NASA-CR-178,044

NASA Contractor Report 178044

ICASE REPORT NO. 86-4

NASA-CR-178044
19860009540

ICASE

REDUCTION OF THE EFFECTS OF THE COMMUNICATION DELAYS IN SCIENTIFIC ALGORITHMS ON MESSAGE PASSING MIMD ARCHITECTURES

Joel H. Saltz
Vijay K. Naik
David M. Nicol

Contract Nos. NAS1-17070 and NAS1-18107
January 1986

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

Operated by the Universities Space Research Association

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665

Reduction of the Effects of the Communication Delays in Scientific Algorithms on Message Passing MIMD Architectures

Joel H. Saltz
Vijay K. Naik
David M. Nicol

Institute for Computer Applications in Science & Engineering
NASA, Langley Research Center, Hampton, Va 23665

Abstract

The efficient implementation of algorithms on multiprocessor machines requires that the effects of communication delays be minimized. The effects of these delays on the performance of a model problem on a hypercube multiprocessor architecture is investigated, and methods are developed for increasing algorithm efficiency. The model problem under investigation is the solution by red-black Successive Over Relaxation of the heat equation; most of the techniques to be described here also apply equally well to the solution of elliptic partial differential equations by red-black or multicolor SOR methods.

This paper identifies methods for reducing communication traffic and overhead on a multiprocessor and reports the results of testing these methods on the Intel iPSC Hypercube. We examine methods for partitioning a problem's domain across processors, for reducing communication traffic during a global convergence check, for reducing the number of global convergence checks employed during an iteration, and for concurrently iterating on multiple time-steps in a time dependent problem. Our empirical results show that use of these methods can markedly reduce a numerical problem's execution time.

Research was supported by the National Aeronautics and Space Administration under NASA Contract Nos NAS1-17070 and NAS1-18107 while the authors were in residence at ICASE, NASA Langley Research Center, Hampton, VA 23665.

N86-19010#

1. Introduction

The efficient implementation of algorithms on multiprocessor machines requires that the effects of communication delays be minimized. Reduction of communication delay effects in message passing machines may be brought about by restructuring algorithms. Ways in which the effect of communication delays can be reduced by such restructurings include: (1) reducing the quantity of information that must be communicated, (2) reducing the the frequency with which messages must be sent, (3) overlapping communication with computation. The above goals may not in practice be mutually compatible. The relative importance of the three aspects of communication delay reduction will depend on the architecture under consideration [SALT85a], [SALT85b],[VOIG85].

The effects of these delays on the performance of a model problem on a hypercube multiprocessor architecture are investigated, and methods are developed for increasing algorithm efficiency. A hypercube multiprocessor[RATT85] is a collection of processors or nodes connected by a communication network with a hypercube topology. A hypercube has 2^d identical nodes where d represents the dimension of the hypercube. Each node in a hypercube is connected to d other neighbors. Nodes are assigned addresses from 0 to 2^d-1. Two nodes of a hypercube are connected when the binary expansion of the nodes' addresses differs in one bit position.

The model problem under investigation is the solution by red-black Successive Over Relaxation [YOUN71] of the heat equation; most of the techniques to be described here also apply equally well to the solution of elliptic partial differential equations by red-black or multicolor SOR methods. The model problem is solved on an N dimensional hypercube by decomposing the domain into 2^N regions and assigning one region to each processor. The regions are chosen either as strips or as rectangles and are mapped onto the architecture using a grey code [SAAD85]. Due to the grey code mapping, processors assigned adjacent regions of the domain are directly connected.

The two sources of communication delays in a simple iterative method such as SOR are the need to exchange information between the boundaries of the subdomain assigned to each processor and the need to check convergence. We examine a variety of factors which help determine the interprocessor communication costs. In this paper methods of reducing both of these sources of communication delays are proposed.

The time required to transmit a packet of information from one processor to another to which it is directly linked may be approximately expressed as

$$\alpha \cdot b + \beta$$

where b is the number of bytes contained in the message, α is the bandwidth of the communication channel, and β is the overhead for sending a message. When α is considerably smaller than β, overhead for communication is high in comparison to the communication bandwidth. In this case, there may be significant performance advantages in arranging an algorithm so that information that must be transmitted is sent in large quantities.

In the Intel hypercube used in the experiments described in this paper, $\beta \gg \alpha$, and communication may not be overlapped with computation to any appreciable extent. Given a multiprocessor with these characteristics, reducing the number of messages that must be sent by processors is consequently the main goal of the work to be described. In sections 3,4 and 5 methods are described that may be used to reduce the number of messages that must be sent by each processor and to consequently reduce the effect of the communication overhead β. Section 3 explores the consequences on the hypercube performance of the simple observation that the number of messages that must be sent by a processor when a domain is partitioned into strips is at most two, while a processor may have to send four messages when a domain is partitioned into square regions. Allowing iterations to sweep over more than one timestep is a method used in section 5 to decrease the number of messages that must be sent.

In a hypercube, it is possible to perform communications that combine results from all processors and to disseminate the results thus obtained in a time that grows logarithmically in the number of processors involved. When communication overheads are large, convergence testing may be quite costly despite this logarithmic growth. In section 4.1 two logarithmic methods for combining the results obtained from local convergence checks are advanced and compared. In both methods, for all but very small hypercubes, the communication delays resulting from convergence checking are comparable or greater in size to the delays arising from the communication of the boundary variable values.

In both of the above schemes, communications for global convergence checking occur after each iteration. Two methods are proposed and tested for reducing the frequency with which communication is required for global convergence checking. The first method discussed in section 4.2 checks for global convergence only when certain necessary conditions are fulfilled. One necessary condition is that all subdomains have detected convergence at some point in their computations. Other necessary conditions result from the fact that global convergence requires all processors to detect local convergence at a given iteration. In section 4.3 a method is proposed that utilizes a logarithmic method for checking for convergence but employs a statistical methodology to schedule convergence checks only at critical iterations.

In the following sections experimental results will be presented that pertain to the solution of a model problem. The partial differential equation being solved is the heat equation on the unit square with dirichlet boundary conditions and with the first two modes used as the initial condition. The heat equation is solved using optimally over-relaxed red-black SOR on 64 by 64, 128 by 128, 256 by 256 point meshes with timesteps of 0.004, 0.002 and 0.001 respectively. All experimental results were obtained using a 64 processor Intel iPSC hypercube.

2. Effect of Problem Size on Performance

As has been widely reported [FOX84],[ORTE85], in order to obtain better performance from the multiprocessors, one should have a balance between computation and communication costs. If the communication costs are too high as compared to the costs of computations, then the performance is bound to deteriorate. The obvious way to improve the performance is to increase the size of the subdomain assigned to each processor without increasing the communication costs proportionately. In Figure 1, we depict the performance of the system in terms of efficiency, as the domain size is varied from 64 by 64 through 256 by 256 grid sizes. Here the *efficiency* of an N-processor system is defined as the ratio of the time taken to solve the problem on one processor to the time taken to solve the problem on N processors, times the number of processors. As expected the efficiency drops as the number processors is increased, but the rate at which it drops is much more gradual as the grid size is increased. In all the experiments performed here the domain was subdivided into strips. The model problem was solved for five timesteps, and the efficiencies were computed by measuring the elapsed time to find the solutions of the first five time-steps.

3. The Effect of Domain Partitioning on Performance

The cost of communicating information from one processor to another in a hypercube multiprocessor is a function of the amount of data that must be sent, the number of packets of data into which the data is placed, and the logical distance of the processors from one another in the hypercube. The domain of a PDE may be decomposed into regions with a variety of shapes, with certain shapes provably optimal with respect to minimizing the number of variable values that must be communicated across boundaries.[1] The regions may then be mapped onto a hypercube in a way that attempts to minimize the number of intermediate nodes that messages must traverse in going from the boundary of one region to another [SAAD85].

We considered domain decompositions consisting of strips and rectangles; such shapes are easier to program and can be mapped onto a hypercube so all processors that need to send messages to one another are directly linked. It is easily demonstrated that

[1] Reed, D., Patrick, M., and Adams, L. to be submitted to IEEE Transactions on Computers

in a domain divided into rectangles, less information must be transmitted across boundaries during each iteration than would be the case with a domain divided into strips. On the other hand, in a domain divided into rectangles, regions may have four neighbors while when the domain is divided into strips, each region may have no more than two neighbors.

We examine the trade-off in costs between the division of a domain into rectangles and the division of a domain into strips in C color SOR. Assume a 2^n by 2^n point domain and 2^m processors. The domain may be divided into 2^m $2^{n-\lfloor \frac{m}{2} \rfloor}$ by $2^{n-\lceil \frac{m}{2} \rceil}$ rectangles, or alternately into 2^n by 2^{n-m} strips. In C color SOR the values of points are adjusted one color at a time. Each time a color is adjusted, all rectangles in the interior of the domain must send four packets. If the number of colors used in the SOR sweeps is not equal to the number of points on a side of a rectangle or strip, the number of values to be communicated may differ by one between sweeps over different colors.

