NASA Contractor Report 177979

ICASE REPORT NO. 85-39

NASA-CR-177979
19860002389

ICASE

TOWARDS DEVELOPING ROBUST ALGORITHMS FOR SOLVING
PARTIAL DIFFERENTIAL EQUATIONS ON MIMD MACHINES

Joel H. Saltz

Vijay K. Naik

INSTITUTE FOR COMPUTER APPLICATIONS IN SCIENCE AND ENGINEERING
NASA Langley Research Center, Hampton, Virginia 23665

Operated by the Universities Space Research Association

NASA

National Aeronautics and
Space Administration

Langley Research Center
Hampton Virginia 23665

NF01008

1 Report No NASA CR-177979 ICASE Report No. 85-39	2 Government Accession No	3 Recipient's Catalog No		
4 Title and Subtitle TOWARDS DEVELOPING ROBUST ALGORITHMS FOR SOLVING PARTIAL DIFFERENTIAL EQUATIONS ON MIMD MACHINES		5 Report Date September 1985		
		6 Performing Organization Code		
7 Author(s) Joel H. Saltz and Vijay K. Naik		8 Performing Organization Report No 85-39		
		10 Work Unit No		
9 Performing Organization Name and Address Institute for Computer Applications in Science and Engineering Mail Stop 132C, NASA Langley Research Center Hampton, VA 23665		11 Contract or Grant No NAS1-17070		
		13 Type of Report and Period Covered		
12 Sponsoring Agency Name and Address National Aeronautics and Space Administration Washington, D.C. 20546		Contractor Report		
		14 Sponsoring Agency Code 505-31-83-01		
15 Supplementary Notes Submitted to Parallel Computing Langley Technical Monitor: J. C. South Jr. Final Report				

16 Abstract

Methods are proposed for efficient computation of numerical algorithms on a wide variety of MIMD machines. These techniques reorganize the data dependency patterns so that the processor utilization is improved.

The model problem examined finds the time-accurate solution to a parabolic partial differential equation discretized in space and implicitly marched forward in time. The algorithms investigated are extensions of Jacobi and SOR. The extensions consist of iterating over a window of several timesteps, allowing efficient overlap of computation with communication.

The methods suggested here increase the degree to which work can be performed while data are communicated between processors. The effect of the window size and of domain partitioning on the system performance is examined both analytically and experimentally by implementing the algorithm on a simulated multiprocessor system.

17 Key Words (Suggested by Author(s)) MIMD, multiprocessor, shared memory, iterative methods, performance of parallel processors.	18 Distribution Statement 59 - Mathematical & Computer Sciences Unclassified - Unlimited			
19 Security Classif (of this report) Unclassified	20 Security Classif (of this page) Unclassified	21 No of Pages 52	22 Price A04	

TOWARDS DEVELOPING ROBUST ALGORITHMS FOR SOLVING

PARTIAL DIFFERENTIAL EQUATIONS ON MIMD MACHINES

Joel H. Saltz
Institute for Computer Applications in Science and Engineering

Vijay K. Naik
Institute for Computer Applications in Science and Engineering

Abstract

Methods are proposed for efficient computation of numerical algorithms on a wide variety of MIMD machines. These techniques reorganize the data dependency patterns so that the processor utilization is improved.

The model problem examined finds the time-accurate solution to a parabolic partial differential equation discretized in space and implicitly marched forward in time. The algorithms investigated are extensions of Jacobi and SOR. The extensions consist of iterating over a window of several timesteps, allowing efficient overlap of computation with communication.

The methods suggested here increase the degree to which work can be performed while data are communicated between processors. The effect of the window size and of domain partitioning on the system performance is examined both analytically and experimentally by implementing the algorithm on a simulated multiprocessor system.

i

N86-11856 #

1. Introduction

It is widely recognized that parallel computation is essential to overcome the fundamental physical limits due to circuit switching and signal propagation delays on the computational speeds of sequential processing. Recent technological advances in the integration of electronic circuits has made it possible to conceive of large systems of processing elements working together in parallel on a single problem. In order to achieve high performance from such systems, the problem is divided into small tasks that can be solved in parallel. The flow of data between tasks must be matched to the characteristics of the interprocessor communication network so that tasks running on separate processors can cooperate in an efficient manner during the computation [HOCK81], [VOIG85].

In recent years most algorithms developed for parallel processing have been designed for implementation on vector computers or single instruction multiple data (SIMD) type systems, mainly because of the relatively wide availability of such machines [ORTE85]. Fewer algorithms have been developed specifically for implementation on multiple instruction multiple data (MIMD) type systems [VOIG85]. In such systems individual processors independently execute their own set of instructions and may operate asynchronously. In many MIMD machines that have been either built or proposed, each processor has a local memory to which it has relatively rapid access [HWAN85]. In this paper we discuss some algorithms and methodologies for efficient numerical computations on such systems. The processors may communicate by using a communication network or by passing messages through a shared memory.

The design of any parallel algorithm involves partitioning of the problem into individual tasks which can be executed concurrently on the processors of

a parallel processing system. If the problem can be subdivided among the processors in such a way that each processor can proceed at its own rate in solving its portion of the problem, without requiring information from other processors, the coordination is simplified. In the cases where this is not so, algorithms and machines must be matched so that the necessary data are made available where they are needed when they are required. For numerous algorithms, the total time required to move data to the appropriate processing element is larger than or equal to the time required to perform the computation [GENT78], [ORTE85], [SAAD85].

Complications may arise when processors require data computed in other processors to continue performing useful work on a problem. A number of factors can conspire to create a situation in which data are not available where and when the data are needed. Two such factors are: 1) interprocessor communication delays may prevent a processor from receiving a piece of information available elsewhere, when required; 2) the computational abilities of the processors performing work on the problem and the portions of work assigned may not be matched. In this paper we concern ourselves with the first problem. The question of load balancing is addressed in [SALT85].

The degree and properties of interprocessor communication delays are dependent on the characteristics of the computer architecture on which the problem is run, the algorithm used to solve the problem concurrently, and the mapping of the problem onto the architecture. If the scheme employed for solving the problem in parallel has the property that results are needed for use in other processors soon after being computed, the scheme will be sensitive to delays in the transmission of data between the processors, and the utilization of the system as a whole is apt to be degraded. Here we

define the utilization of the system as the average amount of time spent by the processors in performing useful operations divided by the total time required to solve the problem.

In this paper we will demonstrate that it is possible to reorganize computations in each processor so as to increase the amount of useful work that can be performed on the problem while each processor waits for data from other processors. This study will be based on a model problem of obtaining the time-accurate solution to a linear parabolic partial differential equation discretized in space and implicitly marched forward in time. The algorithms investigated are iterative methods that are extensions of block Jacobi and SOR.

In the next section we discuss the general principles involved in parallel implementations of basic iterative methods such as Jacobi, Gauss-Seidel, and SOR methods. In Section 3 the model problem is introduced. After showing the difficulties due to the communication delays encountered in applying the parallel versions of basic iterative methods, we present two algorithms which are extensions of Block Jacobi and Block SOR and which allow efficient overlap of computation with communication. Schemes for implementing these algorithms on multi-processor systems are discussed in Section 4. In Section 5 we derive bounds on processor utilizations for such implementations. These algorithms are implemented on a simulated shared memory machine and the results are given in Section 6. Experimental results on convergence and overhead of the algorithms developed in Section 3 are presented in Section 7, and it is shown that for the cases considered, the overhead is moderate compared to the gain obtained in employing these algorithms. Finally, concluding remarks are presented in Section 8.

2. Parallel Iterative Methods

In this section the parallel implementation of basic iterative methods will be outlined. For specificity, we consider the standard uniform five point difference approximation to Laplace's equation given by

$$u_{xx} + u_{yy} = 0$$

in a square domain subject to Dirichlet boundary conditions. The difference equation obtained when the mesh is n by n is

$$u_{i,j} = \frac{1}{4} \left(u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} \right).$$

The subscripts i and j range from 1 to n; and $u_{i,n+1}$, $u_{i,0}$, $u_{n+1,j}$, and $u_{0,j}$ are given constants. Notice that the equation for each mesh point involves data at that point as well as the point's north, south, east, and west neighbors.

The Jacobi method for the solution of this system of difference equations may be written as

$$u_{i,j}^{k+1} = \frac{1}{4} \left(u_{i+1,j}^{k} + u_{i-1,j}^{k} + u_{i,j+1}^{k} + u_{i,j-1}^{k} \right)$$

where the superscript denotes the iteration number. The Jacobi method often converges very slowly, but is considered to be a prototype parallel method [ORTE85]. On multiprocessor computers, single mesh-points or groups of mesh points are assigned to each processor. All points are updated each iteration, and since only the values from the last iteration are utilized, each step of the iteration may proceed to completion without any need for communication of data values during the step.

The Gauss–Seidel and SOR methods require new values at each point to replace the old values as soon as the new values are computed. In sequential computers the points in the domain are generally processed one iteration at a time, point by point and line by line. Because of the need for the new values of the variables at the previously computed points, only one mesh point can be dealt with at a time; hence, the process is not suited to multiprocessor machines. To deal with this problem the mesh points are grouped into two subsets such that no points in a subset are coupled to other points in the same subset [ERIC72], [HAYE74], [LAMB75], [ADAM82]. The most common way of obtaining these sets is to use the classical red/black ordering [YOUN71]. For the model domain this is obtained by assigning points to the black subset when $i+j$ is odd, and to the red subset when $i+j$ is even. Each iteration can then proceed in two separate phases where each phase has the properties of a Jacobi sweep in that all mesh-points in a given phase may be adjusted independently. Other iterative methods such as the chaotic or asynchronous relaxation involve the updating of mesh points with variable values that need not come from the previous iteration [CHAZ69], [BAUD78], [DEMI82]. In the simplest form of chaotic relaxation, each processor uses whatever values are available to compute the next value of the iterate at a given point, without any restriction on the iteration number of the variable value chosen.

It is possible to implement Jacobi, Gauss Seidel, and SOR iterative methods so that all new values corresponding to a group of variables are determined at once. Such schemes are referred to as block iterative methods. The simultaneous determination of the values of groups of variables involves the solution of subsystems of equations, generally by direct methods. Under specific conditions the block iterative methods are known to converge at a

rate faster than the corresponding point iterative methods [YOUN71], [VARG62].
Various solutions are proposed to introduce parallelism in the block SOR
methods, including the scheme of partitioning the domain into red and black
blocks [ERIC72], [LAMB75], [PART80], [PART82], [FABE81].

3. Multistep Iterative Methods

In many algorithms for the solution of time dependent problems, it is
necessary to solve a sequence of linear systems of equations involving the
same matrix but different right hand sides. In these cases, the right hand
sides of the consecutive systems of equations are dependent on the solutions
of the equations earlier in the sequence. The implicit solution of the
parabolic partial differential equation using the Crank-Nicholson method in
time gives rise to one such algorithm. Typically when an iterative method is
applied to a time dependent problem, iterations proceed successively over the
systems of equations for each timestep [HAGE81]. Since the consecutive
systems of equations depend on the solutions of the earlier equations, such
implementations limit the extent of possible parallelism. In this section we
present algorithms which exploit a new level of parallelism in solving time
dependent problems. We achieve this by traversing more than one timestep
during the course of a single iteration. These schemes allow coarse grained
multiprocessor implementations, which are minimally dependent on machine
characteristics and have favorable, well-defined convergence properties.

In the following, the multistep parallel iterative methods will first be
discussed in a relatively synchronous context so that the nature of the
numerical algorithm and the most straightforward implementations can be

understood. This will be followed by discussions of ways in which the algorithms described can be implemented so as to allow for less synchronized implementations and high processor utilizations.

