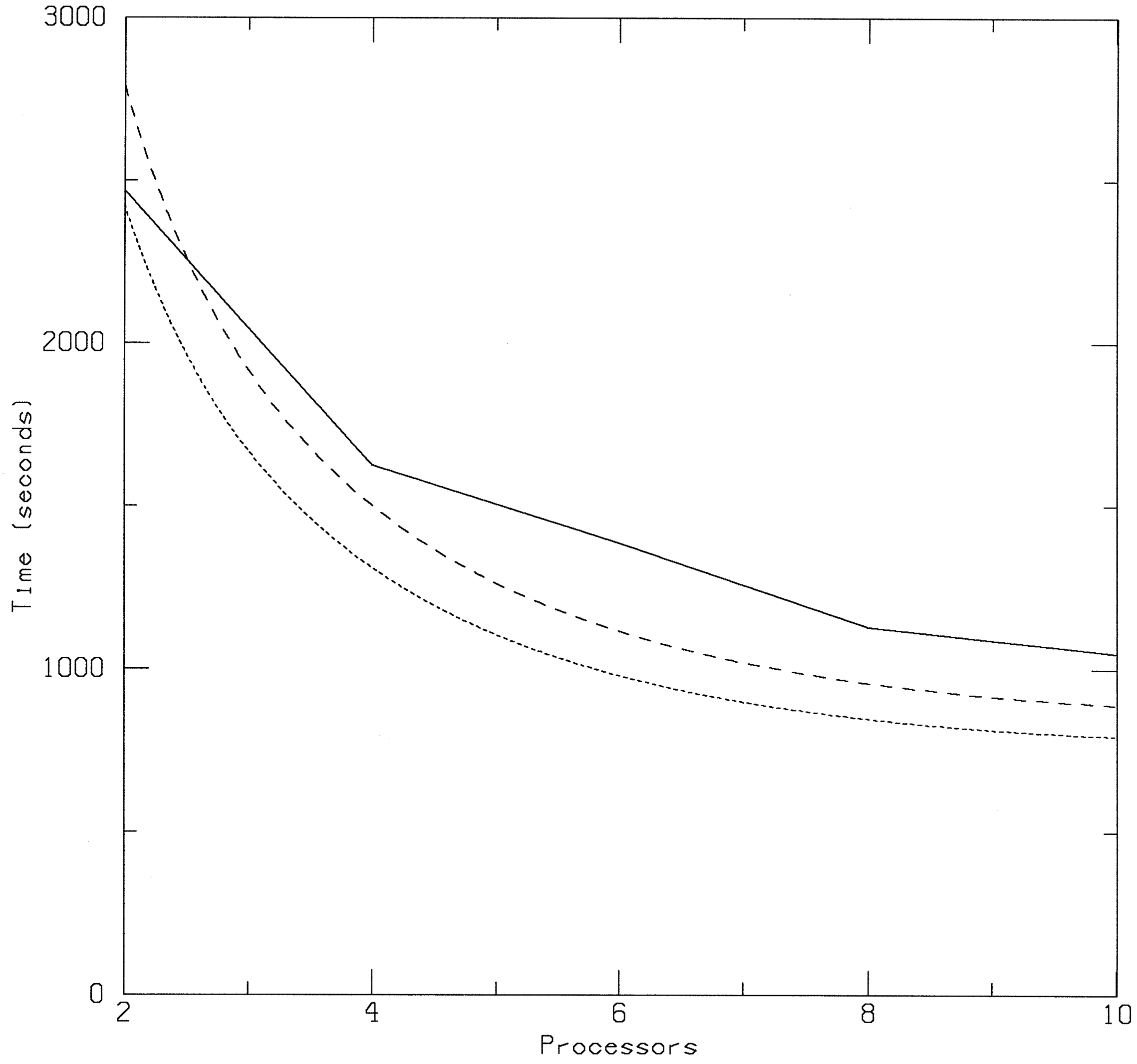

Figure 8: Comparison of predicted elapsed times with the observed times with load balancing. The solid line is the observed time. The dashed line is the predicted time with $f = 0.25$, and the dotted line is the predicted time with $f = 0.20$.

References

[1] Marsha J. Berger and Joseph Oliger, *Adaptive mesh refinement for hyperbolic partial differential equations,* Technical Report ManuscriptNA-83-02, Stanford University, March, 1983.

[2] John H. Bolstad, *An adaptive finite difference method for hyperbolic systems in one space dimension,* Technical Report LBL–13287–rev, Lawrence Berkeley Laboratory, December 1982.

[3] Dennis Gannon and John Van Rosendale, *On the impact of communication complexity in the design of parallel numerical algorithms,* Technical Report 84–41, ICASE, August 1984.

[4] David Gottlieb and Steven A. Orszag, *Numerical Analysis of Spectral Methods: Theory and Applications,* Society for Industrial and Applied Mathematics, 1977.

[5] William D. Gropp, *Local uniform mesh refinement for elliptic partial differential equations,* Technical Report YALE/DCS/RR-278, Yale University, Department of Computer Science, July 1983.

[6] Martin Schultz, *XTRAP,* 1984. In preparation.