Rectangles in the interior of a domain, i.e. those with four neighbors during the course of each iteration, must send $2C$ packets of average size $\dfrac{2^{n-\lfloor \frac{m}{2} \rfloor}}{C}$ and must send $2C$ packets of average size $\dfrac{2^{n-\lceil \frac{m}{2} \rceil}}{C}$. Strips in the interior of a domain, i.e. those with two neighbors, must send $2C$ packets of average size $\dfrac{2^n}{C}$.

The comparison of costs between strips and rectangles depends on the overhead for sending each message, on the per-byte cost of transmitting information, on the number of processors, and on the size of the domain. Figure 2 depicts a comparison between local communication costs when the model problem is solved with domains of varying size. For domains of size 256 by 256 or smaller, the use of strips led to smaller communication delays than the use of rectangles, while for a domain with 512 by 512 mesh points the use of rectangles led to the smaller delays. Figure 3 depicts the local communication costs for rectangles compared to the local communication cost for strips for varying numbers of processors in a 64 by 64 point domain. Note that the communication cost for rectangles exceeds that for strips when at least eight processors are utilized. When eight or fewer processors are used, the number of packets that must be sent in a domain divided into rectangles is equal to the number of packets that must be sent in a strip divided domain. The above local communication costs were measured in the following way. The model problem was solved for a given domain size for 50 iterations over 3 timesteps, and then for the same domain size the program was run without sending any messages. The difference in the execution times between the program runs that sent messages and those that did not was used to give an estimate of time spent in communicating boundary information. Note that neither of the program runs performed any communication for convergence checking.

4. Convergence Checking Schemes

In this section it will be shown that the communication required in testing for global convergence at the end of an iteration may be quite costly. Several methods for reducing this delay are then experimentally examined.

Our experiments presumed that convergence had been achieved when

$$\left|\left|X_i - X_{i-1}\right|\right|_\infty \leqslant \epsilon$$

where X_i is the vector valued solution approximation after the *ith* iteration, and ϵ is our tolerance. The $||\cdot||_\infty$ norm above yields the maximal absolute difference between components of X_i and X_{i-1}. Our techniques do not depend on this particular norm; any other norm could be used. Convergence checking in a multiprocessor involves two distinct costs. The first is simply the time required to compute the component differences. The second cost is the time required for the processors to communicate and combine their respective local convergence results to determine whether global convergence is achieved at the end of an iteration. This latter cost is a function of the multiprocessor communication delay and the scheme used to combine the component differences. We will first look at ways to combine differences during a global convergence check on the hypercube. We then discuss two different ways of reducing the *number* of convergence checks used during an iterative solution.

4.1. Combination Methods

We compared two different ways of combining component differences during a convergence check. Both of these methods first require each processor to find the maximal component difference over its own piece of the domain. Each processor then sets its *convergence flag* equal to 1 if this maximal difference over the processor's domain is less than ϵ; the flag is otherwise 0. Clearly convergence is achieved if and only if every processor's convergence flag is 1. The two methods differ in how they cause processors to exchange and combine these flags. Both schemes have time complexity $O(log_2(m))$, m

being the dimension of the hypercube.

The first scheme, to be called the tree method, requires $2m$ stages. In m stages, a logical AND is taken of all convergence flags and the resulting flag ends up in node 0. If the value of this logical AND is 1, global convergence has occurred; otherwise global convergence has still not been detected. During each stage k, $0 \leqslant k < m$ of communication, each of $2^m/(k+1)$ processors sends to another processor a flag that indicates its current knowledge of whether global convergence has occurred. Let $<d_{m-1},\ldots,d_0>$ be the binary expansion of the address of a node, and let $\overline{d_k}$ represent the complement of binary digit d_k. Let $Flag^k{}_{<d_{m-1},d_k,\ldots,d_0>}$ represent the current knowledge of global convergence in the node with address $<d_{m-1},d_k,\ldots,d_0>$ at the beginning of communication stage k. In stage k, processor $<d_{m-1},d_k,\ldots,d_0>$ will send $Flag^k{}_{<d_{m-1},d_k,\ldots,d_0>}$ to processor $<d_{m-1},..\overline{d_k},\ldots,d_0>$ if $d_0= \cdots =d_{k-1}=0$ and $d_k = 1$. Upon the receipt of the flag, processor $<d_{m-1},..\overline{d_k},\ldots,d_0>$ will set $Flag^{k+1}{}_{<d_{m-1},\overline{d_k},\ldots,d_0>} = Flag^k{}_{<d_{m-1},..\overline{d_k},\ldots,d_0>}$ AND $Flag^k{}_{<d_{m-1},d_k,\ldots,d_0>}$. This process terminates at Node 0. Node 0 at this point distributes the information on whether global convergence has occurred. Distributing the convergence results also requires m stages of communication.

Another method of checking for convergence can be accomplished in m stages of communication. This procedure is similar in principle to the cascade method for computing sums [HOCK81] and will be called the cascade method for checking convergence. In this method, during each stage of communication each processor sends a flag $Flag^k{}_P$ indicating its current knowledge of whether global convergence has occurred to another processor. At the end of the m stages, the resulting flag obtained in all processors is the logical AND of all convergence flags.

The cascade process functions as follows: Let $<d_{m-1},\ldots,d_0>$ be the binary expansion of the address of a node. In stage k, $0 \leqslant k < m$, processor $<d_{m-1},d_k,\ldots,d_0>$ will send $Flag^k{}_{<d_{m-1},d_k,\ldots,d_0>}$ to processor $<d_{m-1},..\overline{d_k},\ldots,d_0>$. Upon receipt, processor $<d_{m-1},..\overline{d_k},\ldots,d_0>$ will set $Flag^{k+1}{}_{<d_{m-1},\overline{d_k},\ldots,d_0>}$ equal to the logical *AND* of flags $Flag^k{}_{<d_{m-1},..\overline{d_k},\ldots,d_0>}$ and $Flag^k{}_{<d_{m-1},d_k,\ldots,d_0>}$. This process terminates after m stages, and at this point the flag $Flag^m{}_{<d_{m-1},d_k,\ldots,d_0>}$ in each processor is the logical AND of all convergence flags. The flow of data corresponding to the two convergence checking processes is depicted in figure 4. This figure illustrates that the tree method has a single processor detect and then report global convergence; the cascade method requires all processors to calculate the global convergence state.

The cost per iteration of the tree and cascade convergence checking methods is depicted in figure 5, along with the cost per iteration of local communication between strips and the per-iteration computation cost for a 64 by 64 mesh point model problem. The cost of convergence checking is estimated by running the model problem for 3 timesteps, 50 iterations per timestep both with and without the convergence checking methods and comparing the run times. When the convergence checking methods were utilized, a minor modification in the program caused the convergence results thus obtained to be ignored. The additional time required for convergence checking information could hence be ascertained by comparing the timings of programs that otherwise performed identical computations.

The cascade convergence checking method requires each processor to send and to receive four messages per iteration when a cube with 16 processors is used. Each iteration, four messages containing boundary variable values must also be sent and received by nodes assigned strips that are on the interior of the domain. When a cube with 16 processors is utilized, the communication cost of sending boundary variable values and of sending convergence flag information is indeed comparable. The cost of sending the boundary variable values is slightly greater, presumably due to larger amounts of data per packet.

The cost of the tree method of convergence checking is only slightly greater than the cost of the cascade method despite the fact that it requires twice as many stages. In the tree method, during each stage of the computation a processor is called upon either to send *or* receive a message, while in the cascade method each processor must both send and receive a message during each stage. Further experimentation has indicated that in a number of contexts, there appears to be a substantial time penalty associated with requiring a processor to both send and receive a message during a given stage of communication.

4.2. Asynchronous Convergence Checking

This section discusses one means of reducing the number of convergence checks required during an iterative solution. This scheme is called *asynchronous convergence checking*, or the *ACC* scheme. The *ACC*'s basic idea is to check for global convergence only when certain necessary conditions are fulfilled. One necessary condition is that all subdomains have detected convergence at some point in their computations. Other necessary conditions, to be expanded upon later, result from the fact that global convergence requires that all processors detect local convergence at a given iteration.

Discussion of the *ACC* is facilitated by a few definitions. At the end of an iteration, a processor's subdomain is in one of two states: *nonconverged*, or *presumptively converged.* This nomenclature emphasizes that convergence over a processor's subdomain does not guarantee global convergence at that iteration, nor does it prohibit a subdomain from oscillating between nonconvergence and presumptive convergence. Global convergence is achieved if and only if all the subdomains achieve presumptive convergence at the end of the same number of iterations. If the computations were to continue, all subdomains would be expected to remain in this state indefinitely. A *convergence sequence* for a subdomain is a maximal sequence of iterations during which the subdomain is presumptively converged; the first iteration of the sequence is the *convergence sequence header.* Thus a convergence sequence header identifies an iteration where a subdomain passes from nonconvergence to presumptive convergence. A subdomain's oscillation between nonconvergence and presumptive convergence gives rise to a series of convergence sequences, each with a distinct header. It is important to note that if global convergence is achieved at iteration j, then j is a convergence sequence header for at least one subdomain. The *ACC* method looks for global convergence only at certain iterations which serve as convergence sequence headers of one or more subdomains, but not at all the convergence sequence headers found in the system. The global convergence test is made centrally, in a processor known as the *c-host.* The high cost of communication between Intel hypercube nodes and their *system host* led us to designate a hypercube node as c-host for global convergence checks.