3.1 Block Jacobi Iteration

We consider the system

$$My = b$$

where M is an n by n matrix and y and b are vectors of length n. We assume for the remainder of the section that the above system is partitioned in the form

$$
M = \begin{bmatrix}
M_{1,1} & M_{1,2} & \cdot & \cdot & M_{1,q} \\
M_{2,1} & M_{2,2} & \cdot & \cdot & M_{2,q} \\
\cdots & \cdots & \cdots & \cdots & \cdots \\
M_{q,1} & M_{q,2} & \cdot & \cdot & M_{q,q}
\end{bmatrix}
\quad
y = \begin{bmatrix} y_1 \\ y_2 \\ \cdot \\ \vdots \\ y_q \end{bmatrix}
\quad
b = \begin{bmatrix} b_1 \\ b_2 \\ \cdot \\ \cdot \\ b_q \end{bmatrix} ,
$$

where the M_{ij}'s are submatrices and y_i and b_i represent subvectors.

The Block Jacobi method is defined as follows,

$$
M_{i,1} \, y_1^{v+1} = - \sum_{j \neq 1} M_{1,j} \, y_j^{v} + b_i
$$

where v represents iteration number. All of the updated values at the v+1st iteration are obtained using values from the vth iteration. The important special case when $M_{1,j}$ are 1 by 1 submatrices gives rise to the point Jacobi method.

3.2 Multistep Block Jacobi Algorithm

Now consider the following set of systems of equations.

$$My(t_r) = Py(t_{r-1}) + b(t_r), \qquad r = 1, \cdots, m \qquad (1)$$

where M and P are n × n matrices, $y(t_r)$ and $b(t_r)$ are vectors of n variables and t_r is the rth timestep. Shortly we will show how these equations arise in the solution of time dependent partial differential equations as well as the integration of first-order ordinary differential equations.

The solution of these equations with standard block or point iterative methods involves the consecutive solution of each of the m systems of linear equations corresponding to the m timesteps. After iterating over each timestep until satisfactory convergence is obtained, one moves on to iterate over the next timestep. On a sequential machine, this is, in fact, the usual way an iterative method is applied to a time dependent problem [HAGE81]. In Figure 1 this algorithm is depicted schematically and a parallel version is outlined below in an algorithmic form. Here ´q´ is the number of partitions into which the domain is subdivided.

Jacobi Method Iterated Until Convergence over Each Timestep

For r=1 to number_of_timesteps
{

   Pardo i = 1 to q
   {
$$y_i^1(t_r) = y_i(t_{r-1})$$
   }

   Beginning with v = 0 **increment** v **until** $\|y^{v+1}(t_r) - y^v(t_r)\| < \varepsilon$
   {

      Pardo i = 1 to q
      {
         Solve:
$$M_{i,i}\, y_i^{v+1}(t_r) = -\sum_{j \ne i} M_{i,j}\, y_j^v(t_r) + \sum_{j=1}^{q} P_{i,j}\, y_j(t_{r-1}) + b_i(t_r)$$
      }

   }

   Pardo i = 1 to q
   {
$$y_i(t_r) = y_i^{v+1}(t_r)$$
   }

}

In the above algorithm as well as in the subsequent algorithms depicted below, we assume that the matrix P is decomposed into blocks using a partitioning identical to that already used for M. The submatrix P_{ij} hence includes the same rows and columns of P as M_{ij} does of M. Each vector $b(t_r)$ is decomposed into subvectors $b_i(t_r)$.

Instead of iterating until convergence over each timestep before moving on to the next, it is possible to iterate over a number of timesteps at once. The set of timesteps over which these extended iterations occur is called a window. After a fixed number of iterations have taken place over the window, or after convergence has been obtained at the first timestep, the window moves up one timestep. When the window shifts upwards, the value of

the solution at the timestep that was at the top of the old window is treated as the initial approximation at the new timestep which is now at the top of the shifted window. At the beginning of the iterations, the initial value specified for the problem is used as the initial guess at all timesteps in the initial window. This multistep generalization of Block Jacobi iteration is called Windowed Block Jacobi algorithm (WBJ). This is schematically depicted in Figure 2.

In WBJ an iteration takes place when possible over a window of w timesteps. The term microstep describes the relative position of a timestep with respect to the beginning of a window. A timestep that is i timesteps above the beginning of a window will be designated as the ith microstep of the window. The number of microsteps advanced from the beginning of iterations performed over a window is designated as the number of cumulative microsteps (cms). In Figure 3 the process of measuring cumulative microsteps is illustrated by means of an example. The concept of cumulative microstep is an essential tool for the development and evaluation of techniques described in this paper. Note in the following algorithm that iterations are never performed on the timesteps beyond ´number_of_timesteps´ specified for the problem. Consequently, the window size during the solution of the last few timesteps may correspondingly be reduced. The following is an explicit outline of WBJ. As before q is the number of partitions, and w is the maximum number of timesteps in a window. ´Window´ is the actual number of timesteps used in a window.

WBJ - Windowed Block Jacobi Method

Pardo $i = 1$ to q
{
    Pardo $s=1$ to window
    {
$$y_i^1(t_s) = y_i(t_0)$$
    }
}
For $r=1$ to number_of_timesteps
{
    Beginning with $v=0$ **increment** v until $\|y^{v+1}(t_r) - y^v(t_r)\| < \varepsilon$
    {
        Pardo $i = 1$ to q
        {
            window = min(w, number_of_timesteps $- r + 1$)
            for $s = r$ to $r +$ window $- 1$
            {
                Solve:
$$M_{i,i}\, y_i^{v+1}(t_s) = -\sum_{j \neq i} M_{i,j}\, y_j^v(t_s) +$$
$$P_{i,i}\, y_i^{v+1}(t_{s-1}) + \sum_{j \neq i} P_{i,j}\, y_j^v(t_{s-1}) + b_i(t_s)$$
            }
        }
    }
    Pardo $i = 1$ to q
    {
$$y_i^1(t_{r+window}) = y_i^{v+1}(t_{r+window-1})$$
$$y_i(t_r) = y_i^{v+1}(t_r)$$
    }
}

In both the algorithms discussed above, during iteration $v+1$ each partition i utilizes variables y_j^v, $j \neq i$ from other partitions. This procedure allows simultaneous computation of all partitions on a multiprocessor system. It should be noted that there may be several blocks assigned to each processor.

The pardo statements above simply serve to denote the existence of a number of subtasks that can run independently.

In [SALT85] it is shown that the spectral radius of the iteration matrix of WBJ for any window w is identical to that of the corresponding block Jacobi method; hence, the asymptotic convergence characteristics of the algorithms are comparable. In the Jacobi method, it is necessary to calculate $\sum_{j=1}^{q} P_{i,j} \, y_j(t_{s-1})$ only once for each timestep t_s over which the solution is integrated. In WBJ this matrix vector multiplication must be performed only once when a timestep is at the bottom of a window, but it must be performed each time a timestep is above the lowest part of the window. Therefore one expects that the overhead involved in the new methods will be moderate when large blocks are utilized but quite high when the matrix is decomposed into very small blocks or points. In the next section, however, it will be shown that a minor reorganization of the computations can considerably reduce the overhead.

Note that instead of utilizing solution values at earlier timesteps as initial approximations at the top of a new window, one may utilize explicit integration methods. In numerical experiments this was shown to aid convergence but in some cases to adversely affect stability.

We now explicitly show how WBJ may be used to solve the set of ordinary differential equations obtained from the spatial discretization of a parabolic partial differential equation. Note, of course, that this method would apply to a system of differential equations obtained in other ways as well. This system may be represented as

$$\dot{y} = Qy + c(t)$$

where y, $\dot{y}$ and $c(t)$ are vector valued functions and Q is a matrix of constants. Assume that $y(t_1)$ is given as an initial condition.

Applying the Crank Nicholson integration method to discretize in time we obtain,

$$\left(y(t_r) - y(t_{r-1})\right)/\Delta t = (Q/2)\left(y(t_r) + y(t_{r-1})\right) + c(t_r)/2 + c(t_{r-1})/2.$$

To advance from t_0 to t_m we solve Eqns. (1) above with the following definitions of M, P, and $b(t_r)$:

$$M = (I/\Delta t - Q/2), \qquad P = (I/\Delta t + Q/2)$$

and

$$b(t_r) = c(t_r)/2 + c(t_{r-1})/2, \qquad r = 1, \cdots, m.$$

Let $Q_{i,j}$ represent the partition of Q that corresponds to the same rows and columns as does the partition $M_{i,j}$ of M and $P_{i,j}$ of P. Recall from the earlier discussion that the equations for y_i^{v+1} at each timestep t_s within the window are given by,

$$M_{i,i} \, y_i^{v+1}(t_s) = -\sum_{j \neq i} M_{i,j} \, y_j^{v}(t_s) + \sum_{j \neq i} P_{i,j} \, y_j^{v}(t_{s-1}) + P_{i,i} \, y_i^{v+1}(t_{s-1}) + b_i(t_s).$$

$$(2)$$

This may be rewritten in the case of this specific problem as follows:

$$\left(I/\Delta t - Q_{1,1}/2\right)y_1^{v+1}(t_s) = \left(I/\Delta t + Q_{i,1}/2\right)y_1^{v+1}(t_{s-1})$$

$$+ \sum_{j \neq 1} \left(\left(Q_{1,j}/2\right)\left(y_j^v(t_s) + y_j^v(t_{s-1})\right)\right) + b_1(t_s). \qquad (3)$$

To realize the advantages of this rearrangement, consider the case that would appear to be the most problematical when (2) is applied, that is, when the block size is one, which corresponds to a point iterative method. Consider the computational overhead arising from iterations over timesteps t_s that are above the lowest point in a window. By applying (2) an extra matrix vector multiplication appears to be required per iteration over each of these timesteps. In applying (3), $y_j^v(t_s) + y_j^v(t_{s-1})$ needs to be computed only once per iteration. Hence, assuming that $(I/\Delta t + Q_{1,1}/2)$ has been precomputed, only one additional multiplication and two additions need be performed per variable for each iteration over t_s. When a number of processors require the value $y_j^v(t_s) + y_j^v(t_{s-1})$, one would expect it to be computed separately in each processor.

3.3 Windowed-Block Successive Overrelaxation Algorithm

One may refine WBJ with the aim of improving convergence by employing a multistep version of Block Successive Overrelaxation (BSOR). Partitioning is performed as described in the previous sections and, as before, each partition is advanced w timesteps during each iteration of the algorithm. In this case, however, the blocks are solved consecutively and estimated values of variables calculated from other partitions are used as soon as they become available. The other refinement involves the use of overrelaxation. A weighted average

of the old and the new variable values for each partition is employed; the old variable values are assigned a negative weight. This multistep generalization of BSOR is called Windowed-Block Successive Overrelaxation (WBSOR). Following is a formal presentation of WBSOR. Meanings of q, w, and window are as before.

WBSOR - Windowed-Block Successive Overrelaxation

Pardo i = 1 to q
{

    Pardo s=1 to window
    {

$$y_i^1(t_s) = y_i(t_0)$$

    }
}
for r=1 to number_of_timesteps
{

    Beginning with v=0 **increment** v **until** $\|y^{v+1}(t_r) - y^v(t_r)\| < \epsilon$
    {

        for i = 1 to q **do** in parallel when possible
        {

            window = min(w, number_of_timesteps - r + 1)
            for s = r to r + window - 1
        {

        Solve:

$$M_{i,i}\, \tilde{y}_i^{v+1}(t_s) = - \sum_{j<i} M_{i,j}\, y_j^{v+1}(t_s) + \sum_{j<i} P_{i,j}\, y_j^{v+1}(t_{s-1})$$

$$- \sum_{j>i} M_{i,j}\, y_j^v(t_s) + \sum_{j>i} P_{i,j}\, y_j^v(t_{s-1}) + b_i(t_s)$$

$$y_i^{v+1}(t_s) = \omega\tilde{y}_i^{v+1}(t_s) + (1-\omega)y_i^v(t_s)$$

        }
        }
    }
Pardo i = 1 to q
    {

$$y_i^1(t_{r+window}) = y_i^{v+1}(t_{r+window-1})$$
$$y_i(t_r) = y_i^{v+1}(t_r)$$

    }
}

If the blocks or partitions are chosen properly, it is possible to order them so that one obtains two sets of blocks such that all blocks in a given set are uncoupled from one another. By convention the blocks in one set are designated as black blocks, and those in the other are designated as red blocks. For example, in the case of a two-dimensional spatial domain the grid of points on which the problem is to be solved may be partitioned into strips. The partitioning should be done in such a way that each strip is coupled at most to the two adjacent strips. The strips are assigned to two sets such that adjacent strips are in separate sets.

