SANDIA REPORT

SAND88—2714 • UC—32

Unlimited Release

Printed December 1988

The Performance of Asynchronous Algorithms on Hypercubes

David E. Womble

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550
for the United States Department of Energy
under Contract DE-AC04-76DP00789

When printing a copy of any digitized SAND Report, you are required to update the markings to current standards.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof or any of their contractors or subcontractors.

Printed in the United States of America
Available from
National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Road
Springfield, VA 22161

NTIS price codes
Printed copy: A02
Microfiche copy: A01

The Performance of Asynchronous Algorithms on Hypercubes

David E. Womble

Numerical Mathematics Division, 1422
Sandia National Laboratories
Albuquerque, NM 87185

Abstract

Many asynchronous algorithms have been developed for parallel computers. Most implementations of asynchronous algorithms, however, have been for shared memory machines. In this paper, we study the implementation and performance of some common asynchronous algorithms on the NCUBE/ten, a 1024 node hypercube. In addition, we summarize existing theoretical work and discuss some classes of algorithms that can be made asynchronous and some that cannot.

*This work was performed at Sandia National Laboratories supported by the U.S. Department of Energy under contract No. DE-AC04-76DP00789.

1 Introduction

One way to achieve the computing power required by scientists and engineers is through massively parallel computers, i.e., computers that have more than one thousand processors. However, for massively parallel computers to be used effectively, new algorithms must be developed and studied. In this paper we study the performance of a class of new algorithms, asynchronous algorithms, which can improve the performance of existing synchronous algorithms by reducing the amount of time a processor spends waiting on other processors.

Many of the parallel computers available today are MIMD (multiple instruction, multiple data) type machines; that is, the processors are allowed to operate independently of one another. It is natural, then, that algorithms designed for MIMD machines should minimize the synchronization and communication between processors. Algorithms in which synchronization has been completely eliminated are most often called *asynchronous algorithms*, although they were originally known as chaotic relaxation algorithms [5].

MIMD computers can be either shared memory machines or distributed memory machines. In a shared memory machine, communication takes place only through access to a common memory, and synchronization is usually accomplished through a set of semaphores or memory locks. In a shared memory machine, memory conflicts are often a major source of program delays.

In a distributed memory machine, on the other hand, each processor has access only to a local memory. Communication takes place by passing messages over communication links between processors. While memory conflicts are eliminated in distributed memory machines, the communication of information can result in message conflicts or major program delays. Synchronization in these machines is "pairwise": one processor must wait for information to be sent by another processor.

Asynchronous algorithms are most often implemented on shared memory machines. Because the communication is implicit (by memory reference) and the synchronization is explicit, an iterative algorithm can be made asynchronous simply by removing synchronization points. In a distributed memory machine, however, the synchronization is implicit in the communication. To implement an asynchronous algorithm, therefore, one processor must be able to send a message without waiting for the receiving processor to read it,

and a processor must be able to check for the presence of a message without having to read one.

Asynchronous algorithms have been applied to a wide variety of problems. These include combinatorial problems (searching, sorting, etc...), minimization problems, operating system problems (task scheduling, memory management, etc...), and the solution of linear and nonlinear systems of equations. In this paper, we consider only the latter class of problems.

In 1969, Chazan and Miranker [5] published an asynchronous relaxation algorithm for the solution of the linear system $Ax = b$ and gave conditions on A under which the algorithm converged. In [10], Miellou generalized the results of Chazan and Miranker to nonlinear contracting operators, while Baudet [3] generalized the definition of asynchronous algorithm and reestablished the convergence results of [5] and [10]. Finally El Tarazi [7] restated the convergence results of [3] so that they were similar to classical fixed point theorems. Similar theorems were given by Bertsekas [4]. Many authors have used the results of [3,7,4] or have used similar methods to establish the convergence of specific algorithms. These include [1,15,16,11,12].

The work mentioned above has developed around the formalism of Baudet. There are some alternate models, which we mention here. The first is the transition model [8], which is based on the idea of flow descriptions in software design [14]. This formalism has resulted in at least one convergence theorem. A second model is the queueing network model [13,6]. This model seems most useful for statistical performance analysis of asynchronous algorithms.

Most of the work cited so far has been strictly theoretical, although some authors have included simulations in their work [9,16]. The lack of experimental results has been due, in part, to the limited availability of MIMD computers. A notable exception is the work of Baudet [3], who studied the implementations of several algorithms on the Carnegie-Mellon multiprocessor computer. Experimental results including comparisons with synchronous algorithms can also be found in [2,1]. All of these experiments were carried out on shared memory machines with a small number of processors.

