Journal of Parallel and Distributed Computing **58**, 446–465 (1999)
Article ID jpdc.1999.1549, available online at http://www.idealibrary.com on IDEAL®

Timing Models and Local Stopping Criteria for Asynchronous Iterative Algorithms

Kostas Blathras, Daniel B. Szyld[1], and Yuan Shi

Temple University, Philadelphia, Pennsylvania 19122
E-mail: kostas@thunder.ocis.temple.edu, szyld@math.temple.edu, shi@cis.temple.edu

Received May 12, 1997; accepted March 29, 1999

Asynchronous iterative algorithms can reduce much of the data dependencies associated with synchronization barriers. The reported study investigates the potentials of asynchronous iterative algorithms by quantifying the critical parallel processing factors. Specifically, a time complexity-based analysis method is used to understand the inherent interdependencies between computing and communication overheads for the parallel asynchronous algorithm. The results show, not only that the computational experiments closely match the analytical results, but also that the use of asynchronous iterative algorithms can be beneficial for a vast number of parallel processing environments. The choice of local stopping criteria that is critically important to the overall system performance is investigated in depth.

Key Words: timing models; asynchronous algorithms; parallel iterative methods.

1. INTRODUCTION

Parallel asynchronous block methods for the solution of a linear algebraic system of the form

$$A \cdot \mathbf{x} = \mathbf{b}, \tag{1}$$

where $A = (a_{ij})$ is a large sparse $N \times N$ matrix have been studied by many authors; see, e.g. [6, 9, 14] and the references given therein.

These methods are usually some generalization of the classical block Jacobi method (see, e.g., [25]), where the matrix $\mathbf{A}$ is partitioned into $p \times p$ blocks

$$\begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1p} \\ A_{21} & A_{22} & \cdots & A_{2p} \\ \vdots & \vdots & \cdots & \vdots \\ A_{p1} & A_{p2} & \cdots & A_{pp} \end{bmatrix} \tag{2}$$

[1] Supported by the National Science Foundation Grant DMS-9625865.

0743-7315/99 $30.00

with the diagonal blocks A_{ll} being square nonsingular of order n_l, $l=1, ..., p$, $\sum_{l=1}^{p} n_l = N$, and the vectors $\mathbf{x}$ and b are partitioned conformally.

ALGORITHM 1 (Block Jacobi). Given an initial approximation of the vector $(\mathbf{x}(0))^{\mathrm{T}} = [(\mathbf{x}_1(0))^{\mathrm{T}}, ..., (\mathbf{x}_p(0))^{\mathrm{T}}]$,

For $t=1, 2,...,$ until convergence
For $l=1$ to p, solve

$$A_{ll}\mathbf{x}_l(t+1) = \mathbf{b}_l - \sum_{j=1, j\neq l}^{p} A_{lj}\mathbf{x}_j(t). \tag{3}$$

The entire linear system can be solved in parallel by p processors, each processor computing one subvector $\mathbf{x}_l(t)$ and the iteration vector at each step is

$$(\mathbf{x}(t))^{\mathrm{T}} = [(\mathbf{x}_1(t))^{\mathrm{T}}, (\mathbf{x}_2(t))^{\mathrm{T}}, ..., (\mathbf{x}_p(t))^{\mathrm{T}}].$$

This algorithm exchanges data for each value of t, and has at that point a synchronization barrier.

In an asynchronous version of (3), each processor would compute $\mathbf{x}_l$ using the most recent information available from the other processors, i.e., values of $\mathbf{x}_l$ which may be older than $\mathbf{x}_j(t)$, say possibly $\mathbf{x}_j(t-k)$, where k would depend on l and j; see further, Section 2, where we use an equivalent notation. In other words, asynchronous iterative algorithms do not require exchange of the most recent values. The convergence of (3) would naturally be delayed by the use of less recent values of $\mathbf{x}_j$, but since no idle time for synchronization is necessary and less overall communication takes place, asynchronous algorithms have the potential of outperforming standard synchronous iterative methods. We should mention here that, under certain hypothesis, asynchronous iterative methods are guaranteed to converge to the solution of the linear system (1); see, e.g., [6, 9, 15].

In practice, each subsystem (3) is solved by another iterative method such as Gauss–Seidel, SOR, and in some cases by conjugate gradient-type methods. These classes of methods are called two-stage iterative methods; see, e.g., [13, 19], and the extensive references given therein.

One of the critical questions about these two-stage methods is what criterion to use to stop the inner iterations in each processor; see, e.g., [15, 21]. In this paper, we compare two widely used stopping criteria: one based on the size of the (inner) residuals, and the other on a fixed number of inner iterations. We report that a fixed number of inner iterations is a better choice for the architecture considered in this paper; see Sections 4.3 and 6. For studies on stopping criteria in different types of architectures, see [5, 23].

Another question which we address is load balancing. Should the partition of the domain be fixed by the number of available processors (static block allocation) or do we gain time savings by dividing the computational domain dynamically to account for the different computational complexity of the tasks in each region? To answer this question we developed a timing model. Using the timing model, we

conclude that when the communication network is slow, as is the case of the hardware considered in this paper, the static block allocation is preferred. It follows as well that a dynamic block allocation would be preferable for faster networks; see Sections 4.4 and 6.

The analysis of asynchronous algorithms can be used directly on system (1) or they can be used as fast parallel preconditioners for conjugate gradients or other Krylov subspace methods; see, e.g. [2, 4].

Other authors have published experiments showing that parallel asynchronous iterative methods can be very beneficial, especially on problems with complicated geometry [15, 21]. Here we concentrate on the inner stopping criteria and the timing models to predict parallel performance while at the same time we illustrate how these methods can be implemented on inexpensive clusters of workstations, even personal computers connected with 10 Mbps shared medium Ethernet. We use a passive object programming system named Synergy [8] that provides tuple space as communication and synchronization mechanisms. The choice of this programming tool is further discussed in Section 4.1.

The overall organization of this paper is as follows. In Section 2, we describe the applicable class of asynchronous iterative algorithms. In Section 3 we describe the computational test problem used in this study. In Section 4 we detail the asynchronous parallel program design and implementation. In Section 5 we present timing analysis for evaluation of various implementation alternatives. In Section 6, we present computational results. Conclusions are found in Section 7.

2. BLOCK ASYNCHRONOUS ITERATIVE ALGORITHMS

For the asynchronous block Jacobi method, unlike in Algorithm 1, the processors are allowed to start the computation of the next iteration of the block without waiting for the simultaneous completion of the same iteration of other components. In other words, components of $\mathbf{x}$ are updated using a vector which is made of block components of previous, not necessarily the latest, iterations. As in the standard references for asynchronous algorithms, such as [6], the iteration subscript is increased every time any (block) component of the iteration vector is computed. Thus, one defines the sets $J_t \subseteq \{1, 2, ..., p\}$, $t = 1, 2, ...$, by $l \in J$ if the lth block component of the iteration vector is computed at the tth step. Thus the asynchronous block Jacobi method can be described as

$$\mathbf{x}_l(t+1) = \begin{cases} \mathbf{x}_l(t), & \text{if } l \notin J_t \\ \text{approximate solution of } A_{ll}\mathbf{x}_l(t+1) = \mathbf{b}_l - \sum_{j=1, j\neq l}^{q} A_{l_j}\mathbf{x}_j(r(j,t)), & \text{if } l \in J_t. \end{cases}$$

The term $r(l, t)$ is used to denote the iteration number of the lth block component being used in the computation of any block component in the tth iteration, i.e. the iteration number of the jth block component available at the beginning of the computation of $\mathbf{x}_l(t)$, if $l \in J_t$. We always assume that the terms $r(l, t)$ of our asynchronous iterative algorithms satisfy the minimal criteria as described in [15] and other references therein:

1. $r(l, t) < t$ for all $l = 1, ..., p$; $t = 1, 2, ...$;
2. $\lim_{t \to \infty} r(l, t) = \infty$ for all $l = 1, ..., p$;
3. the set $\{t \mid l \in J_t\}$ is unbounded for all $l = 1, ..., p$.

