Department of Computer Science Technical Reports

Department of Computer Science

1988

Analysis of a Two Level Asynchronous Algorithm for PDEs

John R. Rice
Purdue University, jrr@cs.purdue.edu

Dan C. Marineseu

Report Number:
88-800

ANALYSIS OF A TWO LEVEL ASYNCHRONOUS ALGORITHM FOR PDES

John R. Rice
Dan C. Marinescu

CSD-TR-800
August 1988

ANALYSIS OF A TWO LEVEL ASYNCHRONOUS ALGORITHM FOR PDES*

John R. Rice*
Dan C. Marinescu*

Computer Sciences Department
Purdue University

CSD-TR-800
August 1988

Abstract

A two level asynchronous algorithm for PDEs is presented in this paper. An analysis of the algorithm and of its implementation on a hypercube shows that the algorithm has several desirable properties; its speed up increases as the number of grid points grows, it is stable with respect to load imbalance effects, and it is well suited for parallel machines with relatively slow communication.

* Work supported in part by the Strategic Defense Initiative under Army Research Office contract DAAL03-86-K-0106.

ANALYSIS OF A TWO LEVEL ASYNCHRONOUS ALGORITHM FOR PDES

*John R. Rice**
*Dan C. Marinescu**

Computer Science Department
Purdue University

CSD-TR-800
August, 1988

Abstract

A two level asynchronous algorithm for PDEs is presented in this paper. An analysis of the algorithm and of its implementation on a hypercube shows that the algorithm has several desirable properties; its speed up increases as the number of grid points grows, it is stable with respect to load imbalance effects, and it is well suited for parallel machines with relatively slow communication.

1. Introduction

As the number of multiprocessor systems increases, practical questions related to their efficient use as well as questions of how to design new systems capable of high performance stimulate performance related studies in parallel processing.

The inefficiency associated with parallel execution usually is attributed to two causes: communication among the tasks executed concurrently, and control of parallel activities including scheduling of tasks. Another potential source of inefficiency which has received less attention is the *synchronization among concurrent tasks*. Intuitively we expect that synchronization leads to an increase in the execution time, hence any model ignoring synchronization is an optimistic one.

Modeling and analysis of synchronization in parallel computing raises difficult questions. Empirical data are largely unavailable, due to the present state of the art in the instrumentation of parallel systems. Only measurements related to the aggregate program behavior such as total execution time, processor utilization, etc., can be carried out routinely, while detailed data concerning communication and synchronization costs are usually unavailable. There are cases in parallel processing when synchronization

* Work supported in part by the Strategic Defense Initiative under Army Research Office contract DAAL03-86-K-0106.

cannot be avoided, for example in case of domain decomposition techniques for partial differential equations (PDEs) [3], [4].

We have developed a non-deterministic model for parallel computation [1] which shows that in the general case the overhead associated with synchronization depends principally upon two factors, namely, the number of PEs running in parallel, and the actual distribution of the execution time on PEs. For particular distributions the overhead associated with synchronization is independent of the number of PEs running in parallel when this number is large, hence massive parallelism does not become prohibitively expensive solely due to synchronization. This is the case for the uniform distributions, when only the coefficient of variation of the distribution determines the synchronization overhead. For other distributions, like the exponential one, the synchronization overhead grows logarithmically in the number of PEs.

In this paper we present a two level asynchronous algorithm for PDEs. Section 2 gives the description of the algorithm and its rationale. Then we outline the basic concept of our unified model in Section 3, and use the framework provided by it to construct the detailed model of the algorithm in Section 4. Section 5 contains a summary of the results of our analysis of the tow level algorithm.

2. A Two Level Asynchronous PDE Algorithms

Consider the PDE problem $Lu = f$ on the domain $D = [0,1] \times [0,1]$ with Dirichlet boundary conditions We first subdivide D to obtain *Level 1* with $N1$ overlapping domains

$$D_k = [0,1] \times [(k-1)/N1,\ k/N1]\,, \quad k = 1,2,\ldots,N1 \quad .$$

We then subdivide each D_k into $N2/N1$ overlapping domains

$$D_{kj} = [0,1]\times[(k-1)/N1 + (j-1)\delta\,,\ (k-1)/N1 + j\delta],\ \ j = 1,2,\ldots,N2/N1 \quad .$$

where $\delta = 1/N2$. This determines *Level 2* with $N2$ domains. The geometry of the algorithm is illustrated in Figure 1 for $N1 = 4$, $N2 = 12$.