In [SALT85] the spectral radius of the iteration matrix of WBSOR is shown to be independent of the window size used. It should be noted that WBSOR with window size one is simply BSOR. The use of WBSOR with any window size yields convergence that is asymptotically identical to that obtained with BSOR.

4. The Implementation of WBJ and WBSOR on Multiprocessor Systems

In this section we consider the implementation of WBJ and WBSOR on multi-processor machines. Here we assume that each processor of the system has a substantial local memory. Information in this local memory may be accessed more easily and inexpensively than information in the local memory of another processor or in any global memory that might exist. The submatrices required to carry out the computations of WBJ and WBSOR on particular portions of the domain of the partial differential equation being solved should be quickly obtainable from memory. Responsibility for work involving particular blocks or portions of the domain is consequently assigned to a specific processor. To reduce interprocessor communication requirements, blocks which share coupling variables should be assigned to the same processor when possible.

We first consider a straightforward multiprocessor implementation of WBJ. A collection of blocks is assigned to each processor. Each processor is programmed so that it may advance all blocks when the variable values required have been obtained from other processors. Upon computing new approximations to the partial differential equations at a given timestep, the processor sends values of the computed variables to other processors that require the information. The operations performed by the processors and the patterns of interprocessor communication when the blocks are of size one by one (i.e., points), are similar to those described in [REED84].

In WBJ, as presented above, synchronization between all processors must occur whenever the window shifts upwards. This requirement exists in order to ensure that convergence has occurred at the lowest timestep in the window before work on that time timestep comes to a halt. In order to advance to the nth iteration mth microstep where $n > 1$, each processor requires coupling variable values from the n-1st iteration, mth microstep. It is clear that all processors need not be performing work on the same iteration and the same microstep simultaneously. Thus, as long as the advancement of individual processes is based on the availability of appropriate coupling variable values, no global synchronization is necessary. Furthermore, as the window size of WBJ increases, the degree to which the processors are constrained by this implicit synchronization requirement decreases. The flow of data in a two processor system with a window of three is illustrated in Figure 4.

We now consider a multiprocessor implementation of WBSOR. Henceforth, it will be assumed that the points or blocks involved in this algorithm are ordered using the red-black ordering. As explained before in WBSOR, as in WBJ, synchronization is performed among all processors whenever the window

shifts upwards. Blocks may be assigned to processors in one of a number of ways. The nature of the synchronization that must occur is dependent on the way in which blocks of different colors are assigned to processors. Consider first the case in which processors contain either only red blocks or only black blocks. In order to advance to the nth iteration, mth microstep where $n > 1$, each processor containing black blocks requires coupling variable values from processors containing red blocks from the n-1st iteration, mth microstep. Each processor containing red blocks requires coupling variable values from processors containing black blocks from the same iteration and the same microstep.

With a window of size one, i.e., the standard red-black BSOR scheme, processors can only be active half of the time given an assignment of only one color block in each processor. This difficulty disappears when a window size of two or more is utilized, and with larger windows the tightness of the coupling between processors continues to decrease. The assignment of multiple red blocks to some processors and of multiple black blocks to others has a potentially serious side effect. Since black blocks are coupled only to red blocks and vice-versa, values of all of the coupling variables in every block must be obtained from other processors. In the point iterative version of WBSOR, this arrangement would require the communication of the value of every variable for each microstep computed at each iteration over every window.

One may assign both red and black blocks to each processor and may thus be able to substantially reduce the interprocessor communication requirements. Consider the case in which one wishes to solve a time dependent partial differential equation differenced with a five point template. The domain of the partial differential equation is divided into regions and the variables in

each region are assigned to blocks. All blocks in each region are assigned to a particular processor. With this arrangement only the values of the variables corresponding to the mesh points at the boundaries of the regions need to be communicated regardless of the size of the blocks.

5. Analysis of Communication Delay Effects

This section demonstrates the usefulness of windowing techniques in ameliorating the effects of communication delays. The delay in sending a message from one processor to another may be written in the form $\alpha + \beta *$ size, where α and β are some parameters, and ´size´ is equal to the number of bytes in the message. The nature of communication delays depends both on the multiprocessor architecture and the demands on the communication network made by the algorithm being run on the multiprocessor. In this paper, interprocessor communication delays will be modelled in the following two ways:

(1) Uniform interprocessor communication delays that vary only with the size of each message, i.e., α and β are constants, and

(2) detailed simulation of a specific family of multiprocessor architectures. The detailed simulations directly model any queuing effects that occur; hence, communication delay increases as a function of the number of messages sent.

An upper bound of utilization for WBJ and WBSOR is derived below. The upper bound calculations assume uniform interprocessor delays as described above. It will be shown that this upper bound decreases with interprocessor communication delay but increases with window size.

First a simple lower bound on the time required for each PE to advance all of its blocks m cumulative microsteps will be derived. This bound is a linear function of the interprocessor communication lag and is inversely proportional to the window size. From this lower bound, an upper bound on processor utilization is obtained. It is assumed that communication lags are constant and unrelated to the amount of information sent.

Fix attention on a given PE P, in a multiprocessor system. We consider the execution of WBJ and WBSOR with window size w, on a multiprocessor system. Assume that to advance all the blocks in a given processor P one cumulative microstep, a total time T would be required. Assume further that communication with other processors requires time cT. The variable c will be called the communication delay. Consider a boundary block B in P which requires coupling variable data from some other processor or processors.

Proposition 1: The boundary block B requires at least $2cT$ time to advance from the end of cms i to cms i+kw, where k = 2 for WBJ and k= 1 for WBSOR.

Proof: The boundary block B at cms i is, by definition, coupled to at least one block B´ in another processor. Consider first the case of algorithm WBJ. In order for B´ to advance to cms i+w it requires variable values from B at cms i. Time cT is required for this information to get to B´. After B´ has computed its results for cms i+w it must send those results to B. This takes time cT. B cannot advance to cms i+2w until the results from B´ corresponding to cms i+w arrive. Thus a lower bound on the time required for B to advance from cms i to cms i+2w is 2cT. A similar proof in the case of

WBSOR demonstrates that block B requires at least 2cT to advance from cms i to cms i+w.

The above proposition provides a lower bound on the time T_{tot} that processor P takes to complete its work. Since P must advance all of its blocks m cumulative microsteps, it cannot complete its work in less than time $2cT \left\lfloor \frac{m}{kw} \right\rfloor$. Now mT is the computation time required to advance all blocks of P, m cumulative microsteps. The total time to complete the problem is hence bounded below as follows

$$T_{tot} \geq \max\left(mT, \ 2cT \left\lfloor \frac{m}{kw} \right\rfloor\right).$$

The utilization U_p of the processor P defined as $\dfrac{\text{computation time}}{T_{tot}}$, is accordingly subject to the following upper bound

$$U_p \leq \frac{1}{\max\left(1, \ \dfrac{2c}{m} \left\lfloor \frac{m}{kw} \right\rfloor\right)}.$$

A very simple asymptotic form will be derived which bounds the utilization achievable, by a function dependent only on the window size w and the communication delay c. Note that,

$$\max\left[mT, \ 2cT \left\lfloor \frac{m}{kw} \right\rfloor\right] \geq \max\left[mT, \ 2cT \left(\frac{m}{kw} - 1\right)\right].$$

Hence

$$U_p \leq \frac{1}{\max\left(1, \ \dfrac{2c}{kw} - \dfrac{2c}{m}\right)}.$$

In the limit of large m,

$$U_p \leq \frac{1}{\max\left(1, \ \dfrac{2c}{kw}\right)},$$

and if $c < \frac{kw}{2}$ then $U_p < 1$. Thus, the communication delay that can be tolerated before a bound on utilization occurs increases linearly with w. If $c > \frac{kw}{2}$, then $U_p < \frac{kw}{2c}$. This means that as the window size increases, the sensitivity of the processor to communication delays is reduced. Note that for WBJ, $k = 2$ and for WBSOR, $k = 1$ and hence for any window size, WBSOR is more sensitive to communication delays than WBJ.

The upper bounds derived above do not depend on the amount of time required to advance any of the blocks in the system. A refinement of these upper bounds may be obtained by taking into account the time required to advance boundary blocks. Refined upper bounds will be computed for WBSOR below; the same principles could be utilized to refine the upper bounds for WBJ.

Assume that all boundary blocks in all PEs require computation time T_{comp} for advancement.

Proposition 2: The boundary block B requires at least $2cT + 2T_{comp}$ time to advance from the end of cms i to cms i+w.

The above proposition is proved exactly as was proposition (1) above, except that: (1) block B´ must now compute for time T_{comp} before sending variable values to B and (2) after receiving data from B´, B must compute for time T_{comp} before it can make variable values available at cms i+w.

Reasoning exactly as before and substituting $k = 1$,

$$T_{tot} \geq \max\left[mT, \left(2cT + 2T_{comp}\right)\left\lfloor \frac{m}{w} \right\rfloor\right]$$

and hence the utilization bound is

$$U_p \leq \cfrac{1}{\max\left(1, \left(\cfrac{2c}{m} + \cfrac{2T_{comp}}{mT}\right)\left\lfloor\cfrac{m}{w}\right\rfloor\right)} .$$

For large m, the asymptotic utilization bound is

$$U_p \leq \cfrac{1}{\max\left[1, 2/w\left(c + \cfrac{T_{comp}}{T}\right)\right]} . \qquad (4)$$

6. Simulation Results

Detailed simulations of a particular architecture were performed, and the algorithms developed here were implemented on this simulated machine to examine the effects of windowing on the system performance. Results obtained from these simulations for the algorithm WBSOR pertaining to the effect of communication delay and window size on processor utilization are outlined in this section. In all sets of the simulations we assume that blocks consist of strips of the domain (Figure 5) and that only the neighboring strips have coupled variables. All processors have the same number of blocks assigned to them, and the blocks assigned to a given processor are physically adjacent to one another.

6.1 A Simulated Shared Memory Machine

A low level simulator called SIMON was employed to simulate a shared memory architecture. SIMON is an event-driven multi-processor simulator consisting of time-sorted queues [FUJI83], [HELL84]. It is capable of

providing nano-second precision and allows the user to control the timing at the instruction level. Utility functions are provided to define multi-processor architecture, send and receive messages, etc. When a message is sent between processes, its arrival time is determined from the send time and a delay representing travel through the interprocessor connections and switches. It allows the user to control the computation costs at each processor as well as the costs involved in interprocessor communication.

The multi-processor system simulated here consists of a number of processing elements and a global shared memory. Each processing element (PE) contains a central processing unit and a substantial local memory. The instructions and data corresponding to the tasks assigned to a processor reside in its local memory and the processor alone has direct access to this memory. The global shared memory is made up of a number of modules and these are accessible to all the processors with equal priorities. The processors are connected to the modules through a crossbar switch. The processors communicate with each other by reading and writing data in the shared memory modules. The access to these modules is brought about by means of input/output handlers (i/o-handler) and an input/output processor (i/o-processor). Attached to each PE is an i/o-handler which takes care of the read/write operations associated with the shared memory and allows its host PE to continue performing computations. The shared memory and the crossbar of the system are controlled by the i/o-processor. Only one i/o-handler is allowed to read or write to a particular memory module at a given time, although a number of i/o-handlers can read or write to distinct memory modules. The i/o-processor arbitrates the access to the shared memory modules by different i/o-handlers through the crossbar switch.