Convergence of the asynchronous block Jacobi algorithm follows from results in [15]; see also [9].

3. THE COMPUTATIONAL TEST PROBLEM

We use a magnetic field simulation problem as the test case for it represents typical physical simulation applications and it has a nonuniform geometry that requires load balancing when processed in parallel. The processing environment is a cluster of shared 10 mbps Ethernet connected DEC/Alpha workstations. Our simulation is to determine the magnetic field in a region outside a permanent magnet [10]; i.e., to solve for ϕ satisfying

$$\nabla^2 \phi = \text{div } \mathbf{M} \tag{4}$$

where div $\mathbf{M}$ is the volume magnetic charge density.

For simplicity we chose to solve the two-dimensional form of the problem; see Fig. 1. The magnet is assumed to be uniformly magnetized; i.e., $\mathbf{M}(\mathbf{r})$ is a constant vector. Thus, Eq. (4) is reduced to the Laplacian everywhere except for the surface of the magnet:

$$\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} = 0. \tag{5}$$

The rectangular domain in Fig. 1 is discretized using a uniform grid with h horizontal and vertical spaces. Thus, the grid points are labeled (i, j), $i, j = 1, 2, 3, ..., n$, and we denote by $\phi_{i,j}$ the value of the function ϕ at the point of (i, j). The discretization of (5) using centered differences is then

$$\frac{1}{h^2}(\phi_{i+1,j} + \phi_{i-1,j} + \phi_{i,j-1} + \phi_{i,j+1} - 4\phi_{i,j}) = 0. \tag{6}$$

Therefore, the value of the magnetic field on each grid point $\phi_{i,j}$ is a function of its four nearest neighbors; see Fig. 2.

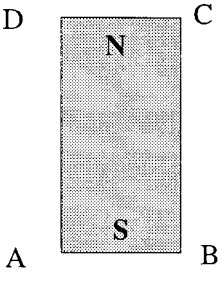

FIG. 1. A rectangular magnet.

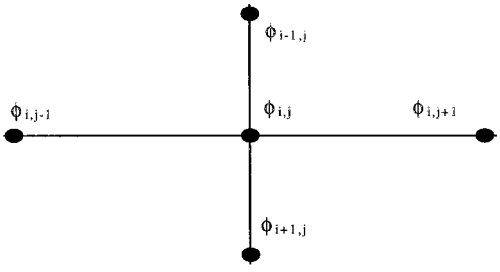

FIG. 2. Grid around point (i, j).

Equation (6) is suitably modified near the boundaries of the rectangular magnet in order to incorporate the discontinuities of ϕ.

4. PARALLEL ALGORITHM DEVELOPMENT

A parallel program is typically more complex than its sequential counterpart. In this section, we first discuss our programming tool choice. We then discuss our partitioning choices and introduce a number of parallel implementation alternatives. Finally, we present an analysis method that can be used to evaluate the parallel algorithms by capturing the inherent interdependencies between the computing and communication overheads.

4.1. Parallel Programming Tool

The choice of programming tool can have a large impact on parallel implementation complexity and processing efficiency. An ideal parallel processing environment for parallel asynchronous iterative algorithms should be totally asynchronous with information *over-write* capabilities. This is because for asynchronous parallel algorithms, the sender and receiver are running independent of each other, and the receiver must obtain the latest information or otherwise introduce artificial synchronization barriers; see [22].

All message passing systems, such as MPI [20] and PVM [7], use bounded buffers. This implies a synchronous semantics, namely the sender should only send when the receiver is there to receive. Moreover, the messages do not over-write each other. If we were to use any message passing system, we had to implement the new messaging semantics on top of the provided messaging channels. This would introduce much programming complexity and processing overhead. Thus, we consider that these systems are not appropriate for programming parallel asynchronous algorithms.

The Synergy system uses passive objects for parallel program communication and synchronization. A passive object is a coarse-grain data structure with a set of predefined operators. An example passive object is a tuple space with three operators: *Read*, *Put*, and *Get*. The semantics of these operators is similar to *rd*, *get*, and *put* operations in the Linda system [1], provided that tuples are uniquely named and first-in-first-out (FIFO) ordered. Writing to the same named tuple

means over-writing the existing tuple's content. This characteristic ideally meets the asynchronous iterative algorithm's requirement. The object passiveness restricts the operators from dynamically creating new objects at runtime. They can only create instances within an object, thus leaving the outset communication topology fixed for each application. This feature was designed to facilitate automatic generation of efficient client/server programs from a fixed application configuration topology. For a more detailed description of Synergy, we refer the reader to [8].

Passive objects embed multithreaded controls under a simple asynchronous programming interface. Thus, there is no explicit process manipulation and synchronization statements in either the sender or the receiver programs. Unless specifically coded, each individual program has a single thread control within its programming space.

4.2. Data Partition Choice

A typical parallel implementation of a block iterative algorithm, such as Block Gauss–Seidel [6, 17], assigns several mesh points to each processor such that each processor only communicates with its four nearest neighbors; see Fig. 3, where there are $p = 9$ processors and $n = 36$.

This intuitive parallel implementation, called *tiles*, has the following drawbacks:

- In comparison with partitioning in *stripes*, the tile partitioning requires less overall interprocessor communication data volume but more communication *sessions* at each iteration, due to more interfaces with the neighbors. Since establishing a communication session between processes needs more time than moving a few thousand bytes of data on a typical interconnection network hardware, for many practical problems the tile partitioning is guaranteed to deliver poor performance when compared to striping. This conclusion was recorded in studies comparing