Under the *ACC* method, the c-host makes an informed guess at when the global convergence might have been achieved; after communicating with the other processors, the c-host either confirms or rejects the guess. A rejection is followed by another guess. This process is continued until the confirmation takes place. Implementation of *ACC* requires each processor to always maintain its subdomain's current convergence state, and the convergence sequence header if the subdomain is presumptively converged. We now separately describe the three component parts of the method, the **initial guess**, the **processor guess response**, and the **guess confirmation/generation.**

Initial Guess

As soon as a subdomain is presumptively converged for the first time, its processor reports the corresponding convergence sequence header to the c-host. The c-host makes the first guess after receiving such a message from each processor; let $\{j_1, \cdots, j_k\}$ be the received header values. The c-host optimistically guesses that global convergence is achieved as early as is possible, at iteration $j_{\max} = \max\{j_1, \cdots, j_k\}$. The c-host then tells each processor that $j_{\max}$ is a potential point of global convergence.

Processor Guess Response

Suppose a processor receives a guess (not necessarily the first guess) j_G from the c-host. At iteration j_p, $j_p \geqslant j_G$, the processor sends back the current convergence sequence header if j_p is part of a convergence sequence. A message is sent immediately after such a j_p is found.

Guess Confirmation/Generation

As described above, each processor responds to a guess j_G by returning a convergence sequence header to the c-host. The c-host either confirms or rejects the guess as soon as every processors' response is received. Letting $j_{\max}$ be the maximal value among these responses, the guess j_G is confirmed if $j_G = j_{\max}$; in this case global convergence is achieved at iteration j_G, and the c-host instructs all processors to stop iterating. If $j_G < j_{\max}$, then j_G is rejected, and the value $j_{\max}$ is sent to all processors as as the next guess.

We can show that the *ACC* identifies an iteration achieving global convergence; furthermore, if the solution cannot drift out of global convergence (even temporarily), then the *ACC* is guaranteed to identify the first iteration achieving global convergence. The first claim is proven by contradiction. Suppose that *ACC* confirms iteration j_A as a point of global convergence, but that some processor P has a nonconverged subdomain at iteration j_A. By the confirmation process described above, j_A is the maximum header value in response to a guess j_G, and $j_A = j_G$. Consider P's response to this guess, at (say) iteration j_p. Clearly $j_p \geqslant j_G$, since P does not respond until it has iterated at least to j_G and is presumptively converged. Furthermore, $j_p > j_G$ since P's subdomain is non-converged at iteration $j_A = j_G$. But j_A is the maximum among all responses to j_G, so $j_p \leqslant j_G$, a contradiction. We also show that *ACC* finds the first globally converged iteration if the solution never diverges from global convergence. Suppose that global convergence is achieved first at iteration j_f, and that the *ACC* first detects global convergence at j_A. For the sake of contradiction, suppose that $j_A \neq j_f$. Since *ACC* does detect global convergence, we must have $j_A \geqslant j_f$; for the sake of contradiction, suppose that $j_A > j_f$. j_A is a convergence sequence header from some processor P; thus P's subdomain was *not* converged at iteration $j_A - 1 \geqslant j_f$. This is a contradiction, since we have assumed that global convergence at j_f implies convergence in P for all iterations $j \geqslant j_f$. Thus $j_A = j_f$, showing that *ACC* finds the first globally converged iteration.

The *ACC* scheme is asynchronous in that the processors never synchronize waiting for convergence information and in that the processors do not necessarily send the messages to the c-host at the end of the same iteration. Unlike asynchronous chaotic methods, we still presume that the processors synchronize with their neighbors at each iteration. The communication costs required by the *ACC* method are quite small. The *ACC* requires substantially fewer messages than standard convergence checking; furthermore, the communication may be overlapped with computation. From this aspect, the *ACC* is clearly superior to standard convergence checking. However, the *ACC* does incur two additional computational costs. The minimal cost is execution of the *ACC* logic; the second cost occurs because each processor continues to iterate until it is told to stop. Thus each processor "overshoots", doing slightly more computation even after global convergence is achieved. In practice, neither of these costs proved to be significant; the improvement over standard convergence checking is substantial, as discussed in section 5.

Further improvements in the *ACC* scheme are possible. With a cube of large dimension, we can distribute the c-host function by forming clusters of smaller cubes; *ACC* is then applied locally at a cluster. A cluster reaching "global" convergence is logically equivalent to presumptive convergence in a subdomain; a central c-host would determine global convergence using ACC at a global level. Another improvement is achieved by checking a subdomain's convergence only at selected iterations. The computational cost of checking convergence is saved, at the risk of doing more iterations than are required. In a similar vein, the scheme discussed in the next section formally *schedules* convergence checks and balances the benefits of skipping checks with its risks.

4.3. Maximized Expected Work

We now consider a second method of reducing the number of convergence checks. This method employs a statistical methodology to *schedule* convergence tests at critical iterations. Upon the completion of a scheduled test, the next convergence test is scheduled on the basis of the cost of testing convergence and the costs of scheduling the next test "too far" in the future after convergence has been achieved. The iteration chosen is the one maximizing the "expected work" per unit time and is thus dubbed the MEW method.

The MEW method entails a certain amount of mathematical formalism. First, we define the *ith* iteration error estimate E_i:

$$E_i = \left|\left| X_i - X_{i-1} \right|\right|_{\infty}.$$

We model the convergence behavior of an iterative method by assuming that

$$E_n \leqslant E_1 \cdot e^{-\lambda \cdot n}. \tag{1}$$

We say that the solution has converged at iteration n if $E_n \leqslant \epsilon$ for our tolerance ϵ. The key issue in this formulation is the estimation of λ. If the exact value of λ were known, then the first converged iteration is found by solving for n in the equation $\epsilon = E_1 \cdot e^{-\lambda \cdot n}$. Since the exact value of λ is not known, it must be estimated.

Our treatment of λ is Bayesian (the reader unfamiliar with Bayesian estimation can consult [SCHM69] or any standard statistical text). We view a convergence test as a statistical observation of λ. The *observation* of λ created by calculating E_j is derived from relation (1):

$$\hat{\lambda} = \frac{1}{j} \cdot \Big(\ln(E_j) - \ln(E_1) \Big) \,. \tag{2}$$

We suppose that an observed value of λ is a normal random variable $N(\lambda,\sigma_s^2)$. λ here is the true unknown convergence rate, and σ_s^2 is a sampling variance which we will also estimate. We furthermore suppose that we have some prior knowledge of what λ might be. This prior knowledge is encoded with a normal (prior) probability distribution $N(\lambda_{pr},\sigma_{pr}^2)$ describing the likelihood of λ taking any particular value. Given the parameters λ_{pr}, σ_{pr}^2 and an observation $\hat{\lambda}$, Bayes' Theorem says that the posterior distribution of λ is normal $N(\lambda_{pt},\sigma_{pt}^2)$ where

$$\lambda_{pt} = \left(\frac{\sigma_{pr}^2}{\sigma_{pr}^2 + \sigma_s^2} \right) \cdot \hat{\lambda} + \left(\frac{\sigma_s^2}{\sigma_{pr}^2 + \sigma_s^2} \right) \cdot \lambda_{pr}$$

and

$$\sigma_{pt}^2 = \frac{\sigma_{pr}^2 \cdot \sigma_s^2}{\sigma_{pr}^2 + \sigma_s^2} .$$

The posterior distribution incorporates our prior knowledge of λ with the additional information afforded by $\hat{\lambda}$. The scheduling of our next convergence test (presuming $E_j > \epsilon$) depends in part on this distribution.

We next examine the mechanics of our convergence test scheduling, temporarily deferring discussion of prior determination and the estimation of σ_s^2. Suppose we have a prior distribution of λ, and we make a convergence test at iteration j. We then calculate λ_{pt} and σ_{pt}^2 as described. From iteration j, we view the probable future behavior of convergence at iteration $j + d$ *as though* we are at the first iteration. That is, we presume that the convergence model for iterations $j + d$, $d > 0$, is

$$E_{j+d} \leqslant E_j \cdot e^{-\lambda \cdot d}.$$

It can be shown that sensitivity to measurement errors in λ is reduced by using this modification. The probability of not observing convergence at iteration $j + d$ is easily seen to be identical to the probability that λ is less than the threshold $T_j(d)$, where

$$T_j(d) = \frac{1}{d} \cdot \ln\left(\frac{E_j}{\epsilon} \right).$$

Appealing to the normal structure of the posterior distribution, we thus have

$$Prob\{\lambda < T_j(d)\} = \Phi\left(\frac{T_j(d) - \lambda_{pt}}{\sigma_{pt}} \right)$$

where Φ is the standard normal cumulative distribution function. We use the probability above as an estimate of the probability that we will not have converged by iteration $j + d$.