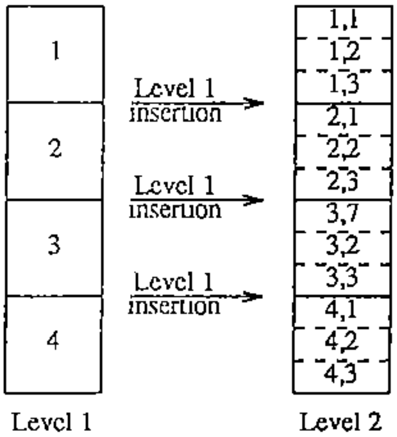
We discretize the linear PDE on each level, using n points on Level 1 and qn points on Level 2. An iteration method is then used to solve the resulting linear system on each level. We anticipate a parallel implementation of the iteration with $N1$ processors assigned to Level 1 and $N2$ processors to Level 2. The PDE algorithm is described at a very high level.

The rationale for this algorithm is illustrated in Figure 2. On Level 1 we have a coarse grid, so the convergence is rapid but the PDE discretization error is large. On Level 2 we have a fine grid, so the convergence is slow but the PDE discretization error is small. When the error in solving the linear system on Level 1 reaches point A, no further gain is made in solving the PDE. At this point, Level 1 is stopped. An idealized view of how the algorithm works is that: (a) Level 1 operates quickly to reach point A, (b) the accurate solution from Level 1 is inserted into Level 2 as boundary conditions on Level 2 subdomains (the size of the Level 1 domains), (c) when Level 1 no longer provides additional accuracy the iteration is continued on Level 2 in the normal manner. This idealized operation is indicated by the path in Figure 2 through points C and D. The path from start to solution using the same methods entirely on the fine grid is indicated by point B.

Reflection shows that the algorithm is unlikely to be as efficient as the ideal because the accurate values received from Level 1 do no propagate instantaneously from the Level 1 boundaries (where information is transferred) to the entire Level 2 grid. This propagation is made by the iteration on Level 2 which apply to the $N1$ subdomains. Since these subdomains are smaller, the iteration converges faster on them (even with the fine grid) than on the whole domain. Thus the actual computation error follows the path through the points C^* and D^*. The issues addressed here are: 1) Is a significant advantage obtained (i.e., where are C^* and D^*?) and 2) How well is this algorithm suited to parallel, asynchronous execution on a hypercube machine?

We have analyzed this algorithm with models that assume rates of convergence for the linear system which range from those of Gauss-Seidel (slow) to SOR with optimum parameter (fast). We have also analyzed the effect of different discretizations, namely, second order and fourth order. These are, of course, very large differences in the performance among these possibilities, but the performances relative to the standard fine grid algorithm are very similar, so we only discuss one of them, the second order discretization and Gauss-Seidel type iteration. See Section 4 for more details of the performance model and analysis.

We basically assume that the PDE error behaves like $1/n^2$ and $1/(nq)^2$ on Levels 1 and 2, and that the rate of convergence of the iteration for the linear system is like $(1 - 1/k^2)$ where k^2 is the number of grid points in the domain.



$D_k, k = 1,2,3,4 \quad D_{kj}; k = 1,2,3,4; j = 1,2,3$

Figure 1. *The Level 1 and Level 2 partitions of D into N1 and N2 domains. The algorithm has information of Level 1 inserted asynchronously into Level 2 from the boundaries of the domains of Level 1.*

Level 1 Algorithm

```
1.  Initialize
2.  Iterate
        For $k = 1$ to $N1$
            Iterate on linear system in $D_k$
            Exchange values with $D_{k+1}, D_{k-1}$
            Insert values to Level 2
            Compute stopping test
        End For
        If all stops true
            Send stopped to Level 2
            Stop
        End If
    End Iteration
```