The communication that occurs in this model machine involves either: 1) messages concerning requests and permissions to read or write, or 2) messages that include variable values that must be transmitted and received. The requests and permission messages are small, consisting of only a few bytes. The transmission of variable values requires messages that are generally much longer, and their size depends on the number of variables that are shared.

The operation of the modeled shared memory machine is carried out as follows. When a processor needs to read or write data to the shared memory, it sends a message to its i/o-handler. This message designates whether a read or a write is requested and also designates the memory module required. When the i/o-handler receives this message, it forwards the request to the i/o-processor. The i/o-processor collects and queues requests to read and write sent by all of the i/o-handlers. When a request to read or write is serviced by the i/o-processor, an i/o-handler is given exclusive permission to interact with a specific memory module. Depending on the request involved, the i/o-handler can either write from the PEs´ local memory to a particular memory module, or read from a particular memory module and write to its PEs local memory.

6.2 Effect of Windowing

The simulated shared memory system, described above, was employed to examine the performance of WBSOR as the window size was varied. The simulations were carried out for a system consisting of eight processors and eight memory modules and also for a system with eight processors and one memory module. Henceforth the former system is referred to as Machine A and the latter as Machine B.

In the simulation runs described below, it is assumed that a model domain decomposition is applied where a uniform grid is divided into strips. Each strip has 1000 mesh points on the boundary. The time required to send each message, sent over a given link in the shared system, is assumed to be 1 microsecond plus 0.025 times the number of bytes in the message. This corresponds to a bandwidth of 40 Mbytes per second per 32 bit wide communication channel. As mentioned earlier, the simulations explicitly take into account the communication requirements of the problem, including the queuing effects on the communication delays and the changes in the data requirements that occur when a window shifts upwards at the beginning of a new timestep. Thus, although the bandwidth of the channels is fixed, the problem parameters affect the communication delays in the system as a whole, and they are accounted for in these simulations. In these experiments it is assumed that the block advancement times for all blocks are identical in all the processors. A block advancement time is defined as the time it takes to perform computations for a single timestep during an iteration over the window, once the data for that timestep are available. Results are presented here for the block advancement times of 0.5 millisecond, 1 millisecond, and 5 milliseconds. Finally, in these simulation experiments, it is assumed that an equal number of iterations are required over each timestep.

The variation of processor utilization as a function of window size when one and two blocks are assigned to each processor of Machine A is depicted in Figure 6 and Figure 7, respectively. Figure 8 and Figure 9 show the same for Machine B. In the case of Machine A, the utilization increases as the window size is increased, when either one or two blocks are assigned to a PE. This is true for all the block advancement times considered. These improvements in

processor utilization with window size taper off for larger window sizes. As the block advancement time is increased, the relative effect of communication delays decreases, and the processor utilization improves.

In Figure 6, where only one block is assigned to a PE, the utilization for window size one does not increase above 0.5, regardless of the time needed to advance a block. This is so because here each PE has either a black or a red block. Ignoring the predictions that occur at the beginning of each timestep, black blocks require variable values from red blocks from the last cumulative microstep, and red blocks require variable values from black blocks from the same cumulative microstep. Therefore, a black block and its neighboring red blocks cannot advance simultaneously when the window size is one; thus, the processor must remain idle approximately half of the time even if the inter-processor communication were instantaneous. This restriction disappears when the window size is greater than one or when more than one block is assigned to a PE. When each PE is assigned one block and when the block advancement time is relatively small, a window size greater than one helps to some extent, and the utilization goes up, but the queuing effects soon catch up with the gain from higher window size. From Figure 7 it can be seen that, if the number of blocks assigned to each PE is increased from one to two, the effect of queuing delays is decreased and much higher utilizations are observed.

The effect of the underlying hardware, specifically that of the number of memory modules in the shared memory, on the performance of WBSOR is observed when the number of modules is reduced to one (Figure 8 and Figure 9). As before the utilizations increase with block advancement time, but much more gradually. The effect of change in the window size is not felt until the block advancement time is large enough to include all the queuing delays at

the module which acts as a bottleneck. The block advancement times, above this threshold, show trends similar to those observed in the case of Machine A. Altering the window size affects the patterns of interprocessor communication, but does not change the amount of information that must eventually be communicated before the problem is completed and hence, under some circumstances, with a smaller number of memory modules, one may expect lower performance improvements through the use of increasing window size.*

The average communication delay between each pair of processors was measured from the simulations for machines A and B when two blocks were assigned to each PE. The maximum of the average interprocessor communication delays for Machines A and B is shown in Figure 10. The block advancement time is assumed to be 1 millisecond. Note that the interprocessor communication time increases quite gradually with window size, when eight modules are assigned, but increases almost linearly with the window size when the machine consists of only one module. An approximate value for the upper bound of processor utilization can be obtained by substituting in (4) the maximum average time needed for the interprocessor communication that must take place between pairs of processors. These upper bounds for utilization, given by (4), are compared with those observed in the simulation experiments for Machines A and B in Figure 11. The upper bound calculated using the maximum average interprocessor communication delay in (4) approximates rather closely the results obtained from the simulations. Thus, the usefulness of windows in mitigating the effects of communication delays is demonstrated in a realistic simulated machine.

---

* A detailed analysis of the influence of hardware parameters on the algorithm performance will be published separately.

7.   Experimental Results on Convergence and Computational Overhead

Even though the spectral radii of the iteration matrices do not vary with window size, the computation time required to complete a problem may increase with window size for the following two reasons: 1) The number of iterations required to bring two successive approximations to within tolerance in a specific norm, in this case the maximum norm, may have a dependence on window size; 2) the computational work required per iteration is expected to increase with window size to a minor degree. Here the experimental results on the effect of window size on the number of iterations required and on the total computation time taken are presented. It will be seen that the cost increase is quite modest and is often outweighed by the increase in the processor utilization attributable to the application of windows.

The results on overhead attributable to the use of windows were found to be similar for both WBJ and WBSOR, and hence the results pertaining to WBSOR are presented. The heat equation was solved using a 50 by 50 point mesh and a timestep of 0.001. The initial condition consisted of the first two modes of the equation. The equation was solved subject to Dirichlet boundary conditions. Iterations were continued until the maximum of the difference between two succeeding approximations at the first microstep in the current window was less than the given tolerance of 1E-5.

The domain was decomposed into blocks of different sizes in different experiments. The domain was divided into 5 strips that were each 50 by 10 points, 10 strips that were each 50 by 5 points, and 25 strips that were each 50 by 2 points. Square blocks that were each 10 points by 10 points were also considered. The equation was advanced 50 timesteps and the average number of iterations required to achieve the prescribed tolerance of 1E-5 was calculated for each of the types of blocks.

For window sizes 1, 2, 3, and 4 the average number of iterations required to reach tolerance 1E-5 is displayed in Figure 12 and Figure 13, when 50 × 2 and 10 × 10 blocks are used respectively. It is clear that the average number of iterations required increases rather gently with window size. The number of iterations required by the first few timesteps of the problems investigated here grows relatively quickly with window size (Figure 12 and Figure 13). This effect is due to the cost involved in getting the multistep algorithm started.

Once a multistep algorithm is underway, the quality of the approximation at a timestep is successively improved as the window creeps upward. The first iterations over a timestep that occur when the timestep is at the top of a window may be thought of as establishing a rough approximation. As the window moves upward, the relative position of the timestep in the window goes down and the approximation to the solution at that timestep is refined. In the first iteration over the first window, the initial condition of the parabolic equation is used as the initial value at the beginning of each timestep in the window. The gradient of rough to fine approximation as a function of the position of the timestep in the window develops as the solution is carried out.

Figure 14 depicts the computational overhead involved in using windows of size greater than one in the solution of the above described heat equation. The solution time increases with the size of the window. Here the overhead was computed by timing the computer runs for a given block domain decomposition when windows of size one through four were examined. The ratio of the time required to advance the problem 50 timesteps with window 1, to the time required to complete the problem with window greater than one, is plotted for

each domain decomposition. For windows of size 2 the overhead observed ranged from 0.02 to 0.07, for windows of size 3 the overhead ranged from 0.07 to 0.13, and for windows of size 4 the overhead ranged from 0.09 to 0.21. The smallest overheads for each window size are seen when the domain is divided into 50 × 10 point blocks.

It is clear that while the use of windows does not affect asymptotic convergence, there is some computational overhead involved in their use that increases with window size. The experimental results described here show that the overhead is quite modest for small windows.

8. Conclusions

This paper explores methods for efficient solution of partial differential equations on MIMD machines. The general objective of this work is to maximize multiprocessor performance by rearranging the order of computations of standard algorithms so that the effects of communication delays are ameliorated, but at the same time the resulting algorithms have favorable, well defined convergence properties.

Using Jacobi and SOR point and block iterative methods as a basis, a new concept of windowing over several time-steps is developed. Both analytical and simulation results demonstrate the usefulness of windowing in decreasing the effects of communication delays on algorithm performance. The spectral radii of the iteration matrices of both of these new algorithms are equivalent to the spectral radii of the analogous standard methods [SALT85]. The use of windows entails a small computational overhead which increases gradually with window size. It was observed that the computational overhead associated with

a window size of two was negligible and the benefits in increased utilization were substantial.

Further investigations are being pursued in a number of ways. The concept of windowing can be extended to other iterative methods. The generalization of WBSOR to multicolor SOR, [ADAM82] or the orderings introduced by O'Leary [OLEA84] would appear to be particularly straightforward. The concept of windowing may also be extended to apply to iterative methods in the solution of the equations arising from Newton-like schemes for the solution of systems of nonlinear algebraic equations. It also may be possible to extend the windowing concept to the solution by functional iteration of nonlinear equations that might be obtained in a method of lines solution to nonlinear parabolic equations.

Acknowledgments

We are particularly indebted to M. Patrick, R. Voigt, and T. Gallie for stimulating and useful discussions and invaluable assistance in the editing of this paper.

References

[ADAM82] Adams, L. and Ortega, J. [1982]. "A Multi-Color SOR Method for Parallel Computation," Proc. 1982 Intl. Conf. Parallel Processing, pp. 53-56.

[BAUD78] Baudet, G. [1978]. "Asynchronous Iterative Methods for Multi-processors," J. ACM, Vol. 25, pp. 226-244.

[CHAZ69] Chazan, D. and Miranker, W. [1969]. "Chaotic Relaxation," J. Linear Algebra Appl., Vol. 2, pp. 199-222.

[DEMI82] Deminet, J. [1982]. "Experience with Multiprocessor Algorithms," IEEE Trans. Comput., Vol. 31, pp. 278-288.

[ERIC72] Ericksen, J. [1972]. "Iterative and Direct Methods for Solving Poisson's Equation and Their Adaptability to ILLIAC IV," Center for Advanced Computation Document No. 60, University of Illinois at Urbana - Champaign.

[FABE81] Faber, V. [1981]. "Block Relaxation Strategies," in <u>Elliptic Problem Solvers</u>, M. Schultiz (ed.), Academic Press, New York, NY, pp 271-275.

[FUJI83] Fujimoto, R. M. [1983]. "SIMON: A Simulator of Multicomputer Networks," Report No. UCB/CSD 83/140, Computer Science Division, University of California, Berkeley.

[GENT78] Gentleman, W. [1978]. "Some Complexity Results for Matrix Computation on Parallel Processors," J. ACM, Vol. 25, pp. 112-115.

[HAGE81] Hageman, L. A. and Young, D. M. [1981]. Applied Iterative Methods, Academic Press, New York.

[HAYE74] Hayes, L. [1974]. "Comparative Analysis of Iterative Techniques for Solving Laplace's Equation on the Unit Square on a Parallel Processor," M.S. Thesis, Department of Mathematics, University of Texas at Austin.

[HELL84] Heller, D. E. [1984]. "Multiprocessor Simulation Program SIMON," Internal Report, Physics and Computer Science Dept., Shell Development Company, Houston, Texas.

[HOCK81] Hockney, R. and Jesshope, C. [1981]. "Parallel Computers: Architecture, Programming and Algorithms," Adam Hilger, Ltd., Bristol.