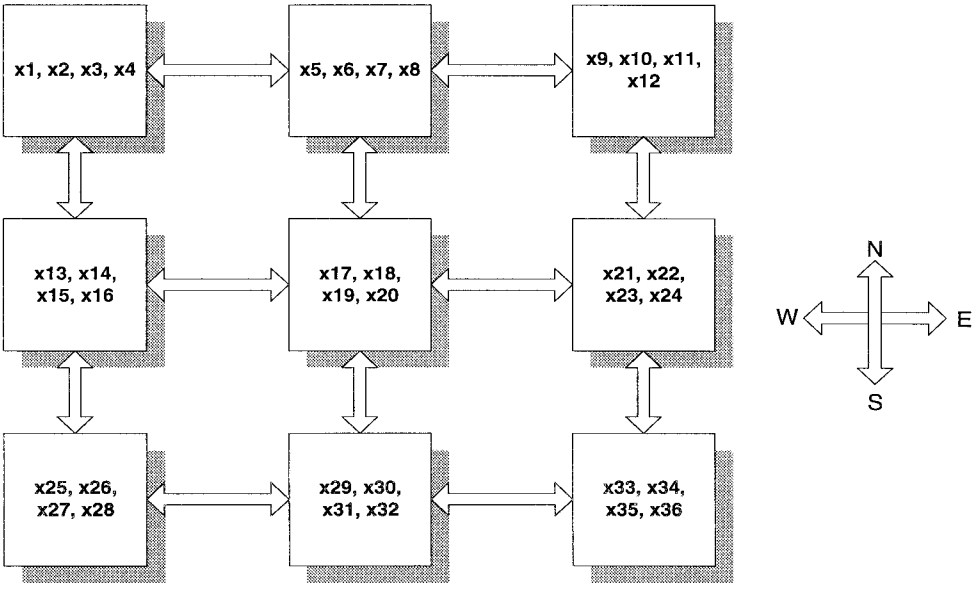

FIG. 3. Typical parallel block-matrix operations.

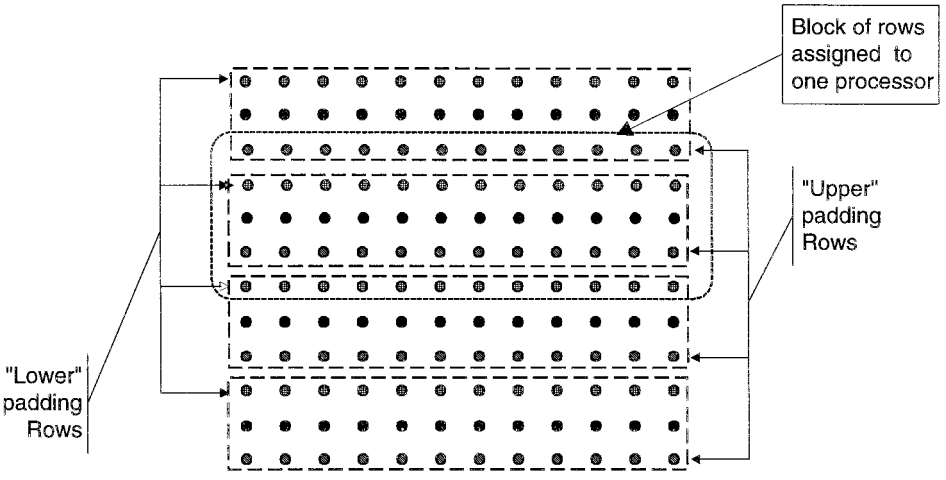

FIG. 4. Row block decomposition.

striping, tiling, and penciled partitions on clusters of workstations [11], the IBM SP2, iPSC/860, and Cray T3D processors [18].

- Low calculation amount per data exchange is due to the small grain size. This also causes low efficiency.
- Pipelined processors with short data streams are not effective in saving time.

A parallel implementation is more effective if we can increase the calculation amount per data exchange, reduce the synchronization frequency, and decrease the communication sessions. Therefore, in this paper, we focus on developing striped block asynchronous algorithms; see Fig. 4.

We abuse the notation and say that $(i, j) \in q$, implying that the grid point (i, j) is in the block of variables assigned to processor q $(1 \leqslant q \leqslant p)$.

Each grid point on the simulation mesh requires the values of its first neighbors during iteration t. If we divide the rectangular grid of points into blocks of rows, with each block considered one *work assignment* and this block of rows corresponds to a diagonal block A_{ll} of the matrix A in (1), as shown in (2). Therefore, in Fig. 4 only the top and bottom rows of each block needs to be exchanged between neighboring processors.

4.3. Stopping Criteria

One way to look at the block asynchronous algorithm is to think of it as a relaxation from the point algorithm. In other words, we reduce the interprocess communications by transmitting the most recent approximations only after each processor has performed a specified amount of calculations (*inner iterations*). We define this amount of calculations as the dataflow reduction criterion (DRC), i.e. stopping criteria in the approximation process. An interprocessor information exchange is called an *outer iteration.*

There are two distinct ways of defining DRC: one by measuring the size of the inner residuals; and another by setting a fixed number of inner iteration limit. For

the first case, let us define the residual computed in the processor q as $R^q(t) = \max\{|\phi_{i,j}(t+1) - \phi_{i,j}(t)|, \ (i,j) \in q\}$. This quantity is kept in the processor's memory between successive iterations, and it is checked if the residual is reduced by a specific amount. We use a threshold $\tau \in (0, 1]$ and we say that the residual threshold criterion is met when the residual is reduced by a factor of τ, i.e. when $R^q(t+1) \leqslant \tau \cdot R^q(t), \ \tau \in (0, 1]$.

4.4. Load Balancing and Block Allocation Methods

The overall processing of the parallel program is as follows. A master process partitions the matrix into work assignments and distributes them to parallel workers. A worker process starts calculations after it receives a work assignment and some global information. It starts information exchange with its neighbors when the DRC is satisfied. Since multiple copies of the same worker code run simultaneously, it is easy to imagine that workers will exchange data with their neighbors at different times.

After a worker receives values of the border elements from its neighbors, it will resume the same calculation process until the next DRC is met. A block is considered solved locally when the residual of the block q falls below a prescribed ε, i.e. when $R^q(t) < \varepsilon$.

The master process then collects all solved blocks and performs a Gauss–Seidel iteration across the reconstructed linear system to check if the local solutions are indeed globally convergent. The system terminates if a global convergence is reached. Otherwise the master repartitions the system and retransmits the blocks. The recalculation cycles can generate very large communication volume. A system becomes unstable when the number of recalculations is too large. An unstable system indicates that the processors are diverging into local solutions.

To further reduce the communication we can restrict the number of workers and put an exact amount of rows on each (static partition). This strategy can cause work load imbalance and, thus, negatively impact our performance since our computational test problem is a magnet with nonuniform geometry. To ease the load imbalance, we can put many smaller blocks (dynamic partitioning) in an FIFO queue, having the processors fetch the block assignments when they become idle. This way computing-intensive blocks will be automatically processed more often. The drawback is that it requires more network traffic.

4.4.1. *Static allocation algorithm details.* A static parallel block iterative algorithm has a master and many workers. The master program is responsible for constructing the $n \times n$ grid geometry, partitioning the grid into blocks of rows, assigning these blocks to the p worker modules residing on different processors, receiving results, and composing the solution matrix. Each row block that is assigned to a worker module is composed of n/p rows; see Fig. 5.