It is difficult to make general statements about the performance of asynchronous algorithms: performance results usually apply only to a specific algorithm on a specific parallel computer. Nevertheless, in this paper, we try to present some intuitive guidance on the performance of asynchronous algorithms on distributed memory machines based on several examples.

We begin by stating a formal definition of asynchronous algorithm in Section 2 and giving some general convergence results in Section 3. Following that, in Sections 4 and 5, we present several specific asynchronous algorithms and discuss their implementation and experimental results on the NCUBE/ten hypercube. We also discuss a class of algorithms whose asynchronous versions do not appear effective.

2 Theory of Asynchronous Algorithms

We motivate the formal definition of asynchronous algorithms with an example. Gauss-Seidel relaxation is an iterative method for finding the solution to an equation of the form $u = G(u)$, where $u = (u_1, \dots, u_p)$, and $G(u) = (G_1(u), \dots, G_p(u))$. The Gauss-Seidel iterates are given by the relaxation equation

$$u_k^{i+1} = G_k(u_1^{i+1}, \dots, u_{k-1}^{i+1}, u_k^i, \dots, u_p^i), \quad k = 1, \dots, p,$$

where i denotes the iteration number.

We see that the computation of the u_k^{i+1} must be carried out in a specific sequence. On a parallel computer, the result is that processors spend a lot of time waiting for other processors to carry out computations. If we remove the requirement that the relaxations be carried out in sequence, the result is an asynchronous version of the algorithm. We see that any formal definition of asynchronous algorithm must allow several components to be updated simultaneously and must allow "old" values of the u_k to be used if new values have not yet been computed. Hence, we introduce an update set and a delay set, denoted by J and S , in the definition below.

The following definition is given in [3,7] and is the most useful definition for the analysis of scientific algorithms. In the following, $\mathbf{R}$ denotes the real numbers, and $\mathbf{N}$ denotes the natural numbers.

We let $E_i = \mathbf{R}^{n_i}$, where $n_i \in \mathbf{N}$, $i = 1, \dots, p$, and let $\|\cdot\|_i$ denote a norm on E_i , and

$$E = \prod_{i=1}^p E_i = \mathbf{R}^n, \quad n = \sum_{i=1}^p n_i.$$

Elements $u \in E$ have the form $u = (u_1, u_2, \dots, u_p)$ with norm

$$\|u\| = \max_i \frac{1}{\gamma_i} \|u_i\|_i,$$

where $\gamma_i \in \mathbb{R}$, $\gamma_i > 0$, $i = 1, \dots, p$. (In a more abstract setting, E_i can be a Banach space.)

We are interested in fixed points of an operator $G : D(G) \subset E \rightarrow E$, where $G(u)$ has the form

$$G(u) = (G_1(u), G_2(u), \dots, G_p(u)), \quad G_k : D(G) \subset E \rightarrow E_k.$$

An *asynchronous algorithm* corresponding to the operator G and starting with the vector $u^0 \in D(G)$ is the sequence $\{u^i\}_{i \in \mathbb{N}}$ of vectors in E defined by

$$u_k^{i+1} = \begin{cases} G_k(\dots, u_l^{s_l(i)}, \dots) & k \in J(i) \\ u_k^i & k \notin J(i), \end{cases} \quad k = 1, \dots, p,$$

where $J = \{J(i)\}_{i \in \mathbb{N}}$, $J(i) \subset \{1, \dots, p\}$ has the property that k occurs infinitely often in the sets $\{J(i)\}_{i \in \mathbb{N}}$, and $S = \{s_1(i), \dots, s_p(i)\}$ is a sequence of elements of $\mathbb{N}^p$ satisfying

$$s_k(i) \leq i, \quad i \in \mathbb{N},$$

$$\lim_{i \rightarrow \infty} s_k(i) = \infty.$$

An asynchronous algorithm is denoted by the quadruplet (G, u^0, J, S) .

The sets J and S characterize the asynchronous algorithm. The set $J(i)$ is the set of components to be updated at iteration i , and the requirement that k occurs infinitely often in the $J(i)$ means that no component can be ignored during the iteration. The number $s_k(i)$ is the delay for component k at iteration i . The requirements on $s_k(i)$ mean that we cannot use information that has not yet been generated by the iteration and that we must use “reasonably current” information in the iterations.