[HWAN85] Hwang, K. [1985]. "Multiprocessor Supercomputer for Scientific/Engineering Applications," Computer, Vol. 18, No. 6, pp. 57-73.

[LAMB75] Lambiotte, J. and Voigt, R. [1975]. "The Solution of Tridiagonal Linear Systems on the CDC STAR-100 Computer," ACM Trans. Math. Softw., Vol. 1, pp. 308-329.

[OLEA83]   O'Leary, D. [1984]. "Ordering Schemes for Parallel Processing of Certain Mesh Problems," SIAM J. Sci. Stat. Comp., Vol. 5, pp. 620-632.

[ORTE85]   Ortega, J. and Voigt, R. [1985]. "Solution of Partial Differential Equations on Vector and Parallel Computers," SIAM Review, Vol. 27, No. 2, June 1985, pp. 149-240.

[PART80]   Parter, S. and Steuerwalt, S. [1980]. "On k-line and k x k Block Iterative Schemes for a Problem Arising in 3-D Elliptic Difference Equations," SIAM J. Numer. Anal., Vol. 17, pp. 823-839.

[PART82]   Parter, S. and Steuerwalt, S. [1982]. "Block Iterative Methods for Elliptic and Parabolic Difference Equations," SIAM J. Numer. Anal., Vol. 19, pp. 1173-1195.

[REED84]   Reed, D. A. and Patrick, M. L. [1984]. "A Model of Asynchronous Iterative Algorithms for Solving Large, Sparse, Linear Systems," Proceedings of the 1984 International Conference on Parallel Processing, Bellaire, Michigan.

[SAAD85]   Saad, Y. [1985]. "Communication Complexity of Gaussian Elimination Algorithm on Multiprocessors," Research Report, Yale University, YALEU/DCS/RR-348.

[SALT85]   Saltz, J. H., "Parallel and Adaptive Algorithms for Problems in Scientific and Medical Computing: Robust Methods for the Solution of

Partial Differential Equations on Multiprocessor Machines," Ph.D. Thesis, Duke University, April 1985.

[VARG62] Varga, R. S. [1962]. <u>Matrix Iterative Analysis</u>, Prentice-Hall, Englewood Cliffs, N.J.

[VOIG85] Voigt, R. G. [1985]. "Where Are the Parallel Algorithms?", 1985 National Computer Conference Proceedings, AFIPS Press, Reston, Virginia, pp. 329-334.

[YOUN71] Young, D. M. [1971]. <u>Iterative Solution of Large Linear Systems</u>, Academic Press, New York.

Iteration until convergence over consecutive timesteps
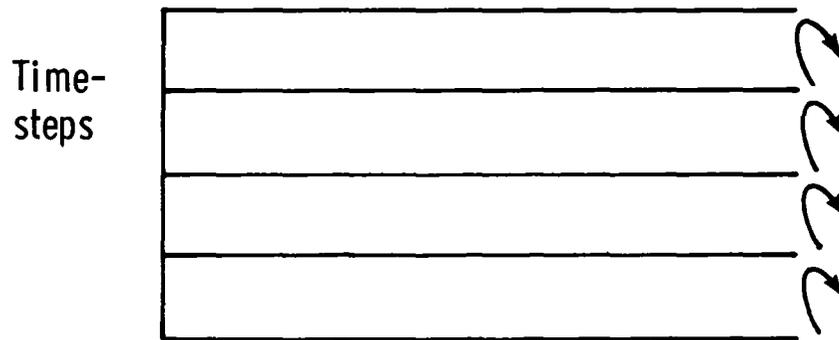
Each timestep is iterated until convergence

Figure 1

Multistep iterative methods

Window = 2
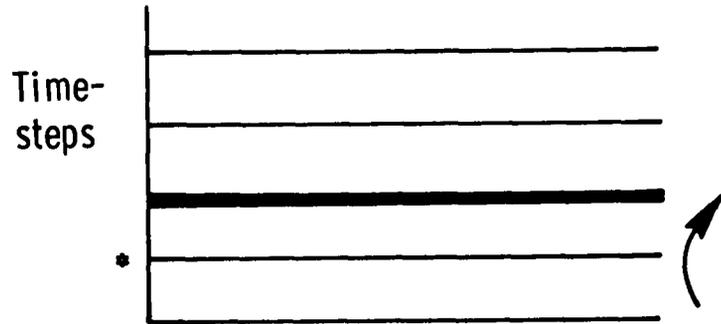
Iterate until convergence at (*).
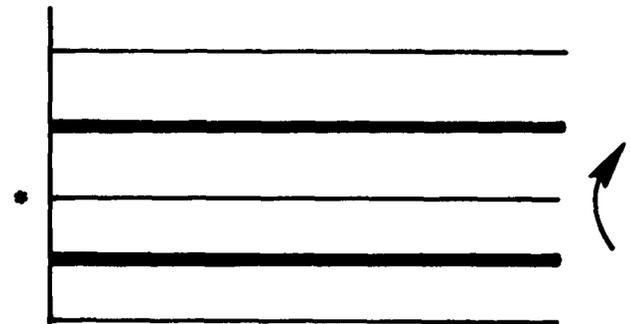
Shift window up <u>one</u> timestep.
Iterate until convergence at (*).