A worker module q performs Gauss–Seidel iterations on the grid points of the assigned block which is composed of rows with indexes between $ROWq(start) = ((q-1) \cdot n)/p$ and $ROWq(end) = (q \cdot n/p) - 1$, until it meets the DRC. At this point it will transmit to their neighbors the values of the grid points on their border rows and receive from them the updated values from the neighbors. In particular, worker q will transmit

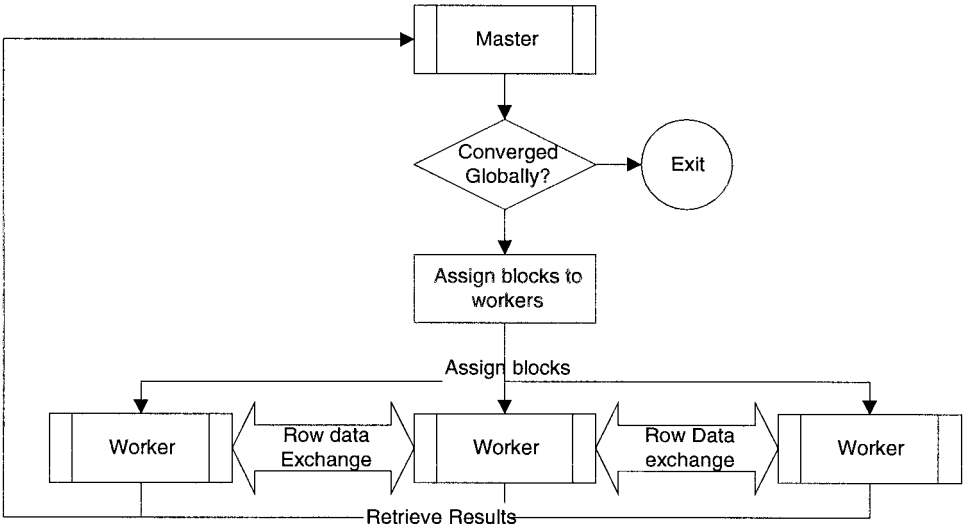

FIG. 5. Static block allocation method.

- to its upper neighbor (worker $q-1$), $ROW_q(start)$ and receive from it $ROW_q(start)-1[\equiv ROW_{q-1}(end)]$;
- to its lower neighbor (worker $q+1$), $ROW_q(end)$ and receive from it $ROW_q(end)+1[\equiv ROW_{q+1}(start)]$.

In both cases, if the neighboring workers have not reached the threshold control criteria yet, worker q is going to use values received from its neighbors in a previous outer iteration. The worker will repeat the above procedure until the local convergence criterion is met. At that point it will return all its approximated grid point values to the master module.

After receiving all locally converged blocks from the workers, the master module checks if the global convergence criterion $R_{i,j}(t) \leqslant \varepsilon \ \forall i, j \in [0, n-1]$, where $R_{i,j}(t) = |\phi_{i,j}(t) - \phi_{i,j}(t-1)|$. If not, then it reassigns the blocks to the p workers. This procedure is repeated until the global convergence criterion is met.

4.4.2. *Dynamic Allocation Algorithm Details.* In this variation of the block-asynchronous parallel algorithm, only the worker module is modified, so that it uses the tuple space as the working assignment queue. As in the static version, a working assignment (a row-block partition of the grid) is a tuple. The tuple space is used as a FIFO queue containing all nonconvergent tuples that have met the given DRC.

A dynamic worker module first reads global data (problem geometry) from the tuple space. It then extracts a work assignment tuple to compute. After the DRC is met, it will insert its border rows to be used by neighboring partitions. If local convergence is reached, it inserts the result for the master to retrieve. Otherwise, it reinserts the work assignment into the working tuple space. Such reinsertion will place the tuple at the end of the FIFO queue (see Fig. 6). If the number of working tuples is greater than the number of processors, a slow converging region will be processed by multiple processors. This can reduce load imbalance.

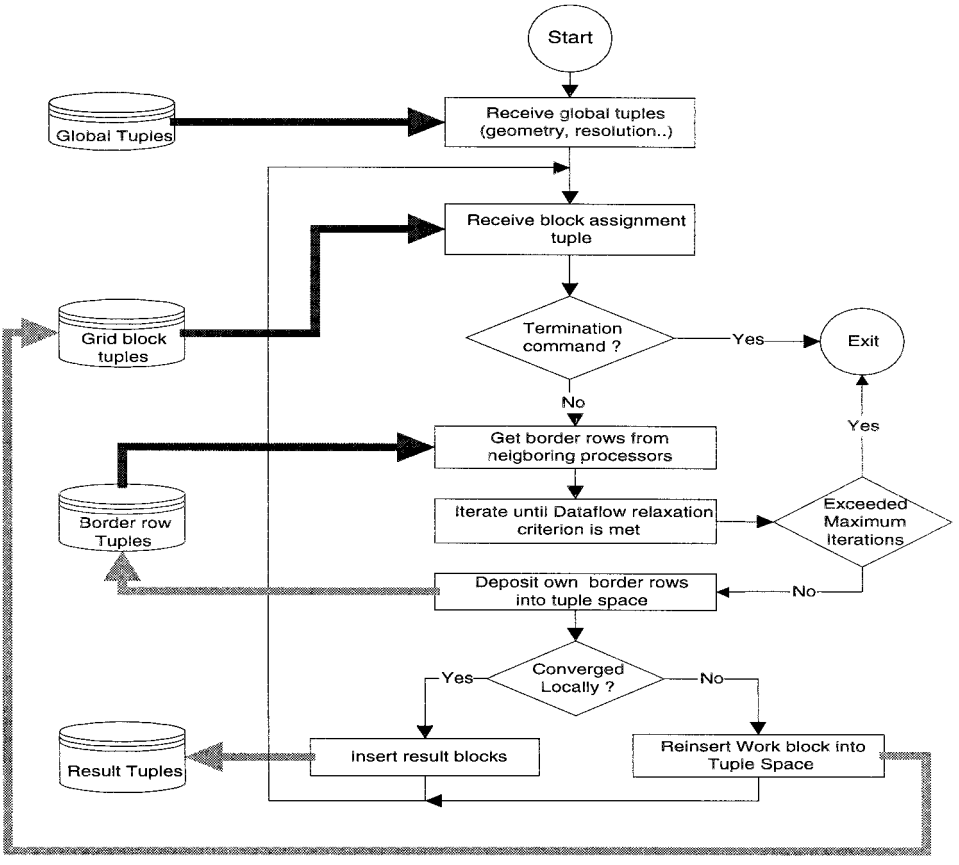

FIG. 6. Dynamic block allocation—Worker module.

5. PARALLEL PROGRAM EVALUATION

In this section, we build timing models for the static and dynamic partitioning algorithms. The objective is to identify the relative merits of both algorithms judging from their inherent interdependencies between the parallel computation and communication times. Note that the performance difference between inner iteration limit control and residual threshold control cannot be modeled analytically but it can be observed via computational results.

Timing models [24] are program time complexity-based models. Timing analysis requires timing models for the sequential algorithm and the corresponding parallel algorithm. Our scalability analysis requires calibrating the processing parameters by running once the sequential and parallel programs on a target environment.

We use the following symbols in the timing model analysis:

N number of grid points ($=n\times n$)

Qc number of floating point operations required per grid point

Q_N number of bytes required per grid point

I number of inner iterations

E number of outer iterations

W processor power in number of algorithmic steps per second

u network capacity in number of bytes per second.

Note that $W = W'/c$ represents the delivered processing power in the number of algorithmic steps (as related to their time complexity models) per second, where W' is the actual processor power in the number of machine instructions per second and c is a constant reflecting the average number of machine instructions generated from each algorithmic step. For our computational test problem, $Qc = 11$.