We see that the Gauss–Seidel iteration described at the beginning of this section can be put into the context of this definition with the following assignments for J and S :

$$s_k(i) = i \quad \text{for } k = 1, \dots, p, \text{ and } i \in \mathbb{N},$$

$$J(i) = 1 + i \bmod p \quad \text{for } i \in \mathbb{N}.$$

In fact, any synchronous algorithm can be written as an asynchronous algorithm with the proper choices for J and S .

We can now state a convergence theorem for asynchronous algorithms. The following is found in [7].

Theorem 2.1 *We let G be an operator mapping $D(G) \subset E$ into E , where*

$$D(G) = \prod_{k=1}^p D_k(G), \quad D_k(G) \subset E_k.$$

We suppose that G has a fixed point, u^ , in $D(G)$, $G[D(G)] \subset D(G)$, and for all $u \in D(G)$*

$$\|G(u) - G(u^*)\| \leq \beta \|u - u^*\|, \quad 0 < \beta < 1.$$

Then u^ is the unique fixed point of G in $D(G)$, and any asynchronous algorithm (G, u^0, J, S) corresponding to G and starting with $u^0 \in D(G)$ converges to u^* .*

Theorem 2.1 is similar in nature to many of the classical convergence theorems for fixed point algorithms. However, even though the iteration scheme in Theorem 2.1 is more general than that of most fixed point algorithms, the application of the theorem is limited to operators that are contractions in the norm $\|\cdot\|$.

3 Algorithms for Linear Systems

In this section, we define three asynchronous algorithms: the names are based on the synchronous algorithms from which they are derived. The first algorithm, Jacobi relaxation, was chosen because it represents a class of algorithms that can be implemented efficiently on a parallel computer. The second algorithm, Gauss-Seidel relaxation, represents a class of algorithms that cannot be implemented efficiently on a parallel computer. The third algorithm, steepest descent, represents a class of algorithms that cannot be implemented asynchronously without making fundamental changes to the algorithm.

We have combined the asynchronous Jacobi and asynchronous Gauss-Seidel algorithms into one description because of the similarities. In Algorithm 3.1 below, the coordinate functionals, $G_k(x)$, of $G(x)$ are specified. The delays, S , are not known a priori: if there is one unknown assigned to

each processor, then the delays are determined by the relative speeds of the processors. If there is more than one unknown assigned to each processor, then the delays enforced within each processor determine whether the algorithm is an asynchronous Jacobi (AJ) or an asynchronous Gauss-Seidel (AGS) algorithm. Specifically, if a processor uses the most recent values of the unknowns assigned to it during each iteration, it is an AGS algorithm. Otherwise, it is an AJ algorithm.

Although Algorithm 3.1 is formulated for linear systems, it can be formulated for any system to which Jacobi relaxation can be applied.

Algorithm 3.1 (AJ, AGS) *We consider the linear system*

$$Ax = b, \quad (1)$$

where A is an $n \times n$ matrix with entries $(a_{i,j})$, and define the operator $G(x) : \mathbf{R}^n \rightarrow \mathbf{R}^n$ by the coordinate functionals

$$G_k(x) = \left(b_k - \sum_{j=1}^{k-1} a_{k,j}x_j - \sum_{j=k+1}^n a_{k,j}x_j \right) / a_{k,k}, \quad k = 1, \dots, n, \quad (2)$$

where x_k denotes the k^{th} element of x . The starting guess, $x^0 \in \mathbf{R}^n$ is arbitrary, as are the sets J and S (although J and S must always satisfy the definition of asynchronous algorithm).

The convergence of Algorithm 3.1 is easy to establish for a variety of linear systems. The following convergence result demonstrates the use of Theorem 2.1.

Theorem 3.1 *If A is a diagonally dominant, $n \times n$ matrix, then Algorithm 3.1 converges to the unique solution of (1).*

Proof. We begin by writing $A = L + D + U$, where L , D and U are, respectively, the lower triangular, diagonal and upper triangular components of A . The function $G(x)$, defined in Algorithm 3.1, can then be written as

$$G(x) = -D^{-1}(L + U)x + D^{-1}b.$$

Because A is diagonally dominant, A^{-1} exists. Thus, there is a unique solution, x^* , to (1), and, by inspection, x^* is a fixed point of G . We now write

$$G(x) - G(x^*) = -D^{-1}(L + U)(x - x^*).$$

Because A is diagonally dominant,

$$\|D^{-1}(L + U)\|_{\infty} = \max_{1 \leq i \leq n} \left(\sum_{\substack{1 \leq j \leq n \\ i \neq j}} |a_{i,j}|/|a_{i,i}| \right) = \beta < 1.$$