Figure 2

Counting of cumulative microsteps
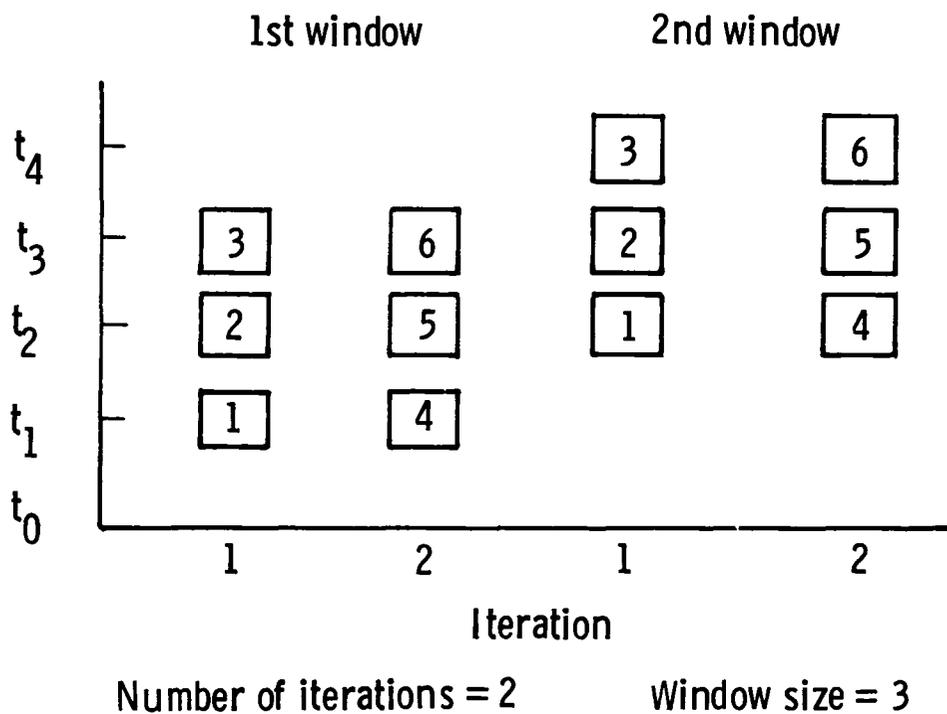
Figure 3
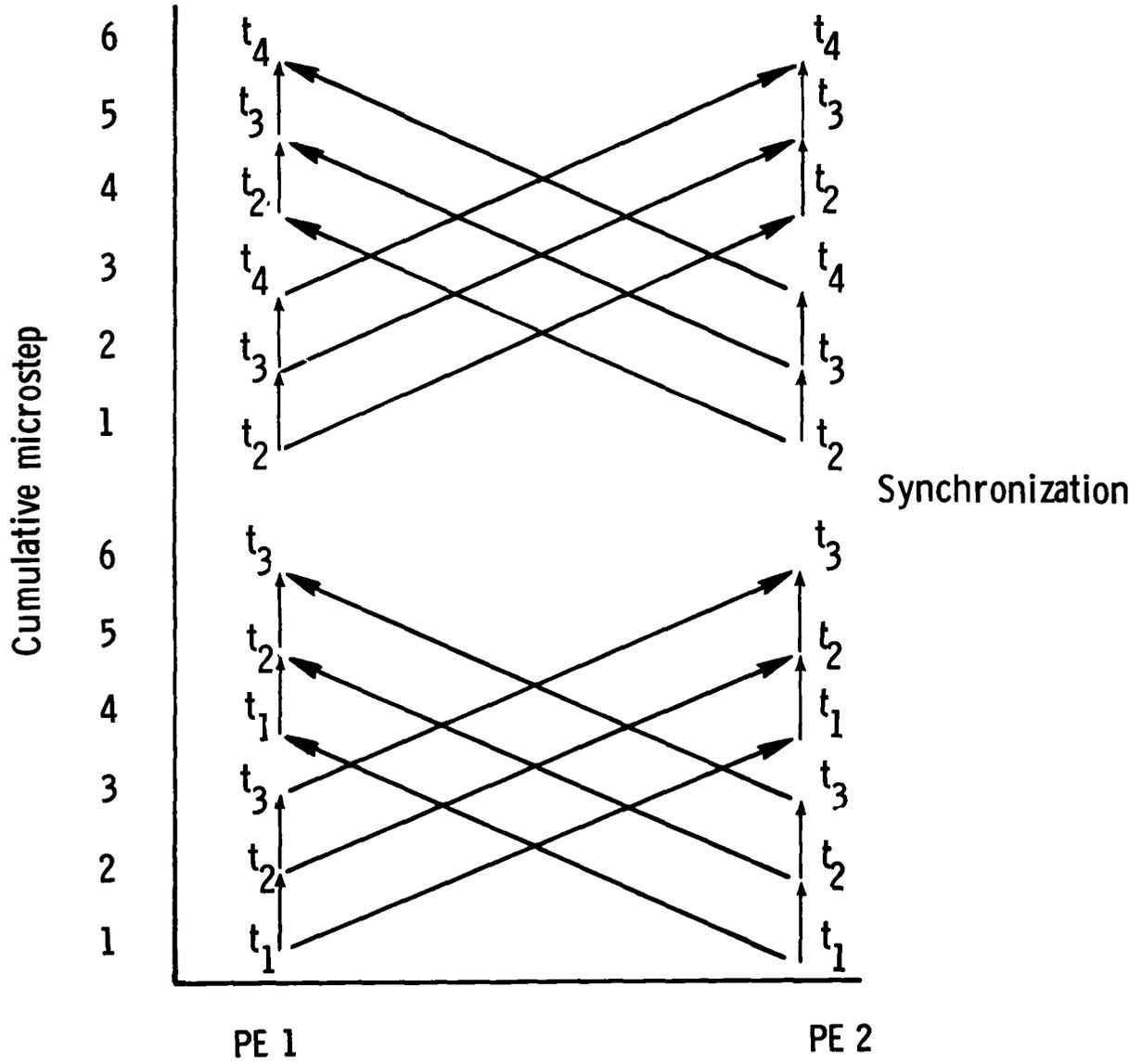
Figure 4
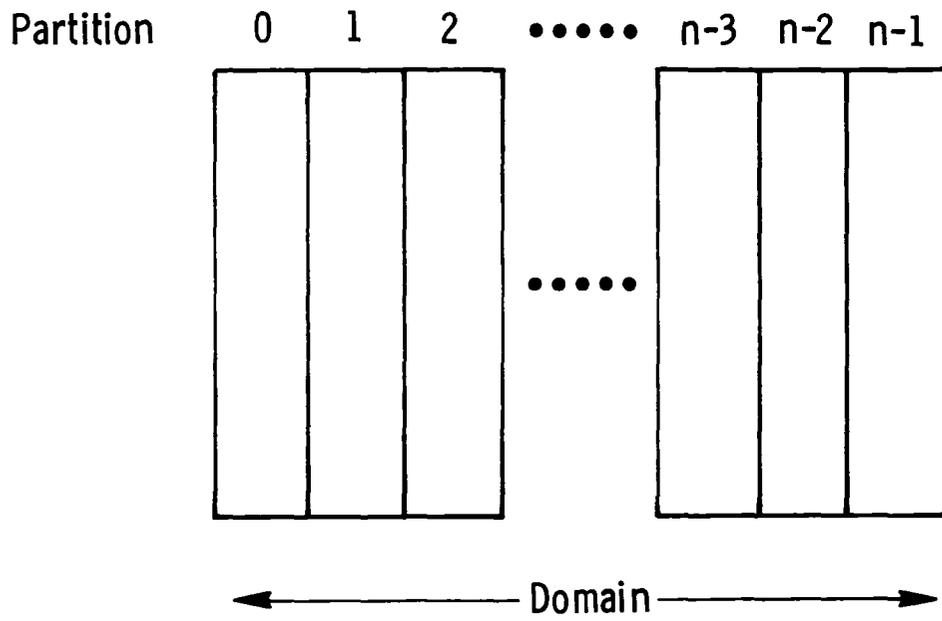
Figure 5
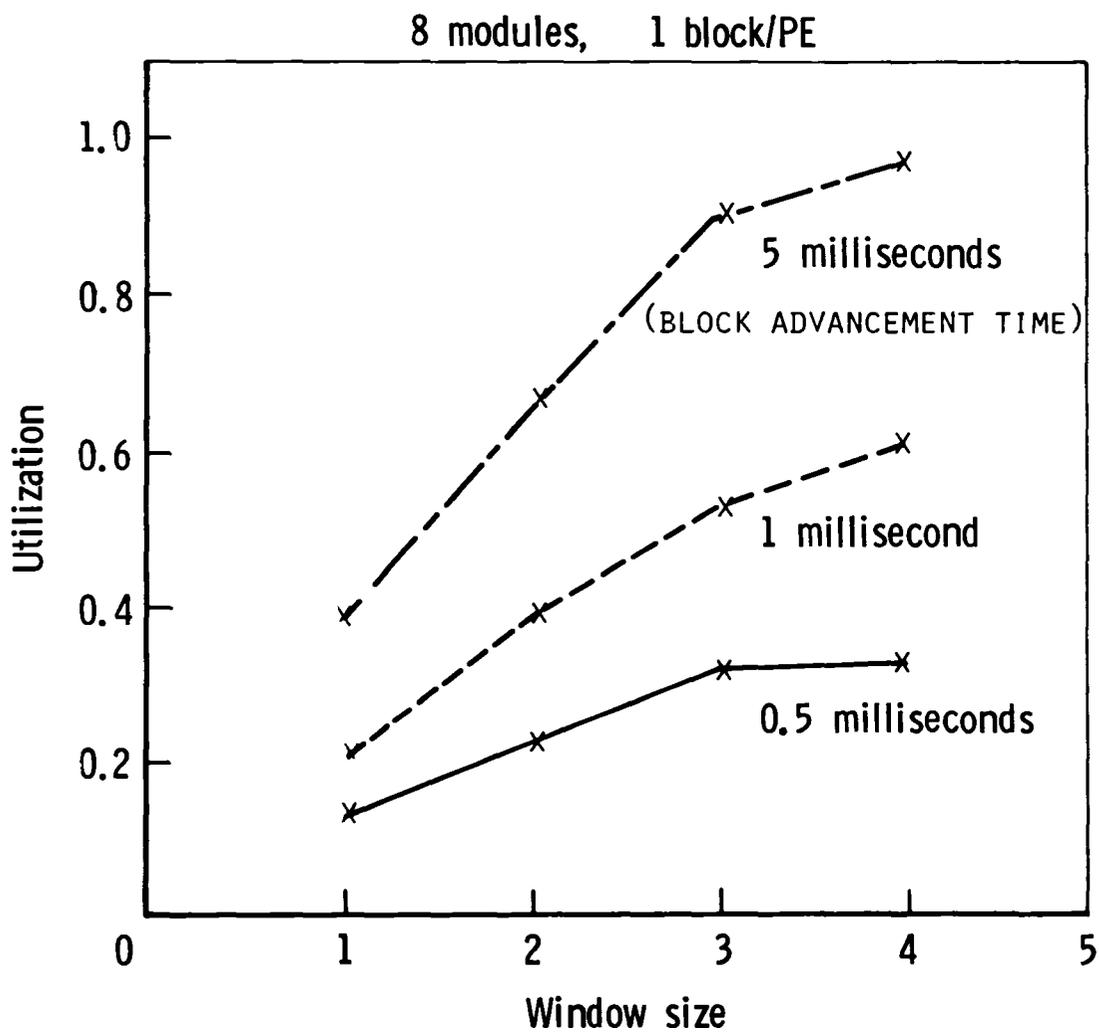
Figure 6
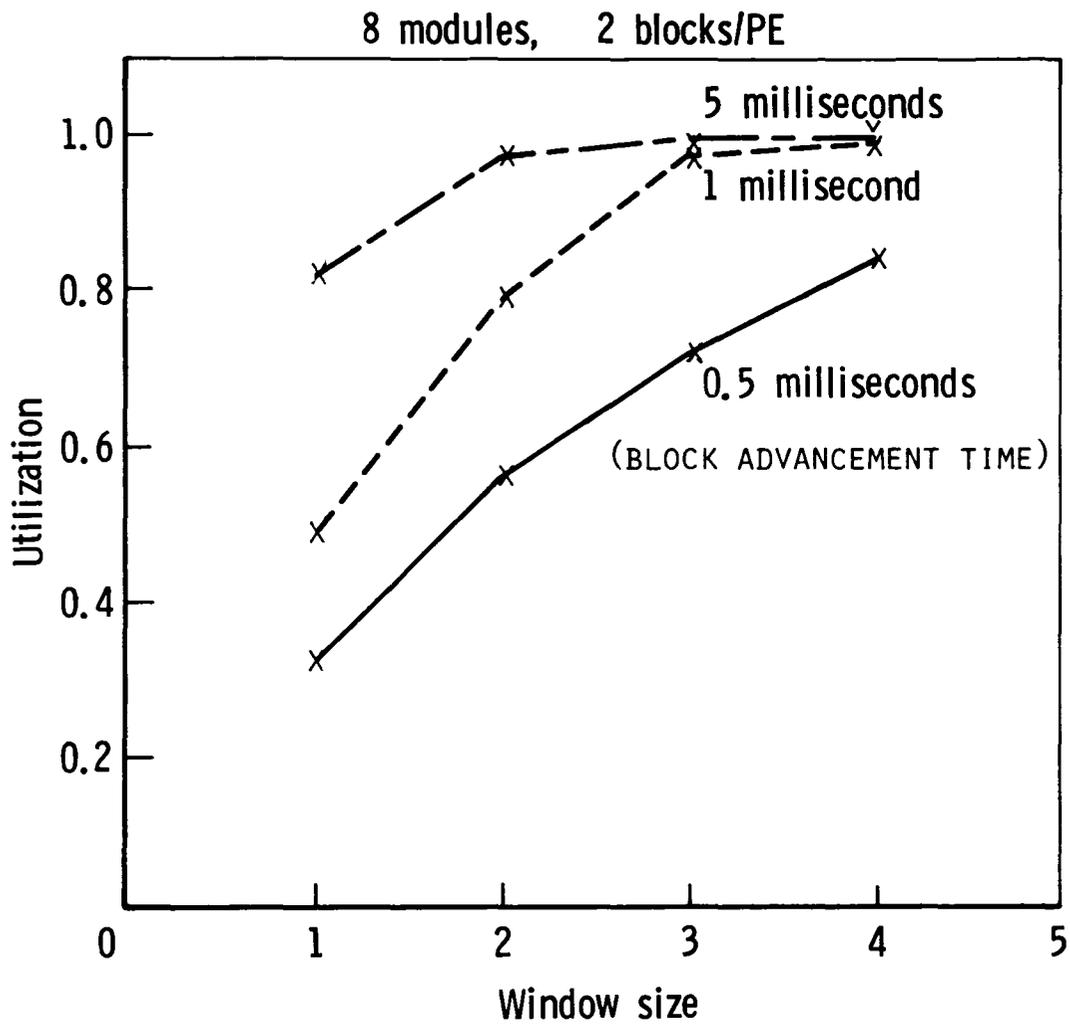
Figure 7