5.1. Sequantial Gauss–Seidel Model

The sequential processing time can be modeled as

$$T_{\text{seq}} = I \cdot \frac{N \cdot Q_C}{W}. \tag{7}$$

To calibrate W, we run the sequential algorithm on two different processors: Intel486/100 MHZ and DEC Alpha/120 MHz. Here is a summary of the statistics:

Resolution $(n \times n)$	32×32	64×64	128×128	256×256
Iterations (I)	612	2258	8812	31790
Intel 486 time (s)	4.7	67.8	1179.3	16810.2
DEC Alpha time (s)	3.0	42.4	789.3	11742.8

Using (7) and the above table, we can derive the graph in Fig. 7 depicting W, in millions of steps per second (Msps) as a function of the problem size N (grid resolution).

From Fig. 7 we can readily observe that there are little swapping effects. All programs can fit into memory. We can also notice the presence of a large constant c for this algorithm, in comparison to other well-known algorithms, such as matrix multiplication and Linpack:

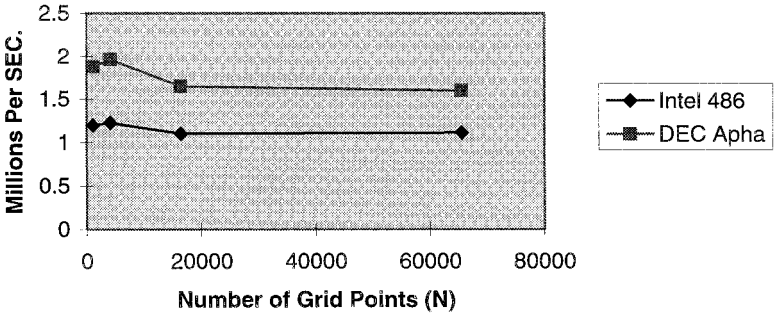

FIG. 7. W as a function of N.

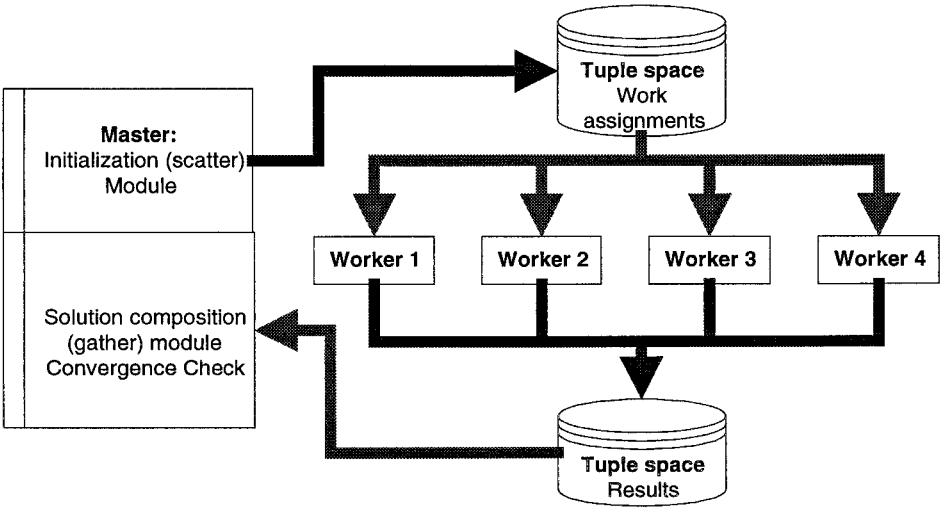

FIG. 8. Compute-aggregate-broadcast (CAB).

	Linpack	Matrix multiply (200×200)
Intel486/100MHz/8MB	2.4 Msps	4.3 Msps
Alpha/120MHz/32MB	6.4 Msps	29.4 Msps

5.2. Static Block Allocation Algorithm

Parallel iterative algorithms can be modeled as a compute-aggregate-broadcast system as shown in Fig. 8.

These types of systems have a typical timing profile as shown in Fig. 9, where T1 is the master initialization time, T2 is the task (tuple) distribution time, T3 is the worker tuple extraction time, T4 is the worker local solution computing time, T5 is the worker result submission time, T6 is the master result tuple extraction time, and T7 is the master sanity checking. The program may re-iterate T2–T7 if the global convergence check fails. Otherwise, it terminates. Note that for parallel processors using shared-medium networks or buses, communication times T2 and T3 do not overlap, nor do T5 and T6.

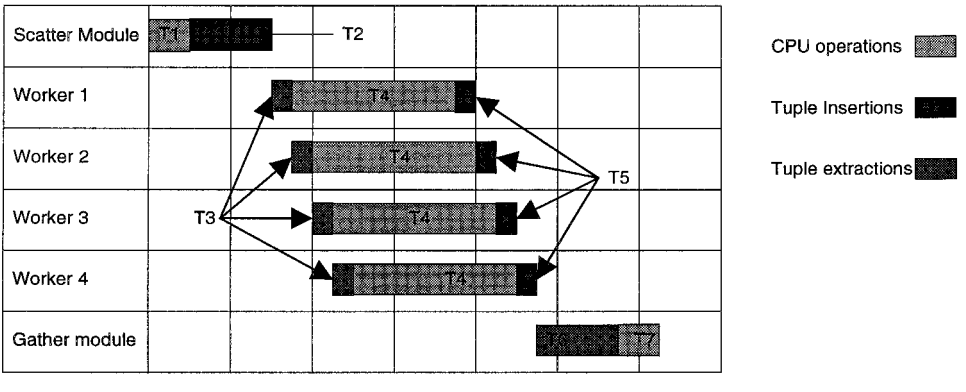

FIG. 9. CAB timing profile.

To simplify the analysis, we express the total parallel processing time as

$$T_{\text{par}} = T_{\text{master}} + T_{\text{worker}} + T_{\text{comm}} + T_{\text{sync}},$$

where $T_{\text{master}} = E \cdot Q_c \cdot N/W$ which defines the master total pure computation time, and $T_{\text{worker}} = E \cdot I \cdot Q_c \cdot N/pW$ which defines the maximum worker pure computation time (among p parallel workers), and

$$T_{\text{comm}} = Tc_{\text{master}} + Tc_{\text{worker}}. \tag{8}$$

In (8), $Tc_{\text{master}} = (2I_r \cdot Q_N \cdot N + c)/u$ which defines the master total communication time, including broadcast of global geometry, distribution of working tuples, and extraction of result tuples, and $Tc_{\text{worker}} = (2I_r \cdot Q_N \cdot N/u) + 4p \cdot E \cdot Q_N \sqrt{N}/u) + p \cdot c/u$ which defines the total worker communication time, including extraction of global information, extraction of working tuples, return of result tuples, and intermediate exchange of data with neighboring processors. Lastly, $T_{\text{sync}} = (I \cdot N \cdot Q_c \cdot E)/pW) \times (\gamma \cdot \delta - 1/\gamma)$ which defines the worst-case load imbalancing overhead, assuming γ times the difference between the fastest and slowest processors and δ times the differences between the fastest converging block and the slowest converging block.