It is clear that $G(x)$ satisfies the conditions of Theorem 2.1, with the choices $E_i = \mathbf{R}$, $i = 1, \dots, n$, and $\|\cdot\|_i = |\cdot|$. We conclude that Algorithm 3.1 converges to the unique solution of (1). $\square$

Corollary 3.1 *If A is a diagonally dominant, $n \times n$, matrix, then both Jacobi iteration and Gauss-Seidel iteration converge to the unique solution of (1).*

Proof. Both Jacobi iteration and Gauss-Seidel iteration are asynchronous algorithms. For Jacobi iteration, J and S are given by

$$s_k(i) = i \quad \text{for } k = 1, \dots, p, \text{ and } i \in \mathbf{N},$$

$$J(i) = \{1, \dots, n\} \quad \text{for } i \in \mathbf{N},$$

while for Gauss-Seidel iteration, J and S are given by

$$s_k(i) = i \quad \text{for } k = 1, \dots, p, \text{ and } i \in \mathbf{N},$$

$$J(i) = 1 + i \bmod p \quad \text{for } i \in \mathbf{N}.$$

The convergence of Jacobi iteration and Gauss-Seidel iteration now follows from Theorem 3.1. $\square$

We conclude our discussion of the AJ and AGS algorithms with another convergence theorem. This theorem establishes the convergence of Algorithm 3.1 for a class of matrices that arise in the finite difference discretizations of many elliptic operators, such as the Laplacian operator.

Definition 3.1 *A matrix A is an M -matrix if the diagonal entries of A are positive, the off-diagonal entries of A are negative, and the entries of A^{-1} are positive.*

Theorem 3.2 *If A is a symmetric, positive definite (SPD), M -matrix, then Algorithm 3.1 converges to the unique solution of (1).*

The proof of Theorem 3.2 can be found in [5].

Our second algorithm, Algorithm 3.2 below, is derived from classical steepest descent and is called asynchronous steepest descent (ASD). As before, we state the algorithm for a linear system.

Algorithm 3.2 (ASD) *We consider the linear system (1), where A is an $n \times n$, SPD matrix with entries $(a_{i,j})$, and b is a vector in $\mathbf{R}^n$ with entries (b_i) . For $I = \{1, \dots, n\}$ and $s_1, s_2 \subset I$, we define the matrix*

$$A_{s_1 \times s_2} = \begin{cases} a_{i,j} & i \in s_1, j \in s_2 \\ 0 & \text{otherwise} \end{cases},$$

and the vector

$$b_{s_1} = \begin{cases} b_i & i \in s_1 \\ 0 & \text{otherwise} \end{cases}.$$

Finally, we denote by g_i , $i = 1, \dots, p$, subsets of I satisfying $\bigcup_{i=1}^p g_i = I$. ASD is defined by $G(x) : \mathbf{R}^n \rightarrow \mathbf{R}^n$, where the coordinate functionals of $G(x)$ are given by

$$G_k(x) = x_{g_k} + \alpha_k r_{g_k}, \quad k = 1, \dots, p, \quad (3)$$

where

$$r_{g_k} = b_{g_k} - A_{g_k \times I} x = (b - Ax)_{g_k},$$

and

$$\alpha_k = \frac{r_{g_k}^T r_{g_k}}{r_{g_k}^T A_{g_k \times g_k} r_{g_k}}.$$

The starting guess $x^0 \in \mathbf{R}^n$ is arbitrary, and the sets J and S are required only to satisfy the definition of asynchronous algorithm.

We note here that in general

$$b_{g_k} - A_{g_k \times I} x \neq r_{g_k} - \alpha_k A_{g_k \times g_k} r_{g_k}.$$

This implies that in the case of more than one group, we cannot eliminate one of the matrix multiplies. In the case of a large number of groups, this

is not a large penalty because a full matrix multiply is not required in the computation of α_k .

We see from the definition of Algorithm 3.2 that if $p = n$, and $g_i = \{i\}, i = 1, \dots, n$, then we recover Algorithm 3.1. Algorithm 3.2 can, in fact, be viewed as a hybrid of block Jacobi and steepest descent. The convergence of Algorithm 3.2 has not been established due to the restriction on norms in Theorem 2.1.

4 Implementations on a Hypercube

The NCUBE/ten at Sandia National Laboratories contains 1,024 processing nodes in a hypercube configuration. Each node contains a processor, a hardware floating point unit with 32-bit and 64-bit IEEE floating point arithmetic, 512 Kb of ECC memory, and eleven bidirectional DMA channels: ten are used for communications within the hypercube; the eleventh provides a connection to an Intel 80286 host processor or other I/O device.