Level 2 Algorithm

```
1.  Wait for first Level 1 values
2.  Initialize
3.  Iterate
        Case (Level 1 not stopped)
            For $k = 1$ to $N1$, $j = 1$ to $N2/N1$
                Iterate on linear system in $D_{k,j}$
                Case ($j = 1$ and $N2/N1$)
                    Use values from Level 1
                    Use remaining values from neighbors
                Case ($j \neq 1, N2/N1$)
                    Exchange values with neighbors
            End For
        Case (Level 1 stopped)
            For $k = 1$ to $N1$, $j = 1$ to $N2/N1$
                Iterate on linear system in $D_{k,j}$
                Exchange values with neighbors
                Compute stopping test
            End For
            If all stops true, stop
    End Iteration
```

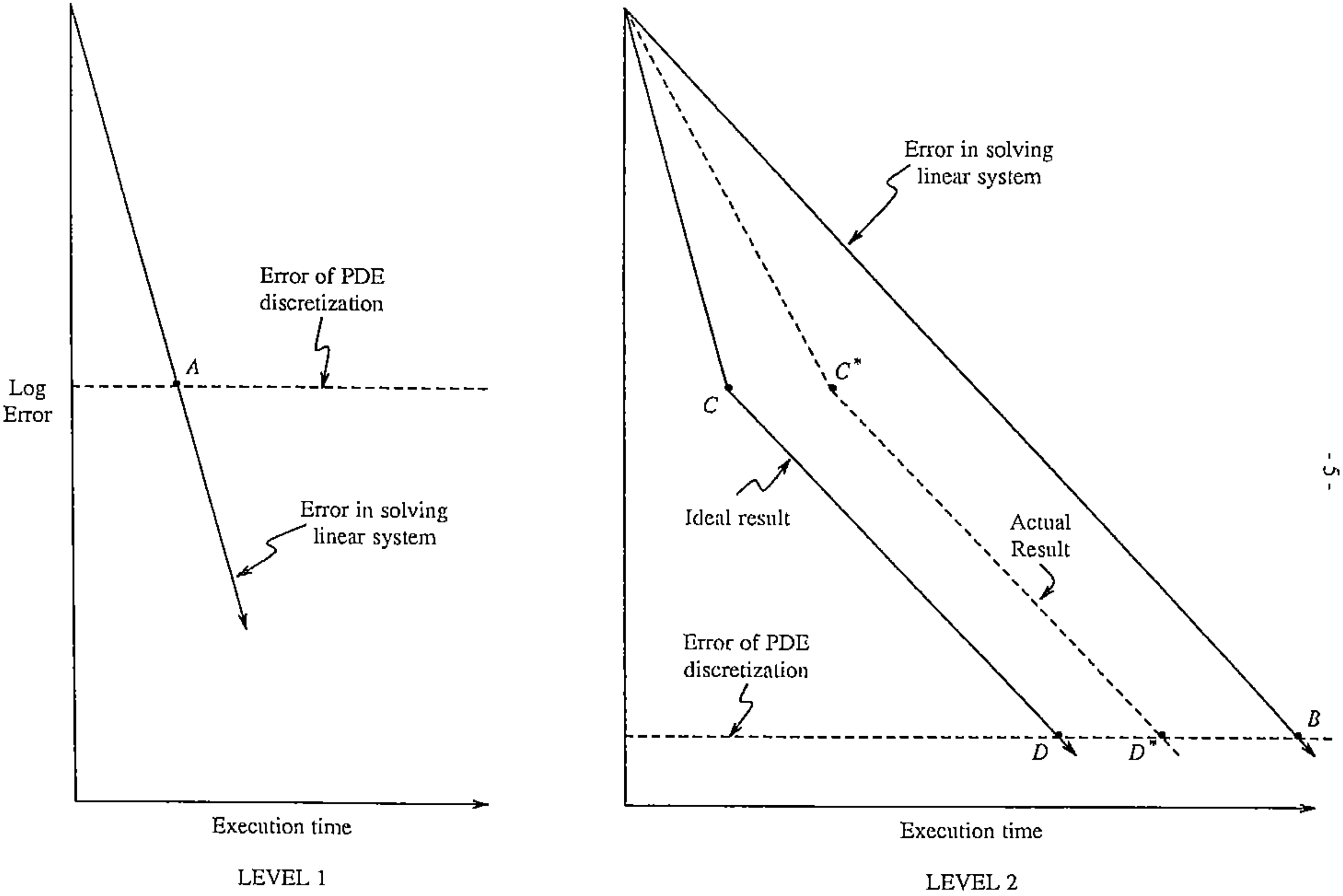

Log Error
Error of PDE discretization
A
Error in solving linear system
Execution time
LEVEL 1
Error in solving linear system
C*
C
Ideal result
Actual Result
Error of PDE discretization
D
D*
B
Execution time
LEVEL 2