We can now detail the scheduling decision. Let I be the delay cost of performing a convergence test, and let D be the time required to perform one iteration without a convergence test. If we schedule the next convergence test at iteration $j + d$, the total time required to do d iterations and perform the test is $d \cdot D + I$. Then the average *required* number of iterations per unit time achieved by this decision is

$$\frac{\sum_{i=1}^{d} \Phi\left(\dfrac{T_j(d) - \lambda_{pi}}{\sigma_{pr}} \right)}{d \cdot D + I}. \qquad (3)$$

We find the $d = d_{\max}$ which maximizes the expression above and schedule the next convergence test at iteration $j + d_{\max}$. Maximization of expression (3) balances the cost of testing convergence with the cost and uncertainity of doing more iterations than are required. As a function of d, expression (3) has at most one local maximum which is easily found. The convergence test scheduling decision at iteration $j + d_{\max}$ uses the $N(\lambda_{pt},\sigma_{pt}^2)$ distribution as its prior.

The mechanics of our scheduling policy illustrate how we deal with uncertainty about λ. This policy is also dependent on quantities we now discuss: the sampling variance σ_s^2 and the initial prior distribution of λ. There are situations where significant prior knowledge of convergence behavior is known. The model problem is time-dependent, so we need to solve the equations at each of a number of time steps. The convergence behavior of the method at time steps in the near past is a good predictor of the convergence behavior in the near future. In fact, after the first three time steps, we were able to construct very reasonable priors *before* beginning a time step's iterations. We simply used the last time step's effective λ as the prior mean (found by solving $\epsilon = E_1 \cdot e^{-\lambda \cdot N}$ for λ, knowing that exactly N iterations were required); we used the sample variance of the last three time steps' effective λ's for our prior variance. The Bayesian formulation can also exploit user experience with the solution method's convergence; this experience could be summarized as a prior distribution.

At the beginning of the computation we might well presume no prior knowledge of the convergence behavior. We gain some insight into this behavior by testing for convergence after each of the first few iterations. As before, a convergence test is viewed as an observation of λ. If we could assume that each observation is independent of any other, we could then use the sample mean as the initial prior mean λ_{pr}, and the sample variance as both the sampling variance σ_s^2 and the prior variance σ_{pr}^2. However, successive errors E_i and E_{i+1} are not independent. Their correlation leads to a biased estimation of λ_{pr} and the underestimation of σ_{pr}^2. To compensate for this conflict of mathematical assumption and practical reality, we devised the *constrained projection* rule. This rule states that if convergence is tested at iteration j, the next convergence test must be scheduled before iteration $2 \cdot j + 1$. This rule forces additional convergence tests at the beginning of the computation, and affords protection from wildly optimistic scheduling decisions. We thus used the sample statistics to construct our prior information, but then protected ourselves from a bad prior with the constrained projection rule. In our experience, this rule was effectively invoked only at the beginning of the computation. After this startup period, our underlying assumption of independence between observations of λ is better satisfied, and the statistics are more accurate. The variance σ_s^2 is then reasonably taken to be the sample variance of the observed λ's to date.

The *MEW* method is an excellent vehicle for encapsulating both our prior knowledge, and the knowledge gained about convergence as the solution progresses. Furthermore, it is simple to program, and its sensitivity to changes in the problem or problem distribution across processors lies only in the parameters I and D. The empirical study described in the next section shows that *MEW* is quite effective in reducing convergence checking delay.

5. Convergence Checking Performance

The effects of employing the three different convergence schemes on the algorithm performance (on a 128 by 128 grid) is depicted in Figure 6. The model problem was solved on a cube with 1, 2, 4, 8, 16, 32, and 64 processors; our implementation of *ACC* dedicated one node to the c-host function, so that we did not test this method with 64 processors (the maximum cube size on our system). In each test the domain was subdivided into strips of equal size, assigned one to each processor so that adjacent strips were mapped onto adjacent processors. In Figure 6, the measured performance in terms of time-steps advanced per second is plotted as a function of the number of processors. Note that the best one can achieve is a line with a slope of one. Figure 6 illustrates that the performance of all three convergence schemes degrades with an increasing number of processors. The standard convergence scheme checking scheme depicted here involves the use of the tree method of global convergence checking used each iteration. The tree method of convergence checking was utilized as scheduled in the *MEW* scheme. Results obtained for the cascade method of convergence checking are quite similar to those depicted here. The standard convergence scheme's deterioration is rapid, while the other two schemes degrade gradually. The difference between the *MEW* and *ACC* schemes is not significant. Both have essentially the same communication costs, as very few *ACC* guesses and *MEW* checks were required on each time-step. The *ACC* overshoot after global convergence was between three to five percent. The *MEW* scheme has a lower computation cost than ACC because it skips local convergence checking on some iterations altogether. The ACC scheme checks the state of each subdomain every iteration, although we observed a ten percent improvement by checking every other iteration. The difference in local convergence checking accounts for the slight difference in the two schemes' performance. All three methods showed similar effects on the algorithm performance when the grid size was changed to 64 by 64 or 256 by 256.

6. Reduction of Communication Delays Resulting from Windows

Reduction of communication delays in the iterative solution of the linear equations produced by a discretization of a time dependent problem can be effected by iteratively solving more than one timestep during a given stage of the computations. The boundary variable values from more than one timestep can be sent in a packet, thus reducing the effect of the message transmission overhead.

In an iterative solution of a time dependent problem, one generally iterates over each timestep individually until convergence at that timestep is detected. One may instead iterate over more than one timestep during each stage of a computation. Assume that we are iterating over variables at timesteps t_1,.., t_n. During each iterative sweep, all variables are updated in each of the timesteps included in the window. Following the sweep, the right hand sides of equations at timesteps t_2,..,t_n are updated to account for changes in the variable values at the earlier timesteps. Convergence is checked at t_1, and when global convergence is detected at this time the window shifts up one timestep to encompass $t_2,...,t_{n+1}$. It has been shown [SALTZ85], that the asymptotic rate of convergence of SOR implemented with windows is equivalent to that of SOR applied to each timestep individually. In other words the total number of sweeps over each timestep required for a given degree of error reduction does not, in an asymptotic sense, change with the window size. For finite difference equations in which time discretization is by Crank Nicholson or Backwards Euler methods, the operation count for each sweep over a timestep is minimally effected by the use of windows. In practice, the computational work required to solve a problem increases quite gradually with window size.

Communication costs are reduced in two ways when one iterates over windows of timesteps in a time dependent problem. The first is the previously stated fact that fewer but larger packets need be sent for the transmission of boundary variable data. Because convergence need be checked only at the lowest timestep in a window, the number of global communications required to check for convergence is reduced by a factor of 1/window size as long as the total number of sweeps over each timestep does not change with window size, as is approximately the case for small windows. Thus if the total number of sweeps over each timestep were independent of window size and the cost per packet were independent of the size of the packet, the overall cost of communication would be reduced by a factor of 1/window size. Finally, some computation time is saved because the computations required for local convergence checking need only be carried out at the lowest timestep in a window.

Algorithm performance is improved by the use of windows of a relatively small size. A number of experiments were carried out to demonstrate this, the results of one set are shown in figure 7. The model problem was solved on a 64 by 64 point domain, and the rate of computation resulting from the use of windows of sizes 1,2 and 3 is shown. A notable improvement in performance is seen in comparing windows of size 1 and 2; a less marked improvement in performance is seen when a window of size 3 was utilized. This result is expected as the computational cost increases as window size is increased, and communication delays can be reduced by no more than a factor of 1/window size.

The use of windows decreases the cost of of communicating boundary variables and communication flags at the cost of increased storage requirements and an increase in the computation required for each timestep. The methods for reducing the costs of global convergence checking described here have minimal costs and storage requirements. It is hence natural that the methods should be used together. In figure 8 are depicted results for the model problem solved on a 128 by 128 point domain using windows of sizes 1 and 2 with Bayesian MEW convergence test scheduling and using windows of sizes 1 and 2 with standard convergence testing. The use of both windowing and Bayesian convergence test scheduling together led to additional improvements in performance over that obtained through the use of either separately.

7. Conclusion

The sources of communication delays in the solution of a model problem by red-black SOR have been identified and their relative contribution quantified under a number of circumstances. A number of methods have been proposed and tested to reduce the effects of each of these sources of communication delay.

In a message passing multiprocessor whose overhead for communication β is substantially larger than the bandwidth α, there is considerable motivation to reduce the number of messages that must be sent. In section 3 the discussion and experimental tests on domain partitioning indicate that improvements in performance may in many circumstances be obtained by reducing the number of messages that must be sent even when the total number of bytes to be sent must increase. In section 5 it is seen that through the use of windows the number of messages that must be sent is decreased. In this case the trade-off is a small increase in the cost of computation, and again an overall performance improvement is noted.