In the Introduction, we stated two conditions that must be satisfied by the hardware of a distributed memory machine in order to implement an asynchronous algorithm. Specifically, a processor must be able to send a message to another processor without waiting for the receiving processor to read the message, and a processor must be able to test for a message from another processor without waiting for that processor to write a message. These conditions must also be satisfied by any language in which an asynchronous algorithm is to be written. The NCUBE/ten and its implementation of FORTRAN satisfy both conditions.

Computation and communication times for the NCUBE depend on both the program and the language of implementation. A computationally intensive, single-node, double precision FORTRAN program can achieve 0.07 to 0.13 MFLOPS. Communication between nodes from within a FORTRAN program occurs at about 0.5 Mb/sec with a setup time of about 350 μ s required for each read and write. Testing for the presence of a message also requires about 350 μ s. Clearly, communication is a time consuming operation, and any efficient implementation of an algorithm will minimize first, the number of communications and second, the number of bytes in each communication, even at the expense of some redundant computation.

The implementation of an algorithm on the NCUBE/ten requires two

programs: a host program and a node program. The host program is responsible for such tasks as allocating all or part of the ten-dimensional hypercube, sending the node program to each processor in the allocated hypercube, and handling all input to and output from the nodes. The node program is responsible for all the computation and communication within the cube.

We now turn our attention to an implementation of Algorithm 3.1. We assume that there are p processors available for the solution of the linear system (1). We also assume, for ease of exposition, that n , the number of unknowns, is an integer multiple of p . Thus, each processor is responsible for evaluating $m = n/p$ of the $G_k(x)$, the coordinate functionals of $G(x)$. Looking at equation (2), we see that if each processor stores the vector x and the m rows of the matrix A corresponding to coordinate functionals that it must evaluate, the evaluations can be carried out without any communication. Once a processor has carried out the evaluation of a coordinate functional, the new value of x must be communicated to other processors. If the matrix is sparse, this information should be communicated to only those processors that need it.

We can now give a pseudocode version of Algorithm 3.1.

Host program:

1. Allocate a hypercube and load the node programs
 2. Send rows of the matrix to each node
 3. For each node, do
 4. Determine what communication is required
 5. Send this communication information to the node
 6. End for
 7. Send a "start computing" message to the nodes
 8. Wait for nodes to compute the solution
 9. Send a "stop computing" message to the nodes
 10. For each node, do
 11. Read the solution and print it
 12. End for
 13. Close the hypercube
- End program

Node program:

1. Read rows of the matrix from the host

2. Read communication information from the host
3. Read a "start computing" message from the host
4. While no "stop message" pending, do
 5. While messages pending, do
 6. Read a message
 7. Update the solution vector
 8. End while
 9. For each coordinate functional this node is responsible for, do
 10. Evaluate the coordinate functional, $G_k(x)$
 11. Update the solution, x
 12. End for
 13. For each processor that needs the new solution, do
 14. Send the new solution
 15. End for
16. End while
17. Send the solution to the host

End program

In the node program, it would be possible to move the loops 5-8 and 13-15 into the loop 9-12. This would have the effect of reducing the delay in communicating information to other processors, thus reducing the overall computation time. However, it would also have the effect of increasing the number of messages and the number of tests for pending messages that a processor must make. In the experimental codes that we have written for the NCUBE, we have found that, in most cases, the pseudocode listed above is most efficient. (In the cases where we have a very sparse matrix and are using a small number of processors, the alternate version would be more efficient. This is also true for shared memory machines [3].)

In the ASD algorithm, processor i is responsible for computing the components of the solution corresponding to the group g_i . As in the AJ algorithm, each processor needs to know the entire solution vector, x , and the rows of the matrix indexed by the group g_i . If the g_i are not pairwise disjoint, then either some rows of the matrix must be stored more than once, or there must be additional communication between processors. If we assume that the g_i are pairwise disjoint, then the pseudocode for processor i in the implementation of ASD is the same as that for AJ with the loop 9-12 replaced by

9. Compute $G_k(x)$ according to equation (3)

10. Update the solution x

One issue of practical importance, which requires further research, is that of stopping criteria. Most stopping criteria involve a global check of a residual; however, this requires the synchronization of processors. Thus, for an asynchronous algorithm, a distributed stopping criterion, which would allow processors to independently determine if convergence has been achieved, is needed. Such stopping criteria would also improve the performance of some synchronous algorithms by removing one synchronization and communication step.