Figure 8

Figure 9

2 blocks/PE

1 millisecond per block advancement
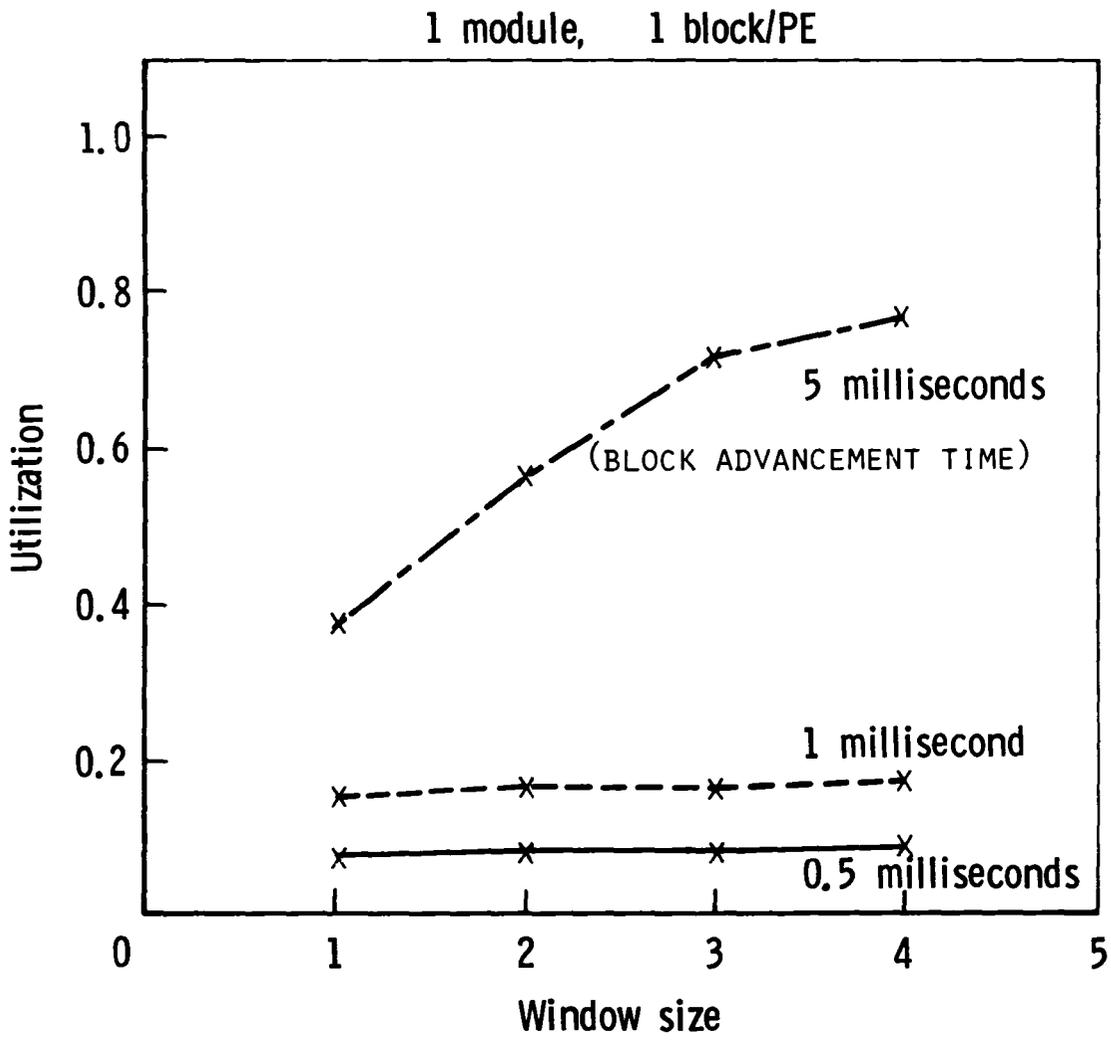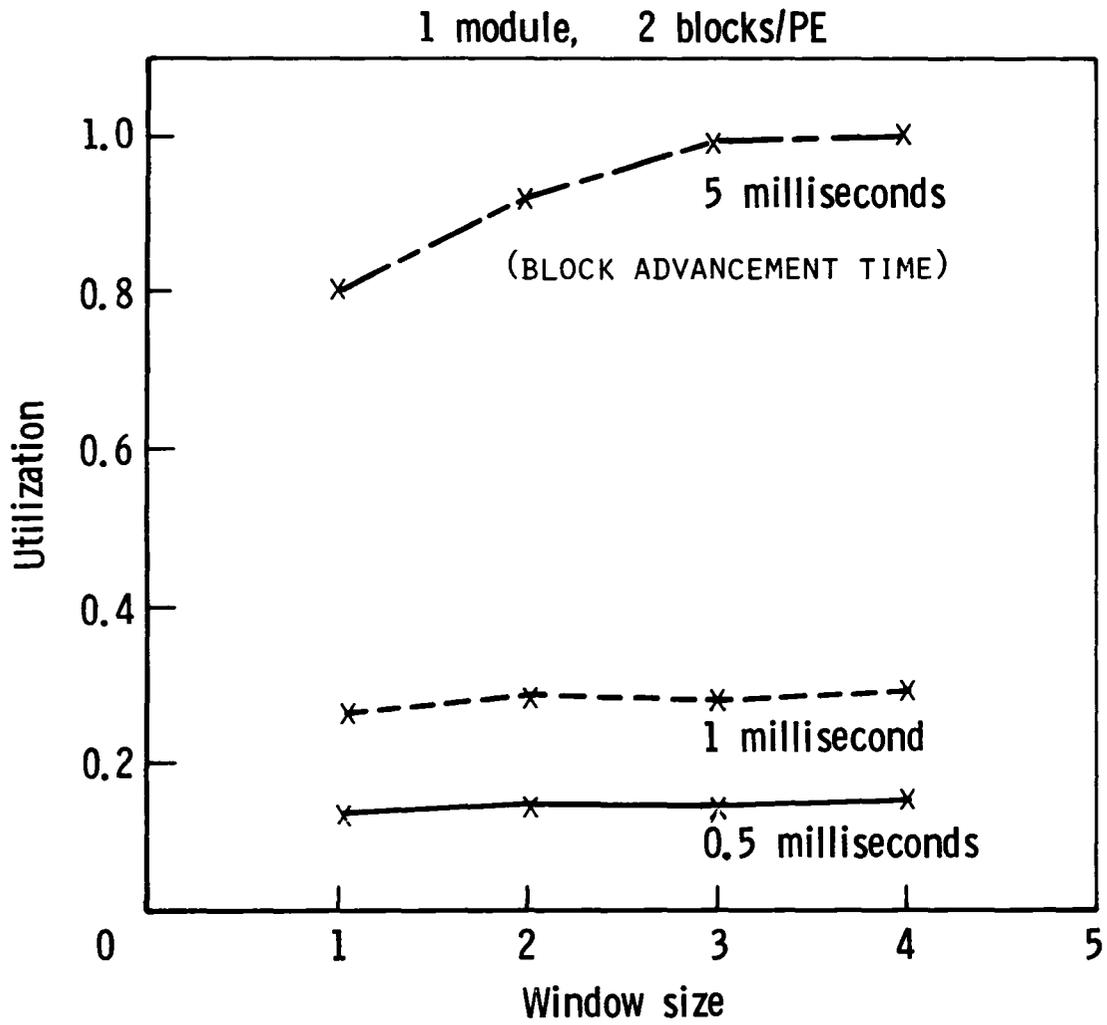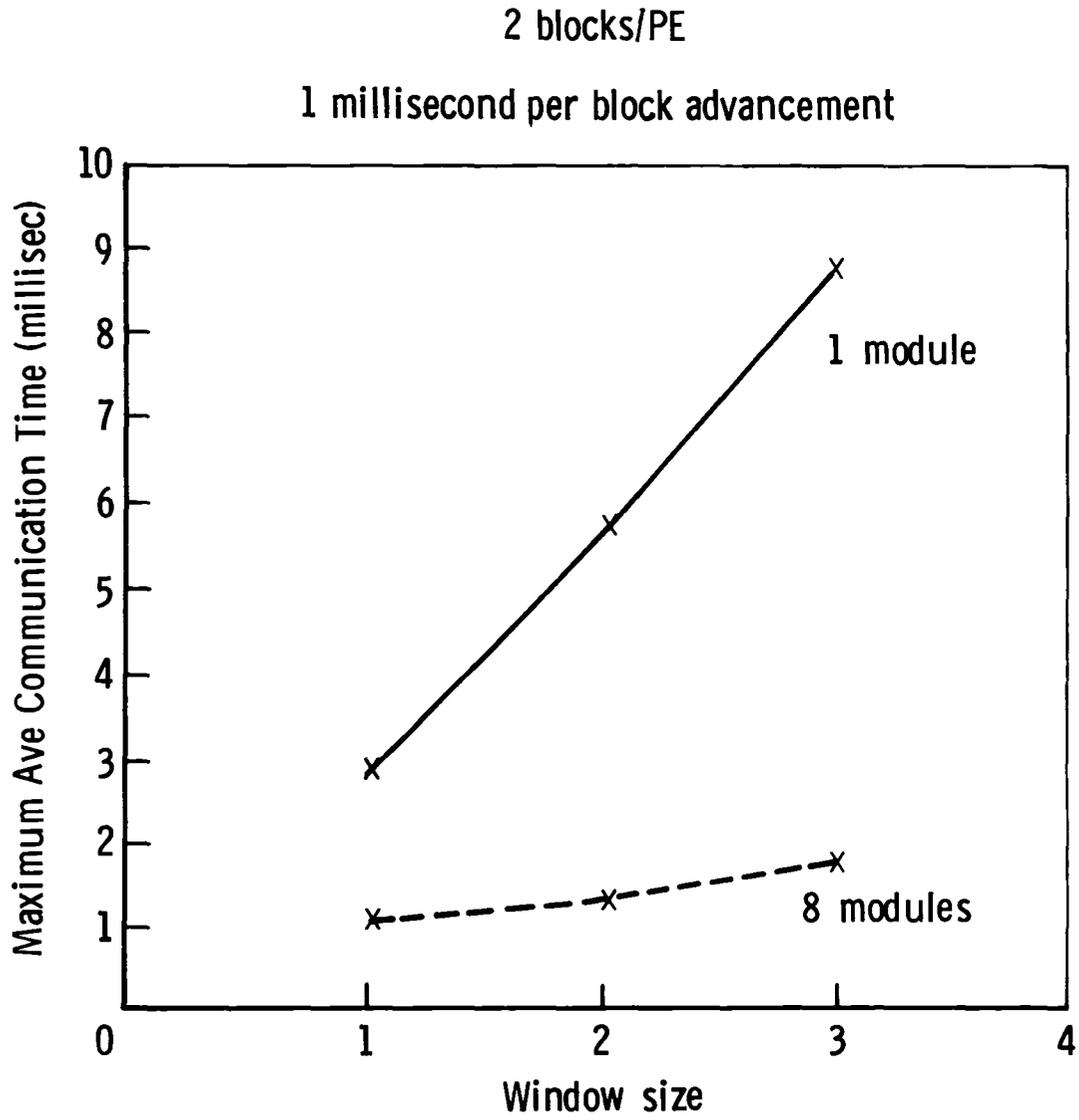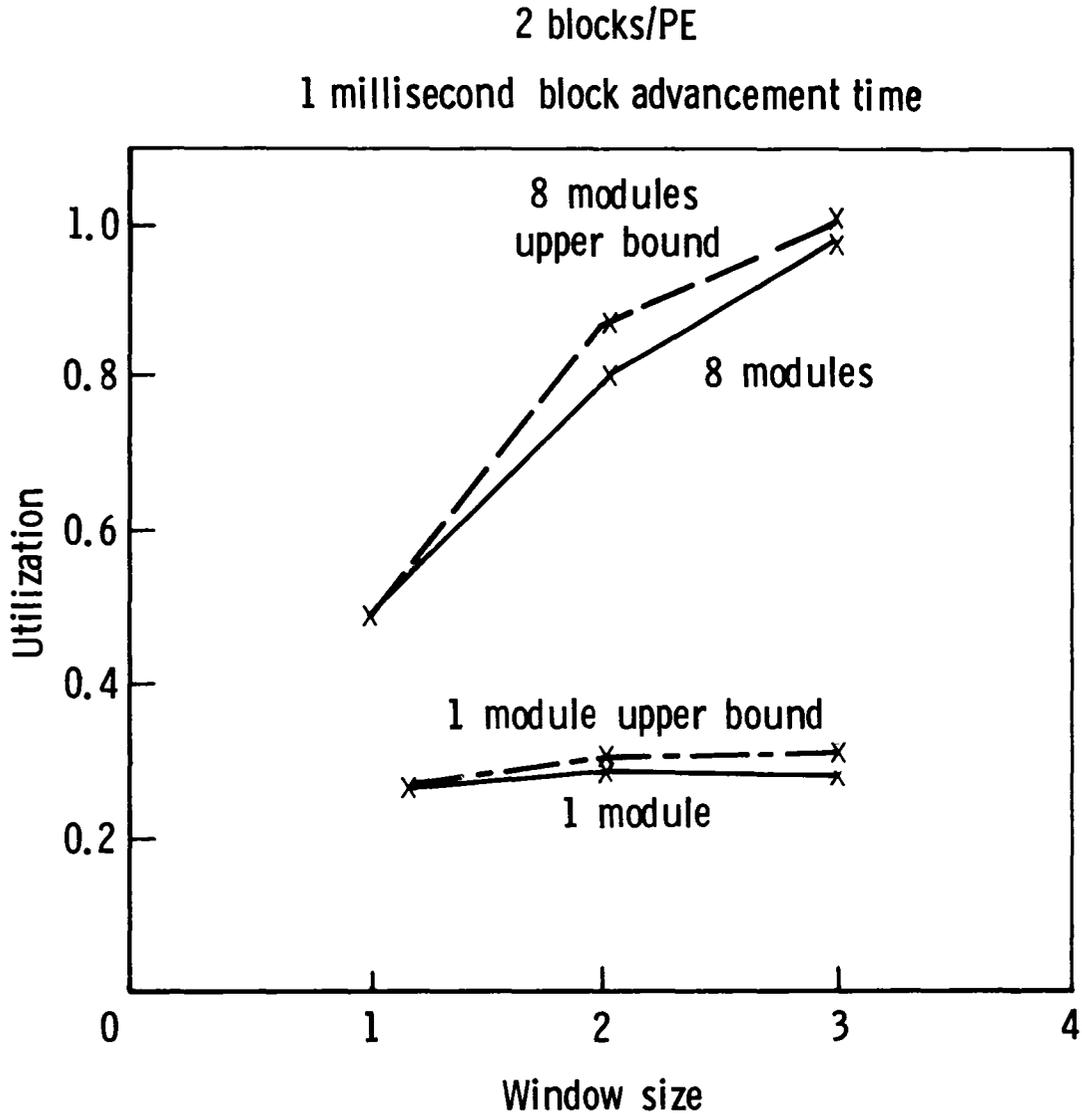
Figure 10