Figure 2. *Schematic for the rationale of the two level algorithm. On Level 1, the point A is reached quickly where the PDE error and linear system error are equal. The partial solution of Level 1 is continually transfered to Level 2 to obtain faster convergence there (ideally at point C, but actually at point C^*). Once all the possible accuracy from Level 1 is transferred to Level 2, then the ordinary iteration continues until point D (or D^*) is reached.*

3. A Unified Model of a Parallel Computation

We have proposed [1] a nondeterministic model of computation which takes into account the load imbalance among processors executing in parallel. We have pointed out that algorithmic as well as non-algorithmic effects may contribute to load imbalance in case of a synchronized parallel execution. The algorithmic load imbalance effects can be minimized by redistributing computations among processors, and by overlapping communication with computation. But a perfect load balance is unattainable since non-algorithmic effects like hardware and software errors and retrys, or data dependent execution time, cause load imbalance. It is clear that such effects are more difficult to control and impossible to eliminate entirely. Though communication and control costs are likely to be the primary source of inefficiency for medium size parallel systems, the load imbalance will probably be a factor of increasing importance in massively parallel systems.

In our model we discuss the issue of load imbalance in the context of synchronization. We view a parallel computation as a sequence of n synchronization epochs each using I_i processors, and we express the expected execution time of a parallel computation as

$$T^* = E(T_c) = \alpha + n\beta + \sum_{i=1}^{n} \mu_i(1 + \Delta_i)$$

In this expression $\alpha = \alpha_1 + \alpha_2 \sum_{i=1}^{n} q(I_i)$ represents the cost due to control of the parallel computation. The communication costs are denoted by β. Note that β does reflect only the cost of communication which cannot be overlapped with computations. It is assumed that the execution time of the I_i processors active in any epoch i, $X_{i,j}$ are independent, identically distributed random variables with mean μ_i, and variance σ_i. The average cost attributed to the load imbalance in epoch i is $\Delta_i = \Delta\ (C_{i,}\ I_i)$, with C_i the coefficient of variation of the distribution of $X_{i,j}$, $C_i = \frac{\sigma_i}{\mu_i}$.

The expected serial execution time of a computation consisting of n synchronization epochs, with I_i processors active in each epoch, and with μ_i the expected execution time per processor in each epoch is:

$$T_S = \alpha_1 + \sum_{i=1}^{n} \mu_i I_i$$

Note that $\mu_i I_i$ is the expected total processor time per epoch, the time necessary for a single processor to carry out the parallel component of the computation.

In the framework of our model the *speed up with P processors*, S_P, ($I_i = P$ for all i) is defined as:

$$S_P = \frac{\alpha_1 + P \sum_{i=1}^{n} \mu_i}{\alpha + n\beta + \sum_{i=1}^{n} \mu_i(1 + \Delta_i)},$$

In general, the communication, control and load imbalance costs, α, β, and Δ depend upon the algorithm, the architecture of the parallel system, the number of processors executing in parallel.

As far as the load imbalance is concerned we have shown that for any distribution of the execution time the load imbalance costs for a synchronization epoch can be expressed as $\Delta_i = f(I_i) \times g(C_X)$ and, for several distributions, we have computed exact expressions for Δ_i [1]. For the uniform distribution $g(C_X) = C_X \sqrt{3}$ and $f(I_i) = (I_i - 1) / (I_i + 1)$. For the exponential distribution $g(C_X) \equiv 1$ and

$$f(I_i) = \log I_i + C \quad with \quad C = 0.577$$

For the standard normal distribution $g(C_X) \equiv 1$ and

$$f(I_i) \simeq (2 \log I_i)^{1/2} - \frac{1}{2} \left\{ 2 \log I_i \right]^{-1/2} (\log 4\pi - 2C - \log \log I_i)$$

4. A Model of the Two Level PDE Algorithm and its Implementation on a Hypercube

The model of the algorithm is constructed in two phases. First we study the algorithm and compare it with a standard, one level algorithm. We define the *algorithmic speed up* as the ratio of the execution time of the one level algorithm and the execution time of the two level algorithm on the same abstract machine.