Checking global convergence when message overhead is high is quite costly, and methods were described in section 3 to perform tests efficiently and to reduce the number of such tests required. It is demonstrated in section 5 that the effects of using windows and of using methods that reduce the effects of convergence costs can have a complementary effect of performance.

It should be noted that convergence testing is the only non-local communication in red-black SOR. Both the ACC and the MEW method greatly reduce global communication and are consequently expected to be quite useful in architectures where the interprocessor connectivity is more restricted, such as a ring or a mesh multiprocessor. In these architectures combining results from all processors may be quite expensive for large machines.

Acknowledgements

We would like to thank the Mathematics and Statistics Research Section at Oak Ridge National Laboratory for generously granting us use of their Intel Personal Super-Computer. We are particularly indebted to Mike Heath and Tom Dunigan for assistance during the course of this work. We are grateful to Bob Voigt for constant support and encouragement.

References

[FOX84] Fox, G. C., 1984, "Concurrent Processing for Scientific Calculations," COMPCON84, 28th IEEE Computer Society International Conference, pp. 70-73.

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

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

[RATT85] Rattner, J., 1985, "Concurrent Processing: A New Direction in Scientific Computing," AFIPS, Conference Proceedings 1985 National Computer Conference, Vol. 54, pp. 157-166.

[SAAD85] Saad, Y. and M. H. Schultz, 1985, "Topological Properties of Hypercubes," Research Report, Yale University, YALEU/DCS/RR-389.

[SALT85a] Saltz, J. H., 1985, "Parallel and Adaptive Algorithms in Scientific and Medical Computing: Robust Method for the Solution of Partial Differential Equations on Multiprocessor Machines," Ph.D. Thesis, Duke University.

[SALT85b] Saltz, J. H. and V. K. Naik, 1985, "Towards Developing Robust Algorithms for Solving Partial Differential Equations on MIMD Machines," ICASE Report No. 85-39, NASA Contractor Report No. 177979.

[SCHM69] Schmitt, S. A., 1969, An Elementary Introduction to Bayesian Statistics, Addison-Wesley, Reading, MA.

[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, Iterative Solutions of Large Linear Systems, Academic Press, NY.

Figure 1. Effect of Domain Size on Parallel Efficiency

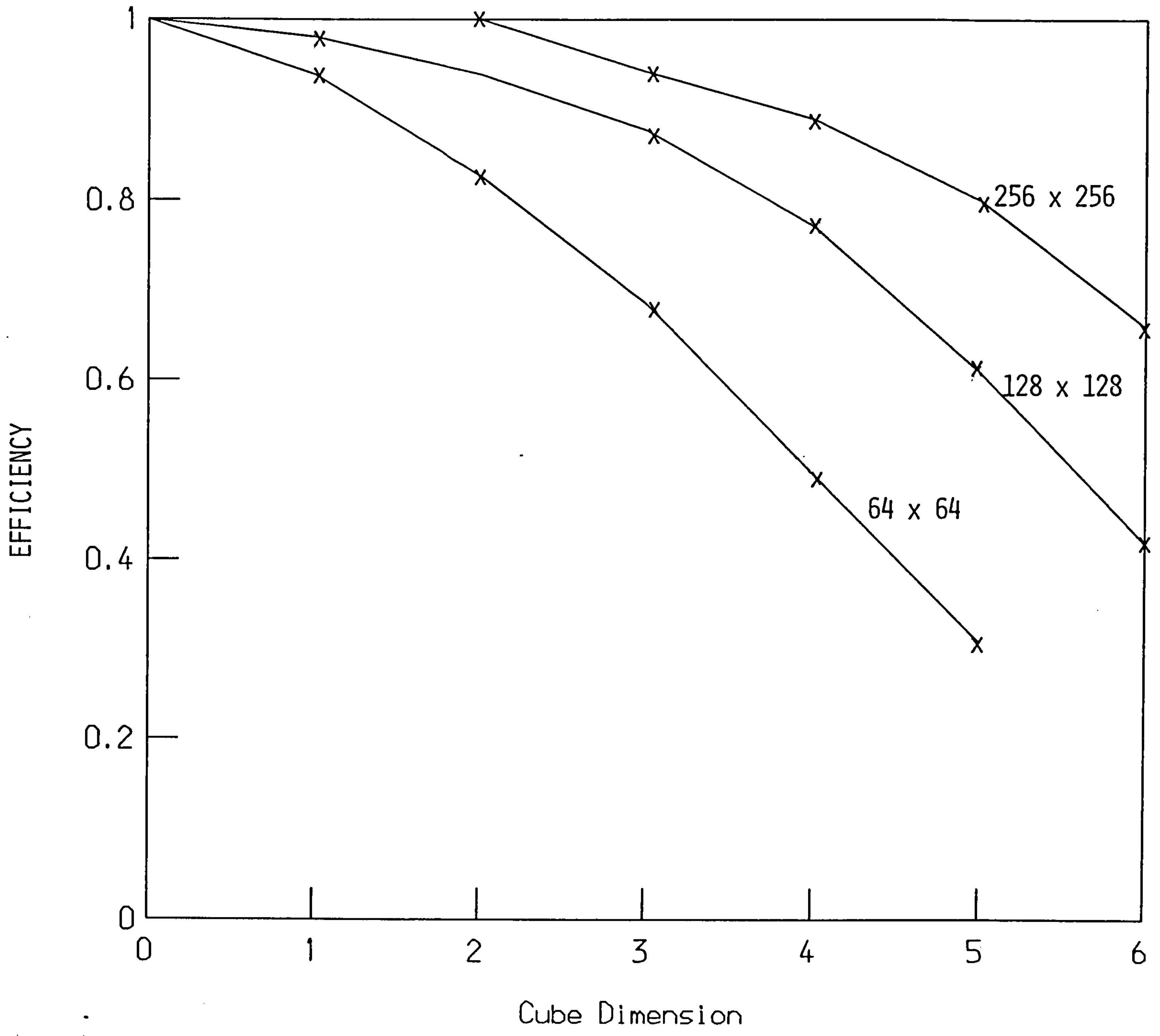


FIGURE 2. EFFECT OF SUBDOMAIN SHAPE ON COMMUNICATION COST PER ITERATION.