Figure 11

Heat equation— Dirichlet boundary conditions.
50 by 50 mesh, timestep 0.001, 50 timesteps.
10 by 10 meshpoint blocks, 1E-5 convergence.
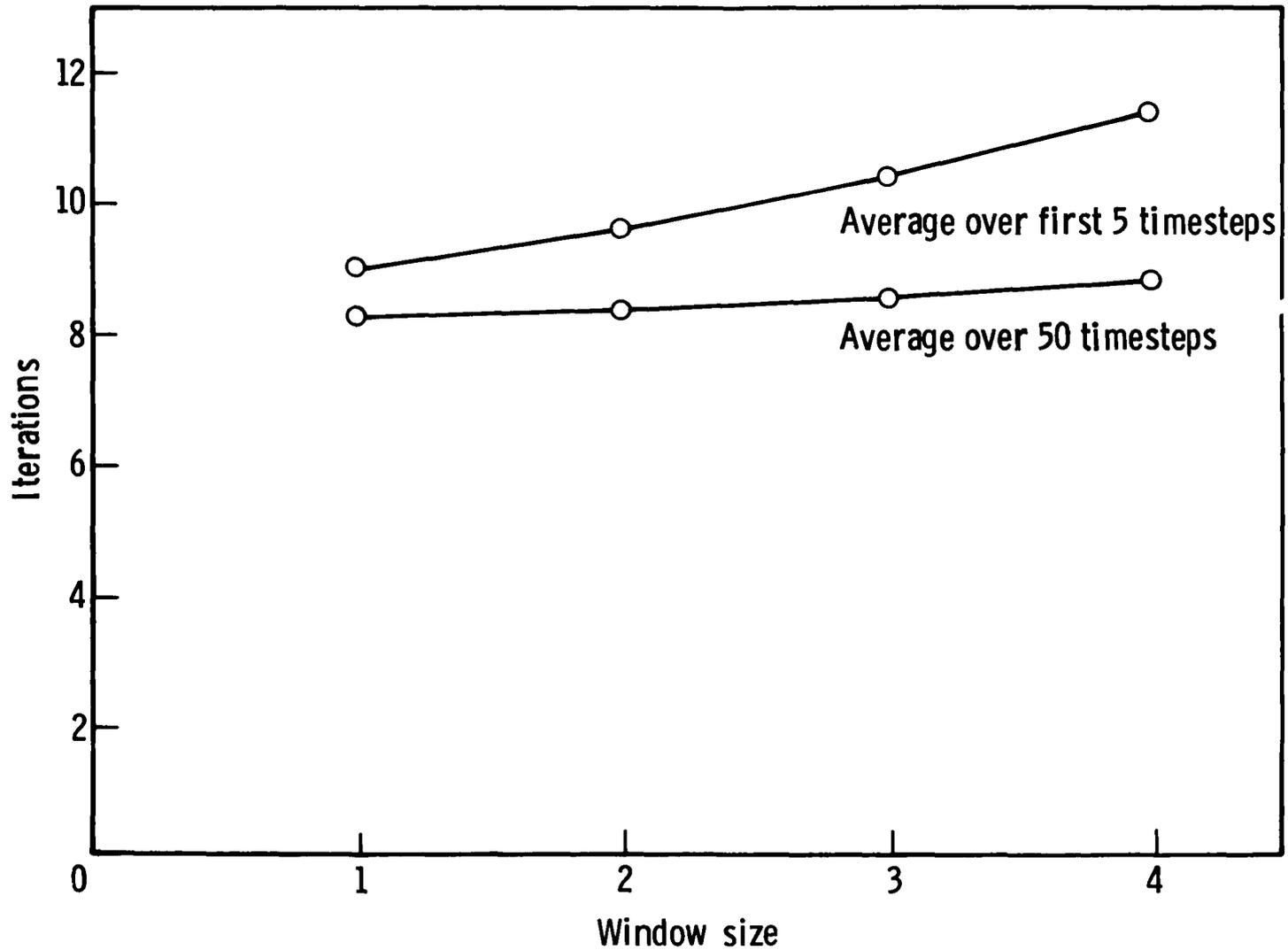
Average over first 5 timesteps

Average over 50 timesteps

Iterations

Window size

Figure 12

-48-

Heat equation—Dirichlet boundary conditions.
50 by 50 mesh, timestep 0.001, 50 timesteps.
50 by 2 blocks, 1E-5 convergence.
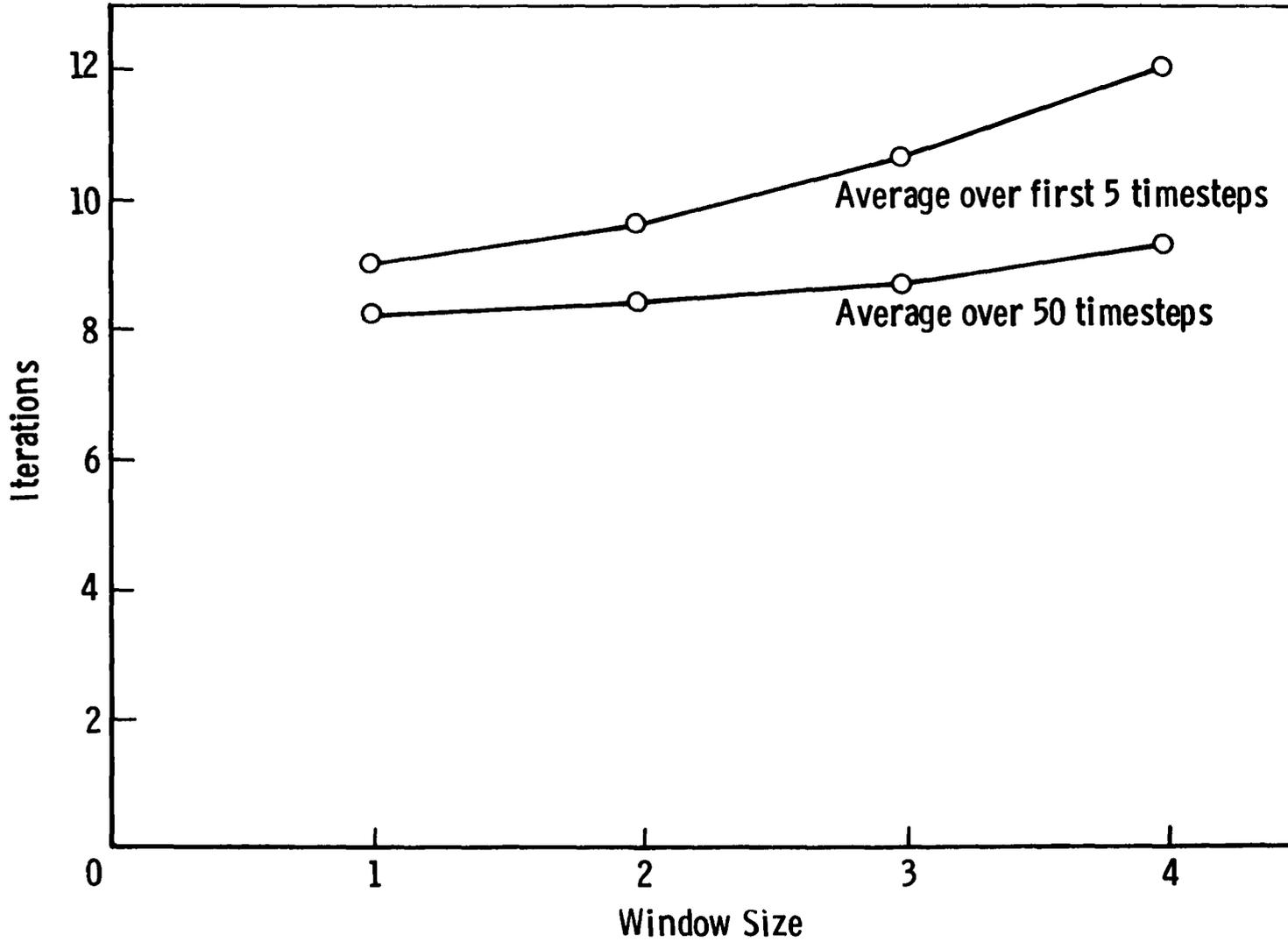
Figure 13

Heat equation—Dirichlet boundary conditions.
50 by 50 mesh, timestep 0.001, 50 timesteps.
Multiple of time required to do window = I, for convergence 1E-5.
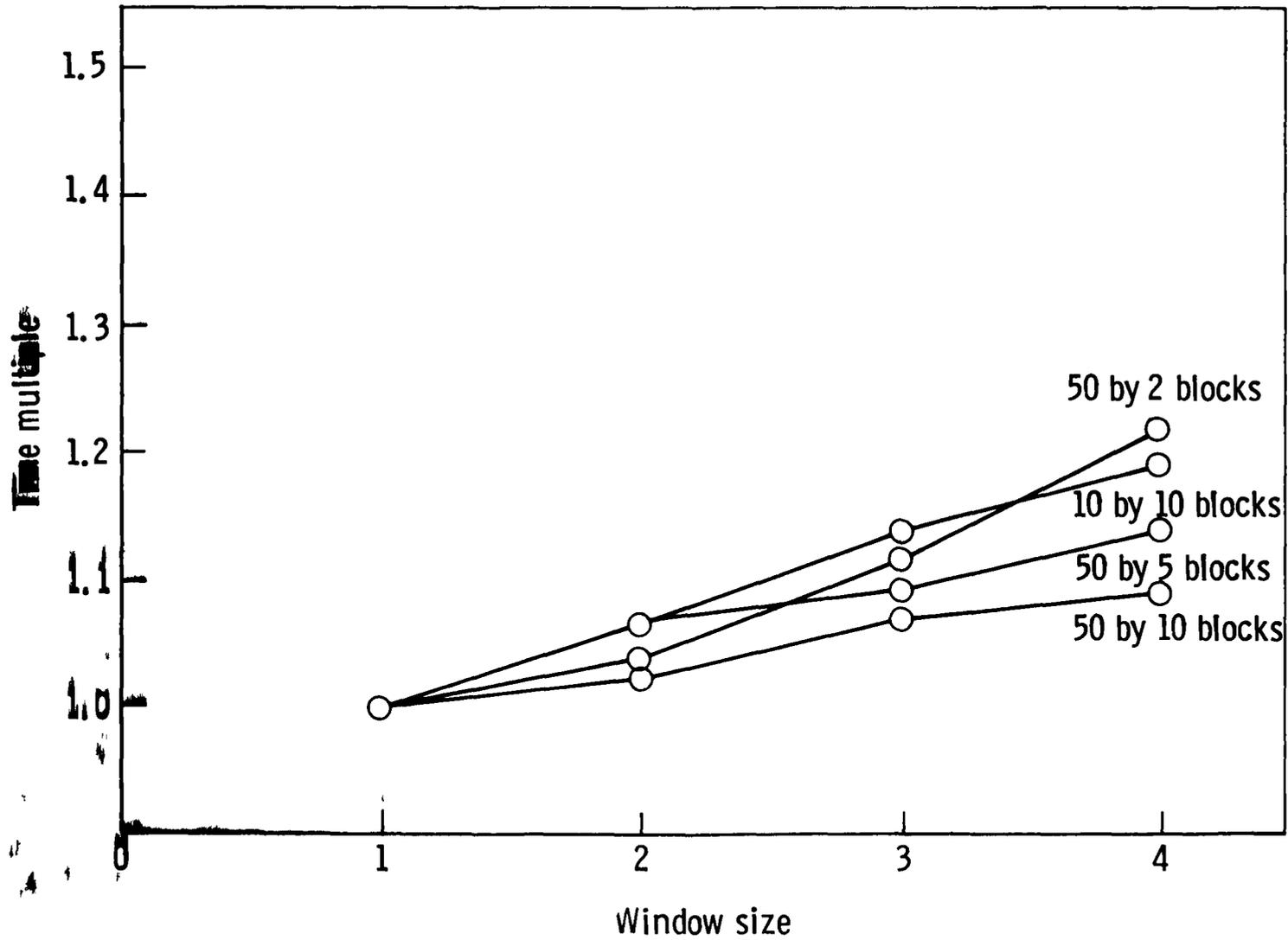
Figure 14

End of Document