Finally, we can define the static block algorithm timing to be

$$Ts_{\text{par}} = \left(1 + \frac{I}{p}\right) \frac{E \cdot Q_c \cdot N}{W} + \frac{I \cdot N \cdot Q_c \cdot E}{pW} \cdot \frac{\gamma \cdot \delta - 1}{\gamma} + \frac{(p+1) \cdot (2I_r \cdot Q_N \cdot N + c) + 4p \cdot E \cdot Q_N \sqrt{N}}{u}. \tag{9}$$

5.3. Dynamic Block Allocation Algorithm

The dynamic block allocation algorithm has a similar model, except that

$$Tc_{\text{worker}} = \frac{2(E + I_r) \cdot Q_N \cdot N}{u} + \frac{4p \cdot E \cdot Q_N \sqrt{N}}{u} + \frac{c \cdot p}{u}.$$

Here, assuming the best load balancing ($T_{\text{sync}} = 0$), the only difference is in the tuple re-insertion overhead. Therefore, the total dynamic time model is

$$Td_{\text{par}} = Ts_{\text{par}} + \frac{2E \cdot Q_N N}{u}. \tag{10}$$

An important investigation is to quantitatively decide when to use static or when to use the dynamic block allocation algorithm, in terms of the parallel processing environments and problem sizes. Figures 10 and 11 are obtained with numerical calculations using (9), (10), and the following assumptions:

$N = 256 \times 256$

$W = 2.5$ Mflops (million algorithmic steps per second)

$u = 100$ Kbytes per second

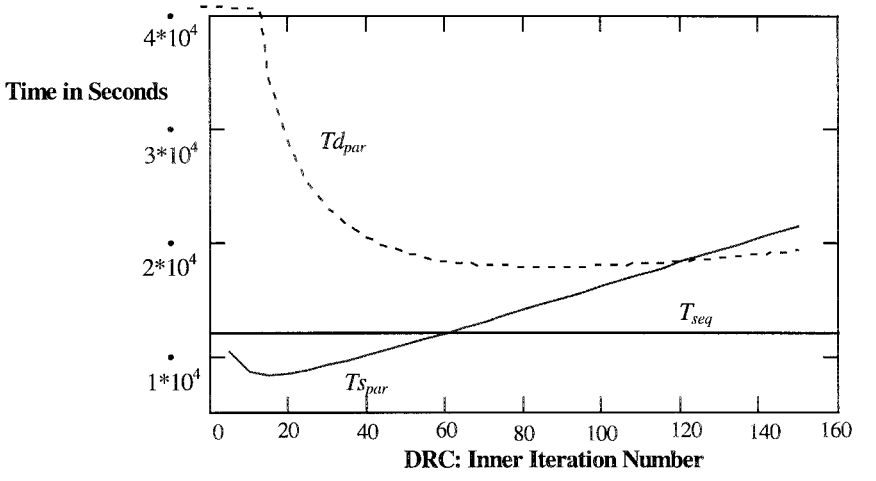

FIG. 10. Static vs dynamic with respect to inner iteration control (I).

$c = 100$ Kbytes (global broadcast data size)

$Qc = 11$ operations per grid point

$Q_N = 8$ bytes per grid point

$p = 5$

$\gamma = \delta = 2$ (two times the difference in load imbalance).

Figure 10 illustrates that for the given parallel processing environment only the static algorithm will outperform the sequential one. However, the static algorithm's convergence time will increase more rapidly than the dynamic algorithm as we increase local iterations to reduce communication. Figure 11 indicates that when the network speed goes above a certain point (approximately sustained 600 Kbps), the dynamic algorithm will out-perform the static algorithm.

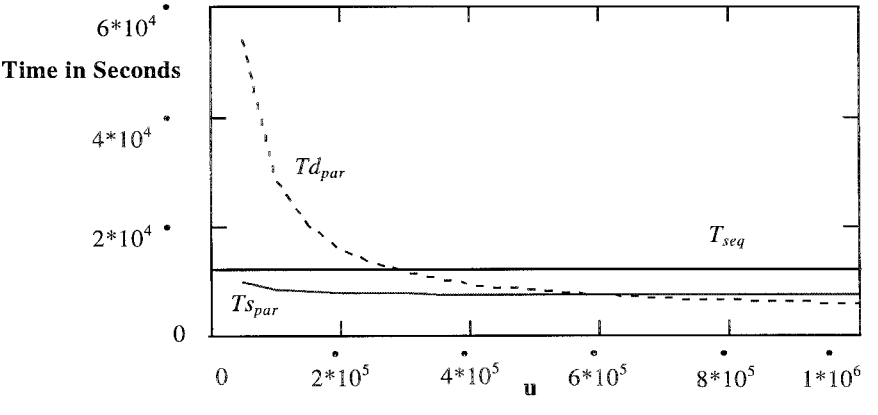

FIG. 11. Static vs dynamic with respect to network speed (u).

6. COMPUTATIONAL RESULTS

Our computational results validate the predictions of the timing models, e.g., the shape of the curve Td_{par} in Fig. 10 is confirmed in Fig. 13(a), and that of Ts_{par} in Fig. 10 is confirmed in Fig. 15(a). The experiments reveal the quantitative performance consequences of data exchange frequencies and compare the residual and inner iteration control methods. We have produced four sets of experiments. These are dynamic block allocation with residual threshold control, dynamic block allocation with inner iteration control, static block allocation with residual threshold control, and static block allocation with inner iteration control. The parallel processing environment is a cluster of five (5) DEC/Alpha workstations running the OSF/1 operating system. The workstations use a shared 10 mbps Ethernet. The parallel programming environment is C and the Synergy V3.0 system.

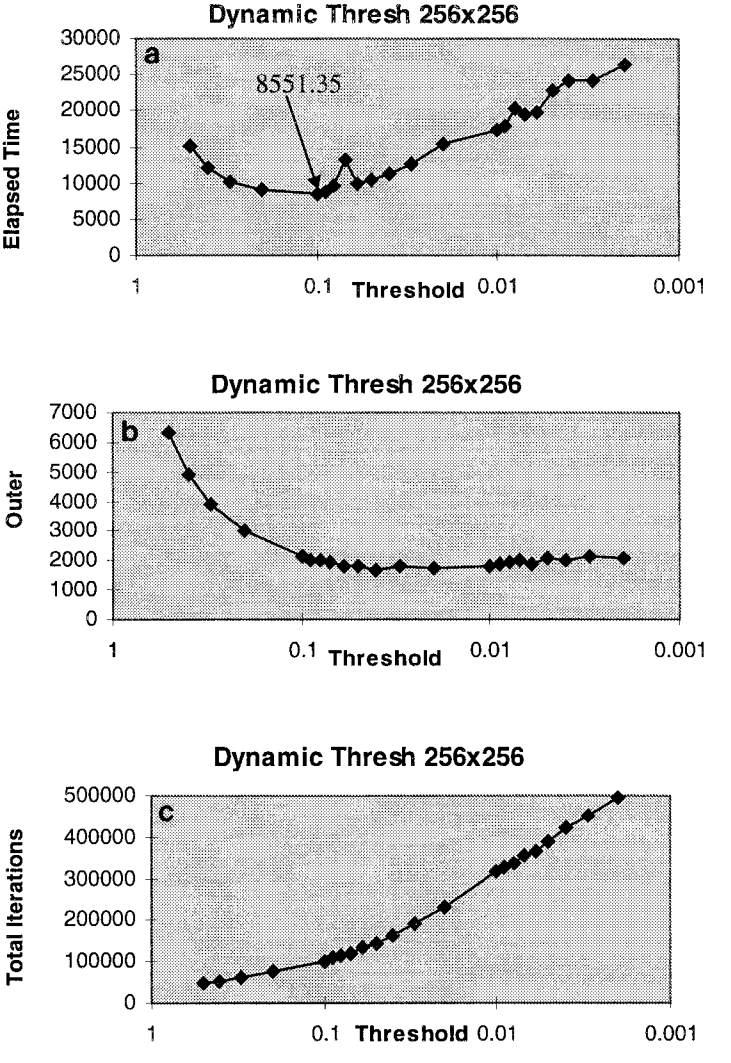

FIG. 12. Dynamic block allocation with residual threshold control.