A couple of programming points should also be noted. The first is that of node-host communication. In our AJ, AGS and ASD algorithms, the host must monitor the nodes and decide when to send the “stop computing” message; however, if there is too much communication, it is possible that the host will not be able to respond fast enough and will abort the program.

The second programming issue is what a processor does between the time that it has converged and the time that it receives the “stop computing” message from the host or more data from other nodes. In some algorithms, such as steepest descent, division by the norm of a residual is required. Once a processor has converged, this will result in a “divide by zero” condition that should not result in the termination of the program.

5 Numerical Results

We have implemented the AJ, AGS and ASD algorithms described in the previous section. In this section, these algorithms will be compared with their synchronous counterparts. As a test problem for each algorithm, we have chosen the linear system arising from the discretization of the Laplacian operator on the unit square using a nine point star with a 65×65 grid (4,096 unknowns). This problem size was chosen because it is the largest problem that can be stored in the memory of one processor with our algorithms.

As a stopping criterion, we use

$$\frac{\|u_c - u_t\|_\infty}{\|u_t\|_\infty} < \text{TOL},$$

where u_c is the computed solution and u_t is the true solution, which is known for the test problems. While this criterion can not be used in practice,

P	Time (s)		% Communication		Iterations	
	J	AJ	J	AJ	J	AJ
1	6,639	6,639	0.0	0.0	5,271	5,271
4	1,769	1,740	7.1	1.7–3.3	5,271	5,338–5,525
16	462	470	15.6	2.8–5.0	5,271	5,643–6,213
64	150	149	49.1	11.9–14.7	5,271	6,301–9,028
256	147	114	86.8	59.4–69.5	5,271	5,271–10,585
1,024	267	159	98.1	91.7–95.0	5,271	5,349–7,822

Figure 1: Comparison of the Jacobi and asynchronous Jacobi methods

it enables us to compare the algorithms directly without introducing any additional synchronization.

In each run of each program, we measured the number of iterations required for convergence, time required for communications on each processor (which includes synchronization time), and the total run time. Based on the measured times, we can compute the percentage of time spent in communications. For an asynchronous algorithm, we quote a range of values for the number of iterations and percent communication. This is of interest because some processors require less communication than others, and if less communication is required, more computations can be performed.

Figure 1 compares the Jacobi and asynchronous Jacobi algorithms. We make several observations here. First, because the synchronization has been removed from the AJ algorithm, the communication to computation ratio can be computed by “% Communication”/(1–“% Communication”). This ratio is fixed (for each processor) for a given problem and number of processors. Thus, the time spent idle by a processor in the Jacobi algorithm is the difference between the % Communication for that processor for the J and AJ algorithms. Based on the numbers in Figure 1, we can compute the maximum and minimum processor idle times for the Jacobi algorithm.

We also observe that there is a substantial increase in the number of iterations required by the asynchronous Jacobi method, which is caused by increasing delays, S , changes in the order in which the unknowns are updated, J , and the amount of communication required of a processor. This increase in

P	Time (s)		% Communication		Iterations	
	GS	AGS	GS	AGS	GS	AGS
1	3,190	3,190	0.0	0.0	2,639	2,639
4	1,688	891	52.7	1.7-3.4	2,639	2,819-2,920
16	431	276	56.2	2.9-5.2	2,639	3,409-3,765
64	126	119	71.0	12.4-15.2	2,639	5,151-7,481
256	134	94	93.0	59.6-70.1	2,639	4,363-8,872
1,024	262	136	99.0	91.8-95.2	2,639	4,596-6,731

Figure 2: Comparison of the Gauss-Seidel and asynchronous Gauss-Seidel methods

the number of iterations offsets the reduced communication cost, and the net effect is that the Jacobi method is competitive with the asynchronous Jacobi method for a small number of processors. (Because there is a relatively low synchronization cost, Jacobi's method is often referred to as embarrassingly parallel.) For large numbers of processors, the communication costs dominate the total time, and small differences in the communication costs are seen as a major difference in the overall times.

Finally, we note that there is a large jump in the percentage of time spent in communication in the asynchronous Jacobi algorithm for more than 64 processors. This corresponds to the case when there is exactly one row of unknowns per processor. If there are more than 64 processors, there is more than one processor per row of unknowns and the solution within a processor must be communicated to almost twice as many other processors. This is a result of the relatively coarse-grained parallelism of our implementations.