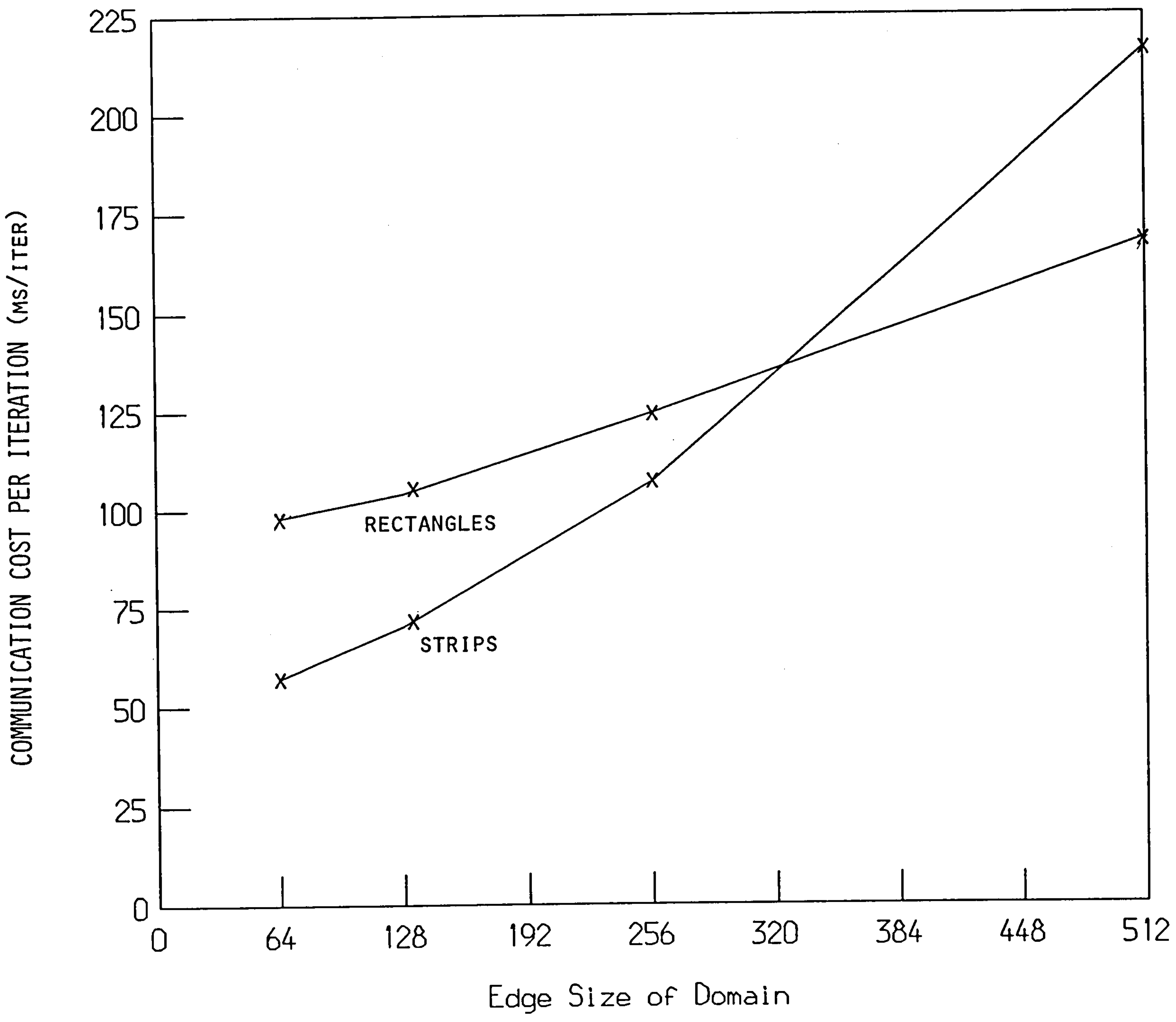
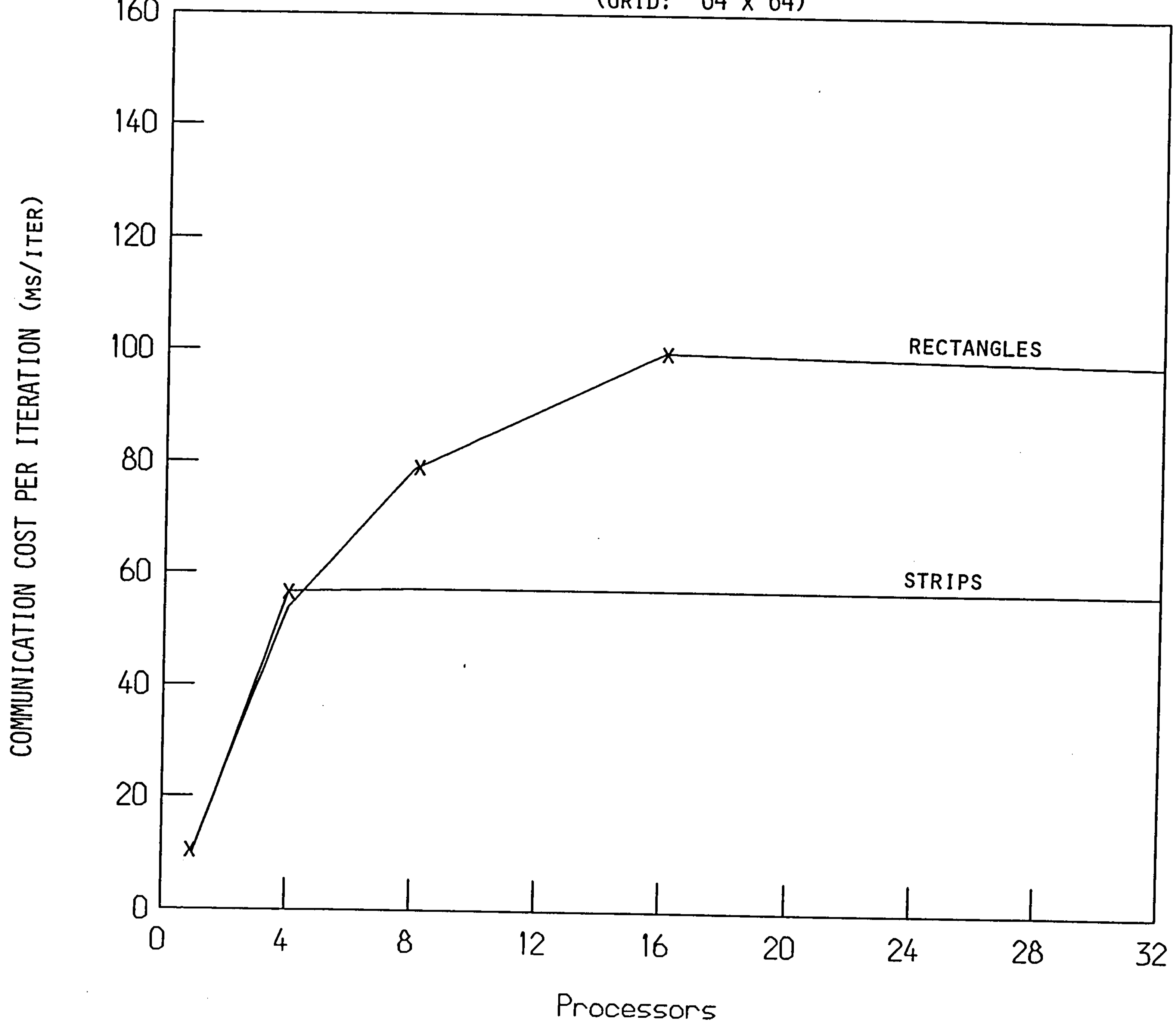
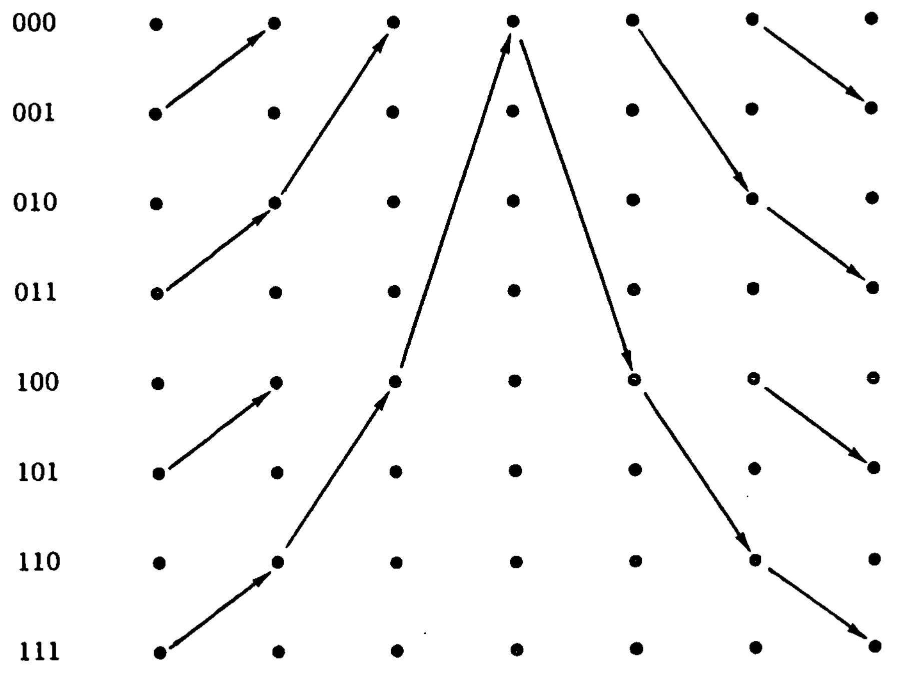


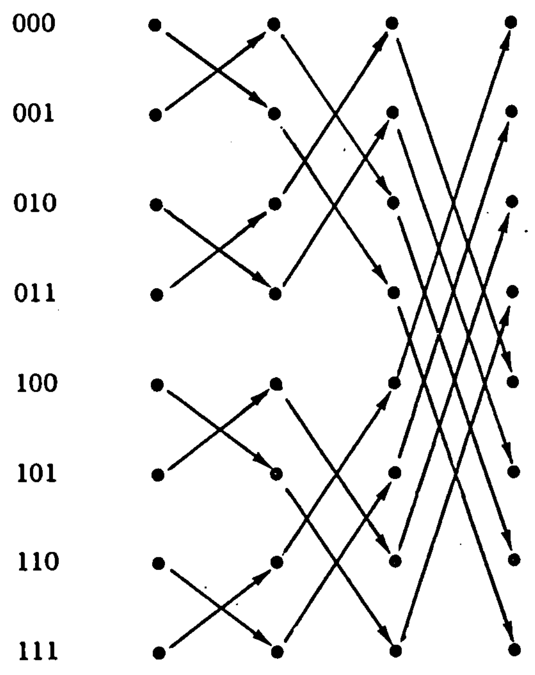
FIGURE 3. EFFECT OF SUBDOMAIN SHAPE ON BOUNDARY VARIABLE COMMUNICATION COST (GRID: 64 x 64)





Tree Convergence Checking

Hypercube Dimension 3



Cascade Convergence Checking

Hypercube Dimension 3

Figure 4.