For each set of experiments, we report elapsed times, total outer iterations, and the number of total iterations. The elapsed time measures the actual wall-clock time of the application. The total outer iteration records the changes in the outer iteration by the respective iteration control methods. The total iteration records the changes in the maximal number of inner iterations performed by a parallel worker process. The problem size is 256×256 grid points (or $N = 65{,}536$ equations). Parallel processing will yield no advantage for problems of smaller sizes.

From the performance charts (Figs. 12–15), we observe:

(a) The elapsed time curves of the inner iteration controls of both dynamic and static allocations obey the timing model predictions; compare Fig. 10, Td_{par} and Ts_{par} curves with Figs. 13(a) and 15(a). Considering that the timing models have ignored some dynamic details, the predicted optimal static DRC is very close

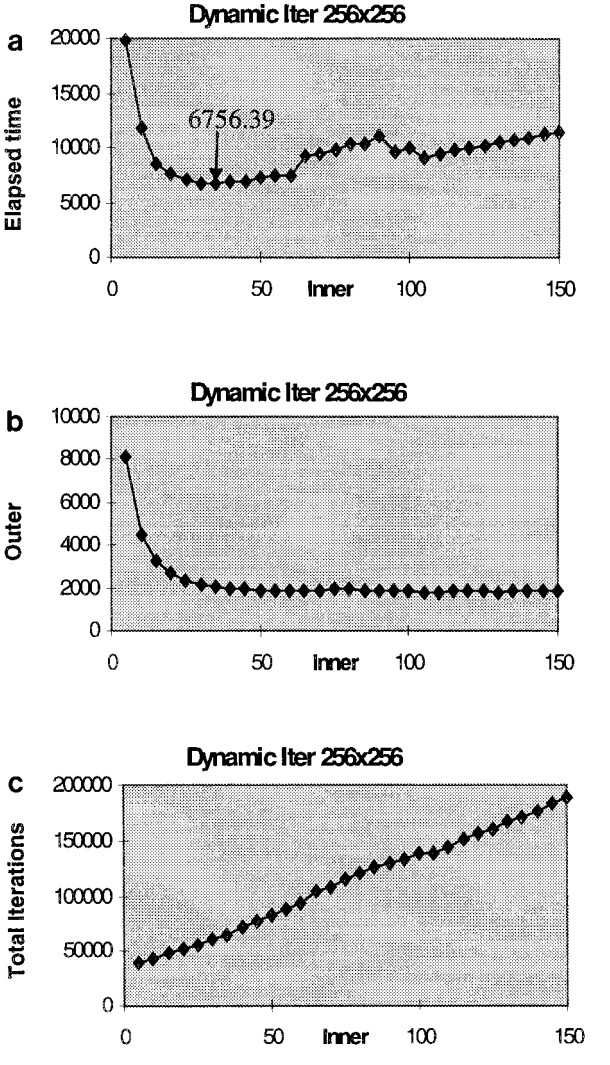

FIG. 13. Dynamic block allocation with inner iteration control.

to computational results. The predicted DRCs are 17 (Ts_{par}) and 80 (Td_{par}), respectively (Fig. 10). The actual static and dynamic optimal DRCs are 19 (Fig. 15(a)) and 30 (Fig. 13(a)).

(b) The total iteration increases linearly as we cut off the data exchanges; see Figs. 12(c)–15(c).

(c) Using the 11,742.8 second sequential elapsed time (see Section 5.1) as comparison we have the following results. Dynamic allocation with threshold control finishes in 8551.35 s, or speedup = 1.37 (see Fig. 12(a)). Dynamic allocation with iteration control finished in 6756.39 s, or speedup = 1.74 (see Fig. 13(a)). Static block algorithms performed better. Static threshold control finished in 5746.01 s, or

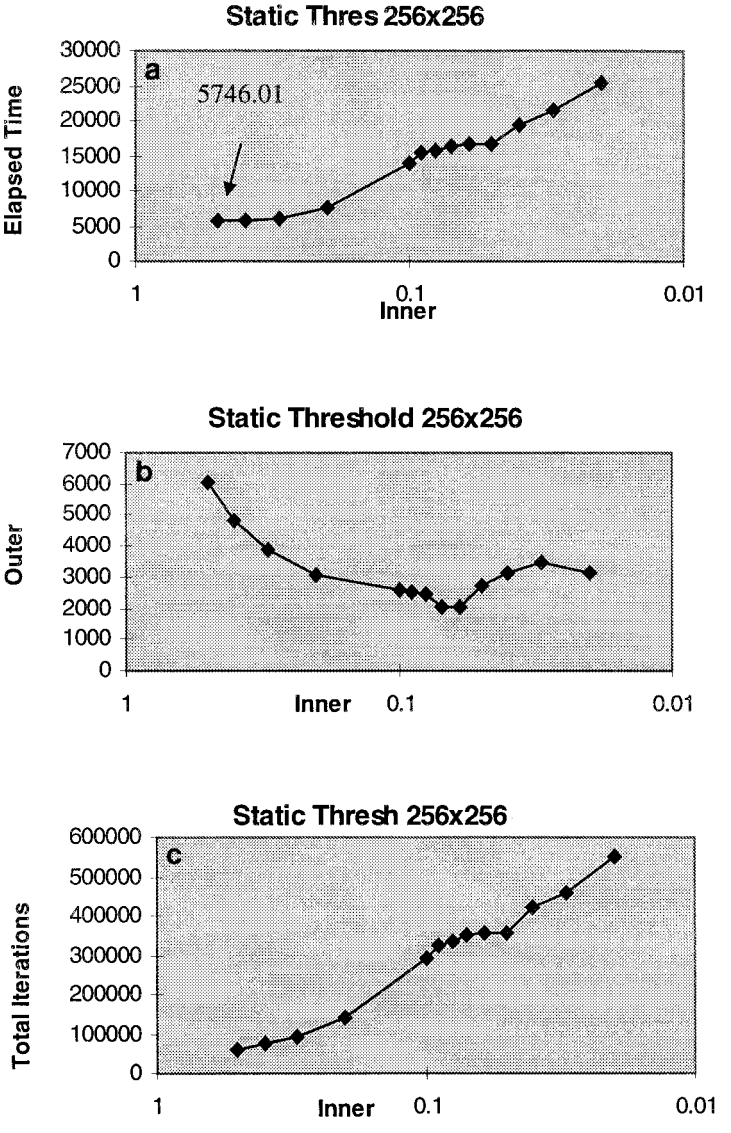

FIG. 14. Static block allocation with residual threshold control.