In Figure 2, we compare the Gauss-Seidel and asynchronous Gauss-Seidel methods. Our first observation here is that the percentage of time spent in communication in the AGS algorithm is approximately the same as that for the AJ algorithm. This is the case because, even though we are performing slightly different computations, the number of communications and the number of floating point operations is the same for the two algorithms. (The communication to computation ratio is the same.)

Our second observation is that because the Gauss-Seidel algorithm requires that the unknowns be updated in a fixed order, there is at least a

P	Time (s)		% Communication		Iterations	
	SD	ASD	SD	ASD	SD	ASD
1	12,198	12,198	0.0	0.0	3,294	3,294
4	3,195	3,517	5.4	0.6-1.1	3,294	3,759-3,826
16	824	889	12.7	1.0-1.8	3,294	3,876-4,118
64	250	221	42.6	4.8-6.3	3,294	4,132-5,465
256	199	113	81.5	37.7-47.6	3,294	3,985-6,593
1,024	395	114	97.4	83.3-88.0	3,294	2,948-5,226

Figure 3: Comparison of the steepest descent and asynchronous steepest descent methods

50% synchronization penalty for two or more processors even though the algorithm allows different processors to compute simultaneously on different iterations. The result is that the asynchronous algorithm is substantially more efficient for a small number of processors.

Figure 2 demonstrates the advantages of an asynchronous algorithm when there is a large synchronization penalty, while Figure 1 demonstrates that even when there is a small synchronization penalty, the performance of an asynchronous algorithm is competitive.

Our final comparison, shown in Figure 3, is between the steepest descent (SD) and asynchronous steepest descent (ASD) algorithms. This comparison differs from the previous comparisons in two ways. First, as more groups are added, the character of the asynchronous algorithm changes. In fact, if there is only one point per group, the algorithm is the same as the asynchronous Jacobi algorithm. Second, in the synchronous steepest descent algorithm, a second communication stage is required to calculate the full residual.

Even with the second communication step, we note that the behavior of the asynchronous steepest descent is much like that of the asynchronous Jacobi algorithm. There is more work during each iteration of steepest descent than during each iteration of Jacobi. Thus, if there is only a small synchronization penalty, but a fairly large increase in the number of iterations for the asynchronous algorithm, then the synchronous algorithm may actually run faster. Based on Figure 3, we see that this is the case, but only for a small number of processors. If there are enough processors, the asynchronous

algorithm will eventually be the faster algorithm.

6 Conclusions

There are many possible asynchronous algorithms. We have presented three, each of which has different features. These algorithms point out some of the advantages, as well as disadvantages, of asynchronous algorithms.

In a distributed memory machine, the communications are relatively slow and must be carried out whether the algorithm is synchronous or asynchronous. Thus, if an algorithm has a small synchronization penalty and small number of communications, the synchronous algorithm can perform as well as the asynchronous algorithm. On the other hand, if an algorithm requires that a large number of small messages be sent, the asynchronous algorithm can perform substantially better. This was shown in Figure 1.

If there is a large synchronization penalty, as in the case of the Gauss-Seidel algorithm (Figure 2), the asynchronous algorithm performs better than its synchronous counterpart on a small number of processors. If enough processors are used, we return to the case where the number of communications dominates the time. Here again, the asynchronous algorithm performs better.

For the third example, we examined an algorithm whose characteristics changed when it was made asynchronous. For the asynchronous steepest descent algorithm (Figure 3), we found that the synchronous algorithm was faster for a small number of processors while the asynchronous algorithm was faster for a large number of processors.

Each of the algorithms showed a change in performance at 64 processors, which demonstrates that algorithms were affected by the structure of the problem. We did not optimize the algorithms for the particular problem that we solved because our purpose was to investigate the dependence of the performance of asynchronous algorithms on the characteristics of the synchronous algorithms.

The three asynchronous algorithms presented performed better than their synchronous counterparts in almost all cases; however, there are exceptions. For example, an asynchronous conjugate gradient algorithm can be derived in a manner similar to that demonstrated in Algorithm 3.2, but this algorithm is almost never competitive with the conjugate gradient algorithm. In fact, the rate of convergence is similar to that of asynchronous Jacobi. This

is because the iteration between groups is a Jacobi iteration, and if there is more than one group, the finite termination properties (in exact arithmetic) of conjugate gradient are lost, even within the groups. Thus, it is clear that the performance of an asynchronous algorithm depends not only on the communication delays of the synchronous algorithm, but also in the theoretical properties of the synchronous algorithm.