Then we model the actual implementation of the algorithm on a parallel machine with a hypercube architecture, the NCUBE. We define the *actual speed up* as the ratio of the execution time on the NCUBE model when Level 2 runs alone as compared with the execution time when Level 1 and Level 2 execute concurrently and Level 1 feeds information to Level 2 as prescribed by our algorithm.

For the algorithmic model, let us use the following notations:

S_a - the algorithmic speed up,

m_α - the number of iterations required by a one level PDE algorithm,

T_α - the execution time per iteration for the one level algorithm on the hypothetical machine,

m_β - the number of iterations required by the tow level PDE algorithm,

T_β - the execution time per iteration for the two level algorithm on the same hypothetical machine.

Then, the algorithmic speed up is

$$S_a = \frac{m_\alpha T_\alpha}{m_\beta T_\beta}$$

As a first approximation $T_\alpha = T_\beta$ and S_a becomes m_α / m_β. From Figure 2, we note that m_α depends upon the error of the PDE discretization and the error in solving the linear system, hence upon the number of grid points. The value of m_β is a function of the number of grid points of the error at Level 1 and of the grid refinement factor q.

The second model involves a more detailed account of computation, communication and synchronization at each level. To describe the model, the following notations are used.

m_B - the iteration count required by a one level PDE algorithm,

T_B - the execution time for the one level algorithm,

T_B^a - the computation component of T_B, the time for arithmetic,

T_B^c - the communication component of T_B,

n - the number of grid points at Level 1,

$n \times q$ - the number of grid points at Level 2, q is the grid refinement factor,

t_a - the time for arithmetic operation per iteration, a typical value for t_a is 5,

f_{sy} - the synchronization factor,

t_c - the communication cost per unit of data transferred, a typical value for t_c is 2,

t_s - the start-up time for a communication act, a typical value for t_s is 200,

N - the total number of processors,

$N1$ - the number of processors assigned to Level 1,

$N2$ - the number of processors assigned to Level 2.

First we analyze the execution time of a one level algorithm, we have

$$T_B = T_B^a + T_B^c$$

with

$$T_B^a = m_B \times \left[\frac{(qn)^2}{N} \cdot t_a \cdot f_{sy} \right]$$

$$T_B^c = m_B \times \left[(qn + \log_2 N) \cdot 2t_c + (1 + \log_2 N) \cdot t_s \right]$$

Note that in our communication model, we take into account local and global communication. Global communication is done by broadcasting and its cost is logarithmic in the number of processors. We assume that local communication proceeds as follows: first, all processors communicate with their lower neighbors, then all communicate with the upper neighbors.

As noted, the synchronization costs depend upon the coefficient of variation of the distribution of the execution time and upon the number of processors executing in parallel. We assume here a uniform distribution with coefficient of variation C_X. Then

$$f_{sy} = 1 + C_X \sqrt{3}\ \frac{N-1}{N+1}.$$

For the analysis of the tow level algorithm, the following notation is used:

T_D - the execution time of the tow level algorithm,

T_D^a - the execution time corresponding to arithmetic (computation),

T_D^c - the execution time corresponding to communication,

m_C - the number of iterations required to reach the error level corresponding to point C in the graph of Figure 2,

m_{CD} - the number of iterations required from point C to point D in Figure 2.

With these notations, we have the following expressions for T_D^a and T_D^c.

$$T_D^a = (m_c + m_{CD}) \times \left[\frac{(qn)^2}{N2}\ t_a f_{sy} \right]$$

$$T_D^c = (m_c + m_{cD}) \times \left[(qn + \log_2(N2)) \cdot 2t_c + (1 + \log_2(N2)) \cdot t_s \right]$$

The *actual speed up* of the algorithm is defined as $S = T_B \,/\, T_D$.