Figure 5. Comparison of Boundary Variable Communication cost, Convergence Checking Cost and Computation Cost

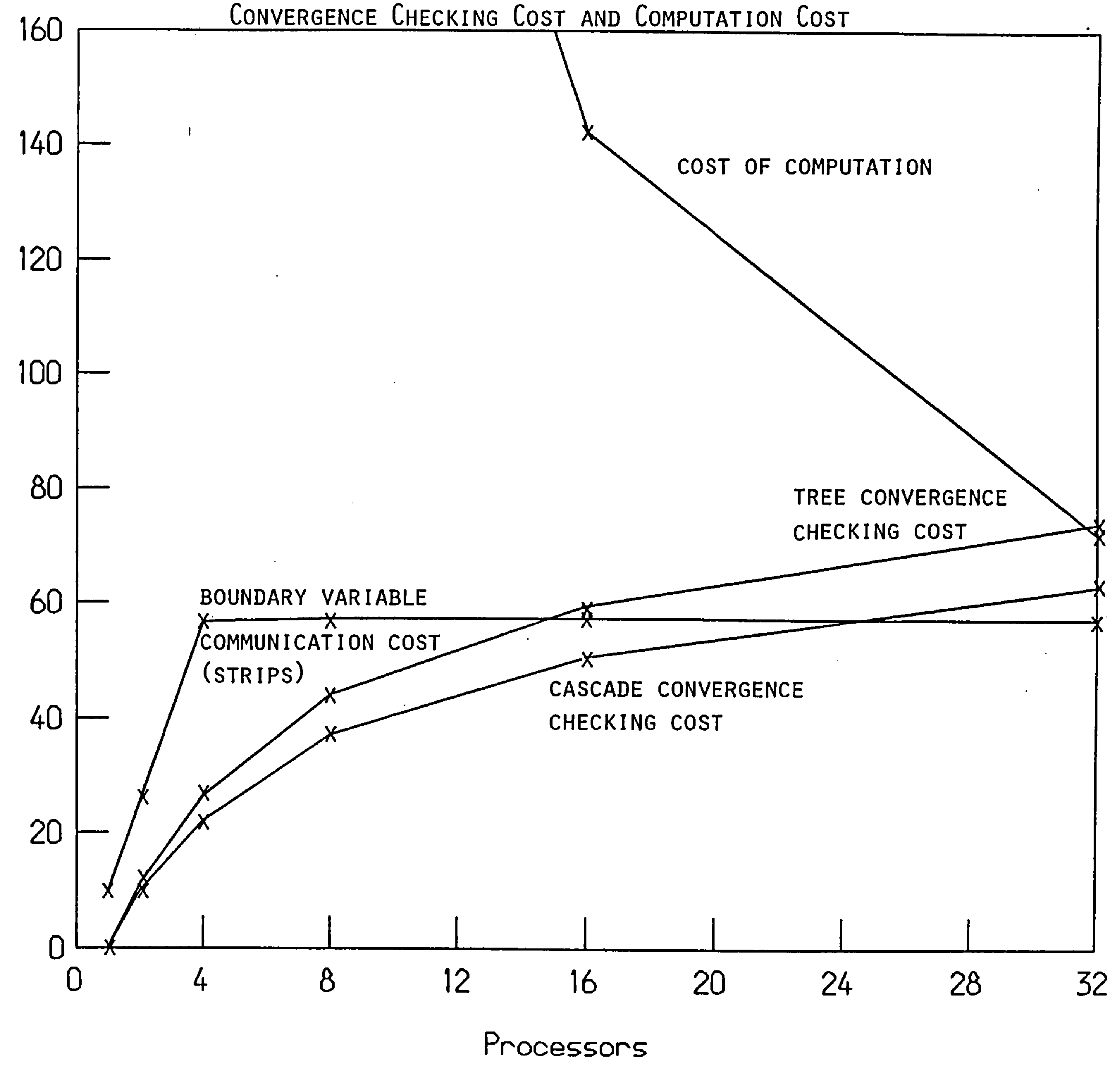


FIGURE 6. EFFECT OF CONVERGENCE CHECKING SCHEMES ON ALGORITHM PERFORMANCE. (GRID SIZE: 128 x 128, SUBDOMAIN SHAPE: STRIPS)

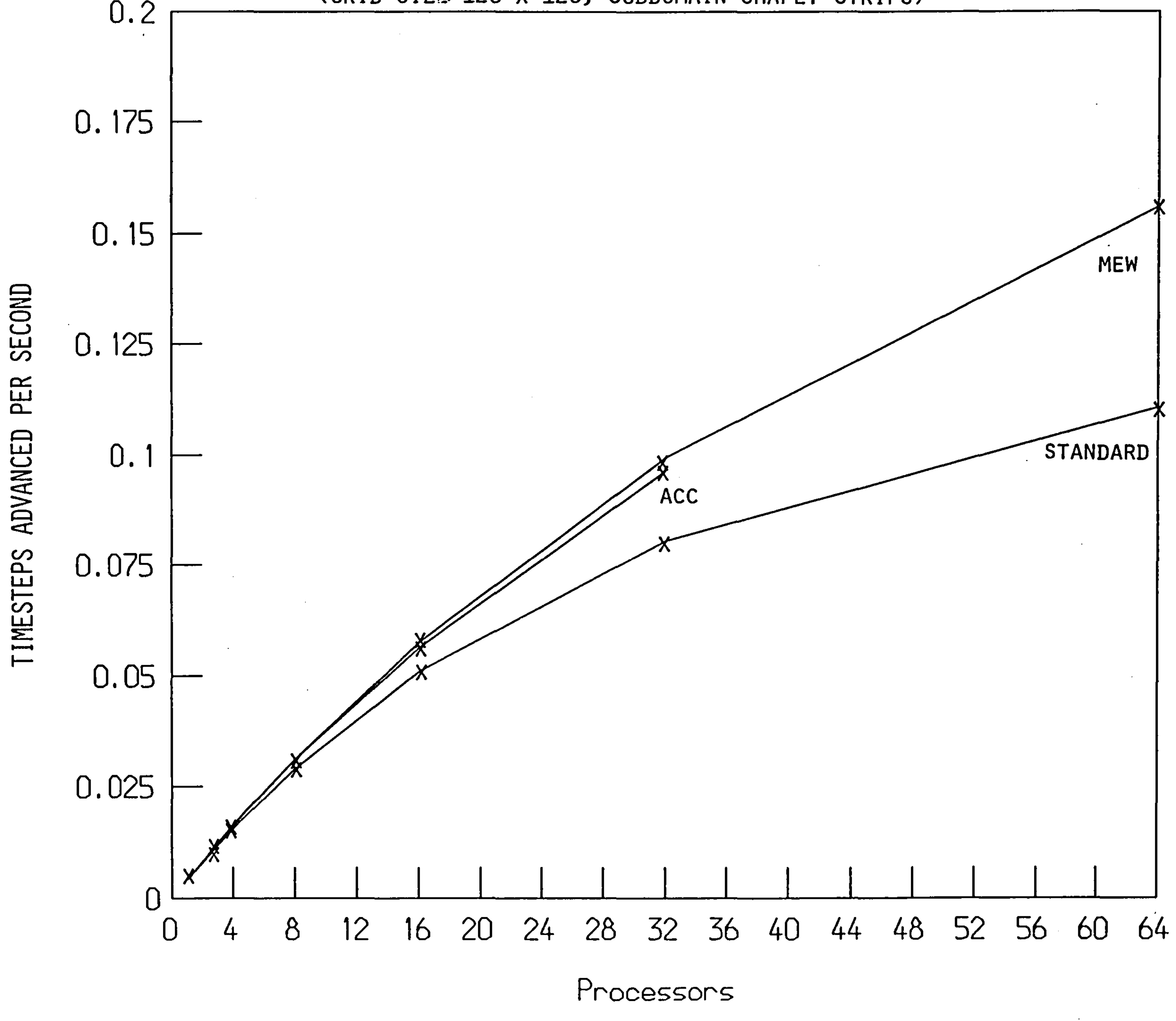


FIGURE 7. EFFECT OF WINDOWS ON PERFORMANCE.
(GRID: 64 X 64, SHAPE: STRIPS, STANDARD CONV. CHECKING)