Finally, we note that future distributed memory machines will incorporate substantially faster communication hardware. Thus, a larger percentage of the cost of parallelization will be synchronization delays, and the advantages of using asynchronous algorithms will be even greater.

References

- [1] M. N. Anwar and M. N. El Tarazi. Asynchronous algorithms for Poisson's equation with nonlinear boundary conditions. *Computing*, 34:155–168, 1985.
- [2] R. H. Barlow and D. J. Evans. Parallel algorithms for the iterative solution to linear systems. *Computer Journal*, 25:56–60, 1982.
- [3] G. M. Baudet. Asynchronous iterative methods for multiprocessors. *Journal of the Association for Computing Machinery*, 25(2):226–244, 1978.
- [4] D. P. Bertsekas. Distributed asynchronous computation of fixed points. *Mathematical Programming*, 27:107–120, 1983.
- [5] D. Chazan and W. Miranker. Chaotic relaxation. *Linear Algebra and Its Applications*, 2:199–222, 1969.
- [6] M. Dubois and F. A. Briggs. Performance of synchronized iterative processes in multiprocessor systems. *IEEE Transactions on Software Engineering*, SE-8(4):419–431, 1982.
- [7] M. N. El Tarazi. Some convergence results for asynchronous algorithms. *Numerische Mathematik*, 39:325–340, 1982.

- [8] T. Kato, K. Wakimoto, K. Inagaki, and T. Sakai. Determinacy problem in relaxation method—analysis of asynchronous parallel processing. *Systems and Computers in Japan*, 17(6):85–94, 1986.
- [9] B. Lubachevshy and D. Mitra. A chaotic asynchronous algorithm for computing the fixed point of a nonnegative matrix of unit spectral radius. *Journal of the Association for Computing Machinery*, 33(1):130–150, 1986.
- [10] J. Miellou. Algorithmes de relaxation à retards. *Revue d'Automatique, Informatique et Recherche Opérationnelle*, 9, R-1:55–82, 1975.
- [11] J. Miellou. Asynchronous iterations and order intervals. In M. Cosnard et. al., editor, *Parallel Algorithms and Architectures*, Elsevier Science Publishers B. V., 1986.
- [12] D. Mitra. Asynchronous relaxations for the numerical solution of differential equations by parallel processors. *SIAM Journal for Scientific and Statistical Computing*, 8(1):s43–s58, 1987.
- [13] J. T. Robinson. Some analysis techniques for asynchronous multiprocessor algorithms. *IEEE Transactions on Software Engineering*, SE-5(1):24–31, 1979.
- [14] A. C. Shaw. Software descriptions with flow expressions. *IEEE Transaction of Software Engineering*, SE-4(3):242–254, 1978.
- [15] P. Spiteri. Parallel asynchronous algorithms for solving boundary value problems. In M. Cosnard et. al., editor, *Parallel Algorithms and Architectures*, Elsevier Science Publishers B. V., 1986.
- [16] P. Spiteri. Simulation d'exécutions parallèles pour la résolution d'inéquations variationnelles stationnaires. *EDF-Bulletin de la Direction des Etudes et des Recherches, Série C*, 149–158, 1983.

Internal Distribution

UC-32 (191)	
E. H. Barsis (5)	1400
P. J. Eicker	1410
J. J. Wiczer and Staff*	1411
S. J. Weissman and Staff*	1412
R. W. Harrigan and Staff*	1414
W. J. Camp	1420
R. J. Thompson and Staff*	1421
D. B. Holdridge	1421
R. C. Allen, Jr.	1422
D. E. Amos	1422
L. S. Baca	1422
P. B. Bailey	1422
L. A. Romero	1422
D. E. Womble (15)	1422
E. F. Brickell and Staff*	1423
G. G. Weigand	1424
R. E. Benner	1424
J. L. Gustafson	1424
G. R. Montry	1424
G. M. Pollock	1424
D. K. Gartling	1511
J. A. Schutt	1513
J. H. Biffle	1523
S. A. Landenberger (5)	3141
W. I. Klein (3)	3151
C. L. Ward (8)	3154-1
M. E. Thompson	6233
J. M. Harris and Staff*	8233
J. C. Meza	8233
R. A. Whiteside	8233
J. A. Wackerly	8524

* Route

External Distribution

Bob Hiromoto
MS B265
Los Alamos National Laboratories
Los Alamos, NM 87545

Wayne Joubert
Center for Num. Anal. RLM 13.150
Univ. of Texas
Austin, TX 77004

Linda Petzold L-316
Lawrence Livermore National Laboratories
P. O. Box 808
Livermore, CA 94550