5. Analysis of the Algorithm and its Performance on the NCUBE

The model of Section 4 is used to investigate the performance of the algorithm on the NCUBE. We study the range of speed ups that can be achieved (the algorithm behavior as the problem size increases), the influence of the grid refinement factor q, the effect of load inbalance, the choices of partitions for the hyypercube ($N1$ versus $N2$) and, finally, the effect of machine characteristics (t_a, t_c and t_s).

Figure 3 presents the variation of the algorithmic and of the actual speed up for a broad range of the number of grid points, $20 \leq n \leq 40000$. We see that the algorithm performs better for larger numbers of grid points, where the speed up is in the 6 to 7 range. This property of the algorithm is extremely important, since large problems require significant amounts of computing time.

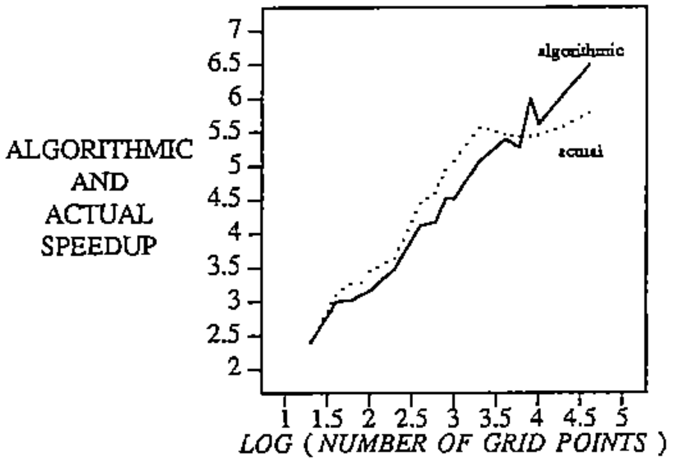


Figure 3. *The algorithmic speed up and the actual speed up function of the logarithm of the number of grid points. The algorithmic speed up is defined as the ratio of the number of iterations needed when Level 2 runs alone, versus the number of iterations at Level 2 when it receives data from Level 1. The actual speed up depends upon the machine parameters, the actual partitioning of processors between the two levels, and the coefficient of variation of the distribution of execution time. All processors assigned to Level 1 are idle during the second phase of execution, after transmitting data to Level 2. The data correspond to* $t_s = 200.0$, $t_c = 2.0$, *and* $C_X = 0.04$.

A rather surprising conclusion is that the algorithmic speed up is very close to the actual speed up, hence the rather simple model presented in the previous section can be used to predict the actual performance with a fairly good accuracy. The actual speed up tends to saturate, while the algorithmic speed up tends to show a linear increase with the number of grid points. A note of caution, Figure 3 summarizes the results for cases when the parameters of the algorithm, as well as the parameters of the implementation, have optimal values. More significant differences between the algorithmic and the actual speed up should be expected for non-optimal cases.

Figure 4 presents the effect of the grid refinement factor q. Optimal values for q are in the range from 2, for small to medium number of grid points, to 4 for large n. Qualitatively we expect that large values of q do not lead to optimal behavior of the algorithm, since the values fed by Level 1 are not accurate enough.

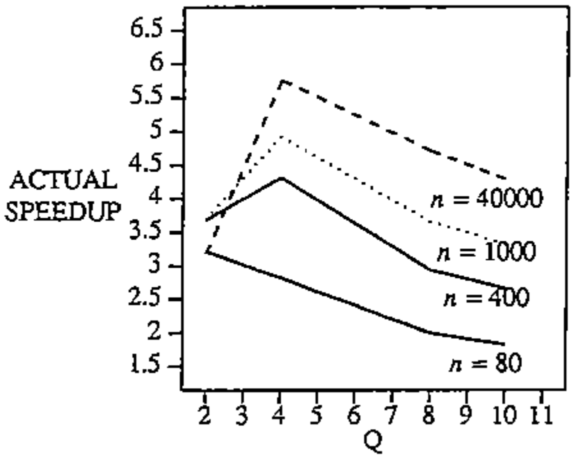


Figure 4. *The actual speed up function of q, the ratio of the number of grid points at Level 2 and Level 1. The number of grid points at Level 2 is n.*

Figures 5 and 6 outline the effects of the load imbalance. Figure 5 shows that the algorithm is stable with respect to load imbalance effects. The variation in the speed up from a totally balanced case, ($C_X = 0.0$), to the case of an imbalance characterized by $C_X = 0.2$, is rather small. Figure 6 shows that a one level algorithm is more sensitive to load imbalance effects. An increase of 25% of the execution time for the same load imbalance conditions can be expected. It is important to observe that the effect of load imbalance become more noticeable as the number of processors increases. We have examined only the case $N = 512$ processors.