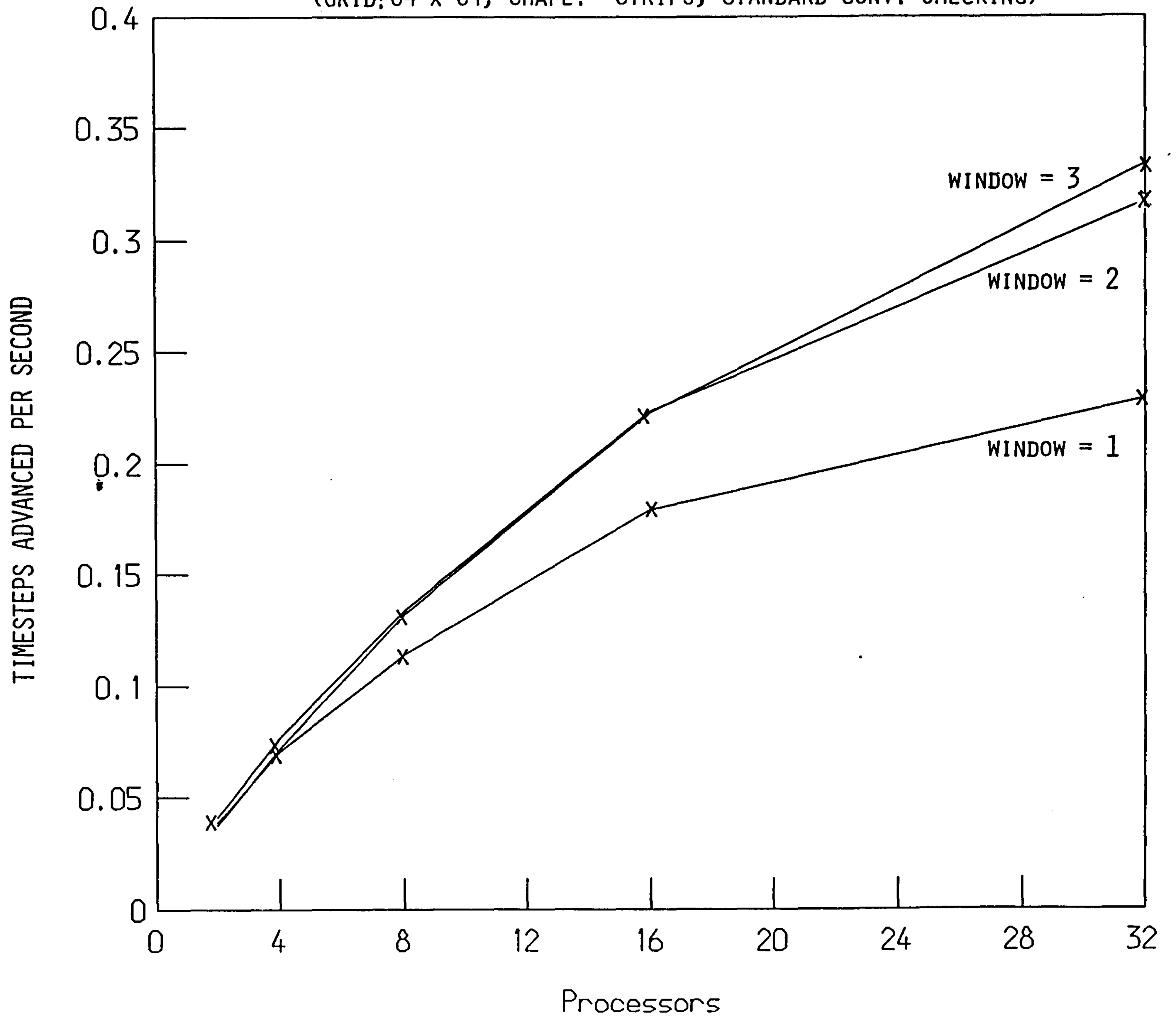
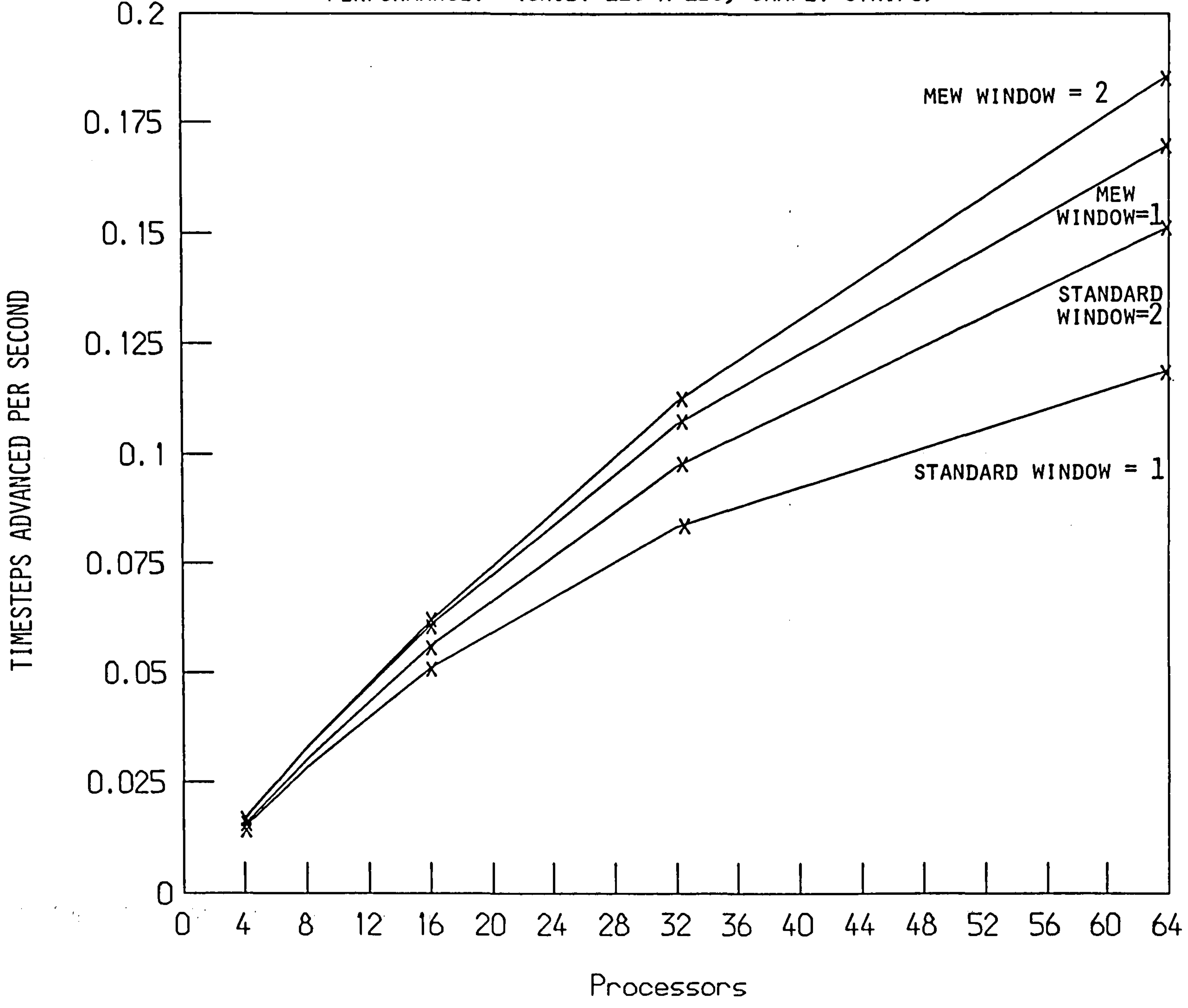


FIGURE 8. EFFECT OF WINDOWS AND OF CONVERGENCE CHECKING SCHEMES ON PERFORMANCE. (GRID: 128 x 128, SHAPE: STRIPS)



Standard Bibliographic Page

1. Report No.
NASA CR-178044, ICASE Report 86-4

2. Government Accession No.

3. Recipient's Catalog No.

4. Title and Subtitle
Reduction of the Effects of the Communication Delays in Scientific Algorithms on Message Passing MIMD Archietectures

5. Report Date
January 1986

6. Performing Organization Code

7. Author(s)
Joel H. Saltz, Vijay K. Naik, and David M. Nicol

8. Performing Organization Report No.
86-4

9. Performing Organization Name and Address
Institute for Computer Applications in Science and Engineering
Mail Stop 132C, NASA Langley Research Center
Hampton, VA 23665-5225

10. Work Unit No.

11. Contract or Grant No.
NAS1-17070; NAS1-18107

12. Sponsoring Agency Name and Address
National Aeronautics and Space Administration
Washington, D.C. 20546

13. Type of Report and Period Covered
Contractor Report

14. Sponsoring Agency Code
505-31-83-01

15. Supplementary Notes
Langley Technical Monitor:
J. C. South
Final Report

Submitted to SIAM J. Sci. Stat. Comput.

16. Abstract

The efficient implementation of algorithms on multiprocessor machines requires that the effects of communication delays be minimized. The effects of these dealys on the performance of a model problem on a hypercube multiprocessor architecture is investigated and methods are developed for increasing algorithm efficiency. The model problem under investigation is the solution by red-black Successive Over Relaxation [YOUN71] of the heat equation; most of the techniques to be described here also apply equally well to the solution of elliptic partial differential equations by red-black or multicolor SOR methods.

This paper identifies methods for reducing communication traffic and overhead on a multiprocessor, and reports the results of testing these methods on the Intel iPSC Hypercube. We examine methods for partitioning a problem's domain across processors, for reducing communication traffic during a global convergence check, for reducing the number of global convergence checks employed during an iteration, and for concurrently iterating on multiple time-steps in a time-dependent problem. Our empirical results show that use of these models can markedly reduce a numerical problem's execution time.

17. Key Words (Suggested by Authors(s))
message passing, MIMD,
communication delays,
parabolic equations, SOR

18. Distribution Statement
61 - Computing Program & Software
64 - Numerical Analysis

Unclassified - unlimited

19. Security Classif.(of this report)
Unclassified

20. Security Classif.(of this page)
Unclassified

21. No. of Pages
37

22. Price
A03

For sale by the National Technical Information Service, Springfield, Virginia 22161

NASA Langley Form 63 (June 1985)