Finally, Figures 7 and 8 focus on the algorithm implementation issues and the interaction of the algorithm with the NCUBE. Figure 7 summarizes the effect of the hypercube partitioning upon the execution time of the algorithm. We see that for optimal values of q ($q = 4$) and for medium to large numbers of grid points, ($n = 800$, and $n = 8000$), the optimal partitioning of the machine requires that at least 32 processors of a 512 processor machine be assigned to Level 1. If a smaller number of processors is assigned to Level 1, then the algorithm performs poorly, since Level 2 wastes too much time before receiving the data from Level 1. If too many processors are assigned to Level 1, then the performance tends to decrease again. All processors assigned to Level 1 become idle after transmitting their values to Level 2 and this effect becomes dominant.

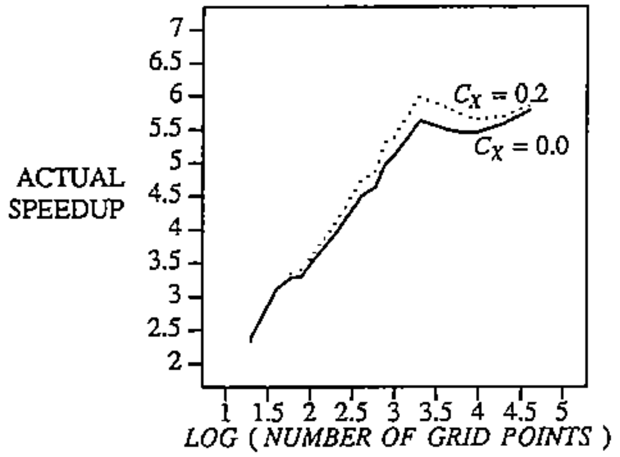


Figure 5. *The actual speed up function of the logarithm of the number of grid points for two load imbalace situations. The coefficient of variation of the execution time is denoted by C_X. The algorithm seems stable with respect to load imbalance.*

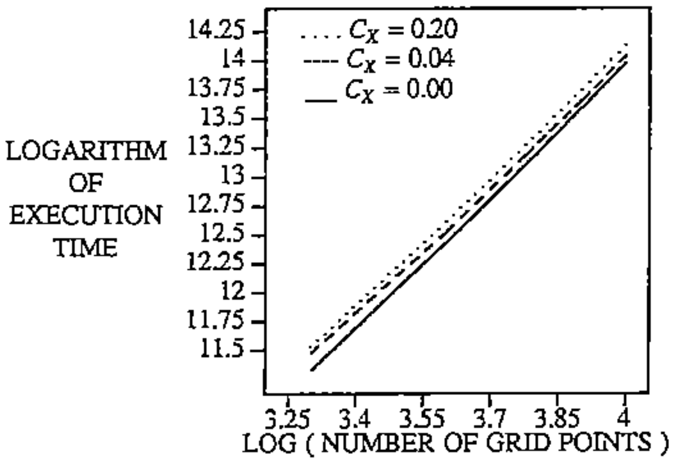


Figure 6. *The total execution time function of the number of grid points for different load imbalance factors. C_X is the coefficient of variance of the execution time on a processor. The number of processors is $N = 2048$.*

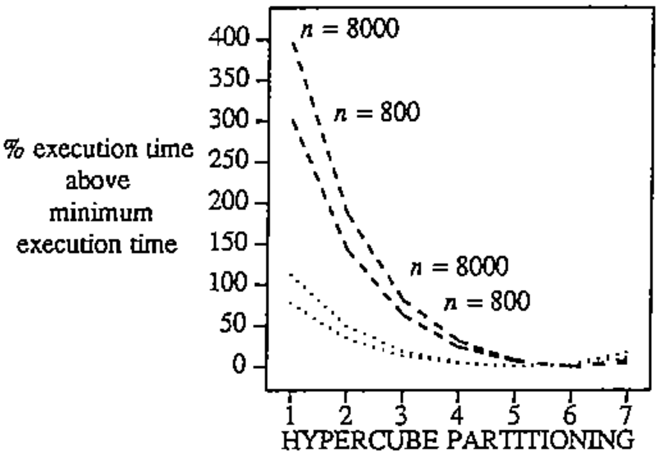


Figure 7. *The effect of hypercube partitioning upon the total execution time of the algorithm.* $N = 512$ *processors, and* $q = 4$ *(dashed) and* $q = 10$ *(dotted). Two cases are presented, the number of grid points* $n = 800$ *and* $n = 8000$. *We show the percentage [T- min (T)]/ min (T), where T is the execution time and the minimum execution time is for different partitionings, the abscissa* $k = N_1/N_2$. N_1 *is the number of processors assigned to Level 1 and* N_2 *is the number of processors assigned to Level 2. Note that:* $N_1 + N_2 = N$. *The points on the graph correspond to:*

1 => k = 2/510 *2 => k = 4/508* *3 => k = 8/504* *4 => k = 16/496*
5 => k = 32/480 *6 => k = 64/448* *7 => k = 128/384*

Figure 8 shows the effect of the machine parameters. We compare a configuration with fast communication ($t_c = 1,\ t_s = 50$) to a much slower one ($t_c = 4,\ t_s = 400$). Surprisingly the speed up is larger in the latter case. Clearly, this does not mean that a slower machine performs better than a fast one, but that the tow level algorithm is capable of absorbing the effect of slow communication better than a one level algorithm on the same machine.

In conclusion, the tow level splitting algorithm leads to a reasonable speed up, performs well for large problem sizes, is stable with respect to load imbalance and is well suited for hypercube machines which tend to have slow communication.

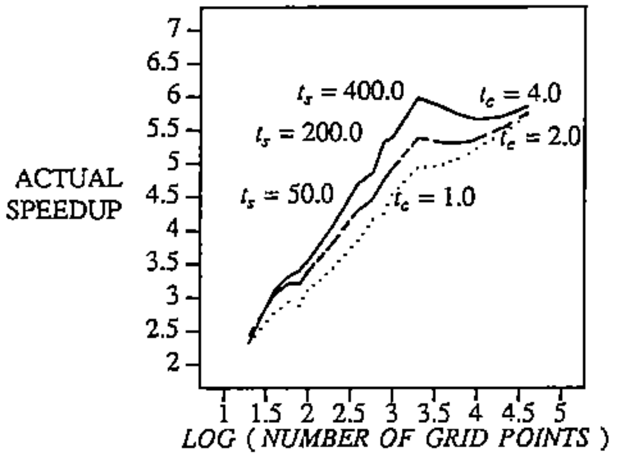


Figure 8. *The actual speed up function versus the logarithm of the number of grid points for three machine parameters. t_s is the communication start-up time and t_c is the communication time per unit of message. Note that the algorithm seems more suited for machines with slower communication, like the NCUBE. The coefficient of variation of the execution time is $C_X = 0.2$.*

Literature

[1] D.C. Marinescu and J.R. Rice, "On the effects of synchronization in parallel computating", CSD-TR-750 Computer Sciences, Purdue University, March 1988.

[2] D.C. Marinescu and J.R. Rice, "Synchronization of nonhomogeneous parallel computations", Proceedings of *Parallel Processing for Scientific Computing*, SIAM, 1988 (to appear).

[3] D.C. Marinescu and J.R. Rice, "Domain oriented analysis of PDE splitting algorithms", *Information Sciences*, 43, pp. 3–24, 1987.

[4] J.R. Rice and D.C. Marinescu, "Analysis and modeling of Schwartz splitting algorithms for elliptic PDEs", in *Advances in Computer Methods for Partial Differential Equations*, VI (Stepleman and Vishnevetsky, eds.), IMACS, Rutgers University, pp. 1–6, 1987.

[5] J.R. Rice et. al., "Experiments on asynchronous multiprocessor methods for PDEs", CSD-TR-799, Computer Sciences, Purdue University, October 1988.