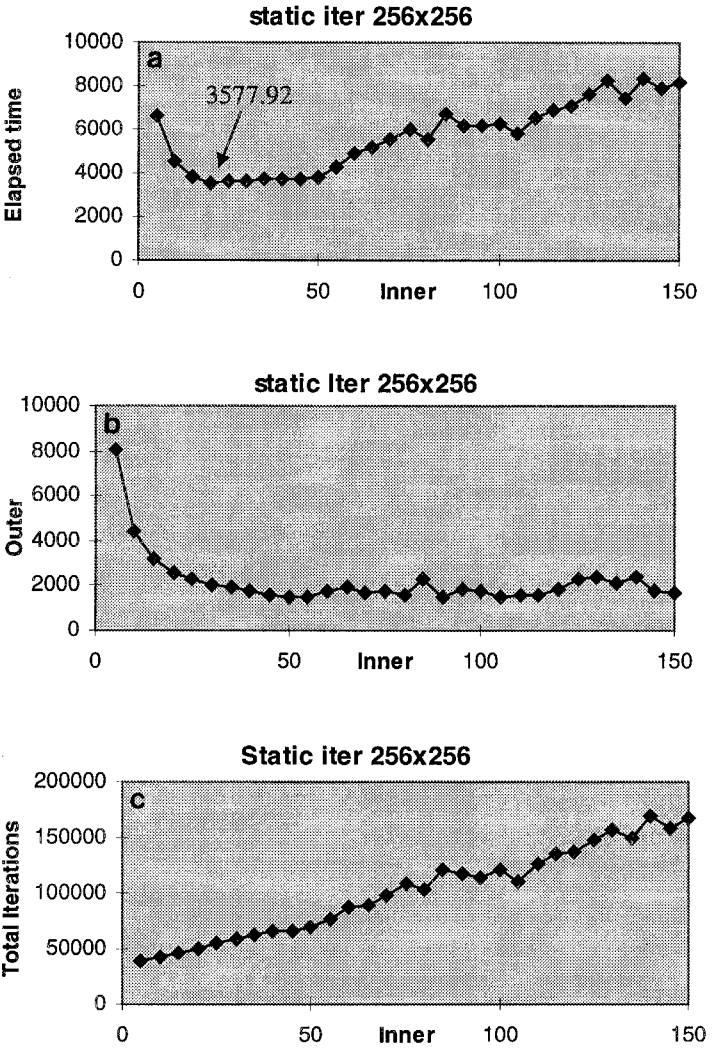

FIG. 15. Static block allocation with inner iteration control.

speedup = 2.04 (see Fig. 14(a)). Static allocation with iteration control finished in 3577.92 s, or speedup = 3.28 (Fig. 15(a)). The best speedup represents 66% parallel processing efficiency using five processors.

(d) Residual threshold control is consistently slower than inner iteration control.

7. CONCLUSIONS

In this paper we report our analysis and computational results of an investigation that trades the abundance of computing power for scarce network bandwidth using asynchronous iterative algorithms. We studied several critical aspects of developing a practical asynchronous parallel linear system solver and conclude that it is

possible to gain overall processing speed by sacrificing some local convergence speed for our test case. The test algorithms were specifically chosen for their proven asynchronous convergence and relatively long running time (1300 sequential processing seconds) in order to reveal the details of timing impact in different factors. For instance, we have found that the convergence time increases linearly (see Section 6) as we reduce the data exchange frequency. Since the overall cumulative speed of multiple processors is typically many times the interconnection network speed, for many practical applications asynchronous iterative algorithms can be more advantageous than synchronous algorithms when processed in parallel.

Our experiences in applying the timing model method to parallel algorithm analysis showed that it is possible to analytically prototype a complex parallel system with little program instrumentation. The results can be used to predict the scalability of the system as well as to identify optimal performance factors.

REFERENCES

1. S. Ahuja, N. Carriero, and D. Gelertner, Linda and friends, *IEEE Comp*. (August 1986), 26–32.
2. O. Axelsson and P. S. Vassilevski, A black box generalized conjugate gradient solver with inner iterations and variable-step preconditioning, *SIAM J. Matrix Anal. Appl*. **12** (1991), 625–644.
3. D. Bailey, "Twelve Ways to Fool the Masses When Giving Performance Results on Parallel Computers," RNR technical report RNR-091-020, NASA Ames Research Center, 1991.
4. R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst, "Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods," SIAM, Philadelphia, 1994.
5. E. Baz, A method of terminating asynchronous iterative algorithms on message passing systems, *Parallel Algorithms Appl*. **9** (1996), 153–158.
6. D. P. Bertsekas and J. N. Tsitsiklis, "Parallel and Distributed Computation," Prentice–Hall, Englewood Cliffs, NJ, 1989.
7. A. Beguelin, J. Dongarra, A. Geist, R. Mancheck, and V. Sunderam, PVM: Heterogeneous Network Computing, *in* "Sixth SIAM Conference on Parallel Processing," SIAM, Philadelphia, 1993.
8. Y. Shi, "The Synergy System: Manual and Programming Guide," CIS Department, Temple University, 1992. [www.cis.temple.edu/∼shi/synergy.html]
9. R. Bru, V. Migallon, J. Penades, and D. B. Szyld, Parallel, synchronous and asynchronous two-stage multisplitting methods, *Electron. Trans. Numer. Anal*. **3** (1995), 24–38.
10. M. I. Darby, Calculation of the fields near permanent magnets, *in* "Physics Programs," pp. 125–149, Wiley, New York, 1980.
11. J. Dougherty, Variable-size partitioning approaches for a distributed application, *in* "Joint Conference on Information Sciences, Pinehurst, NC, November 1994."
12. K. Dowd, "High Performance Computing," O'Reilly, Sebastopol, CA, 1993.
13. A. Frommer and D. B. Szyld, H-splittings and two-stage iterative methods, *Numer. Math*. **63** (1992), 345–356.
14. A. Frommer, H. Schwandt, and D. B. Szyld, Asynchronous weighted additive Schwarz methods, *Electron. Trans. Numer. Anal*. **5** (1997), 48–61.
15. A. Frommer and D. B. Szyld, Asynchronous two-stage iterative methods, *Numerische Mathematik* **69** (1994), 141–153.
16. F. Halsal, "Data Communications, Computer Networks and Open Systems," Addison–Wesley, New York, 1996.

17. D. P. Koester, S. Ranka, and G. C. Fox, A parallel Gauss–Seidel algorithm for sparse power system matrices, *in* "Proceedings of Supercomputing '94, pp. 184–193, 1994."
18. S. Kortas and P. Angot, A practical and portable model for programming for iterative solvers on distributed memory machines, *Parallel Computing* **22** (1996), 487–512.
19. P. J. Lanzkron, D. J. Rose, and D. B. Szyld, *Numer. Math.* **58** (1991), 685–702.
20. Message-passing Interface Forum, "MPI: A Message Passing Interface Standard," CIS technical report CS-94-230, University of Tennessee, Knoxville, 1994.
21. J. C. Miellou, D. E. Baz, and P. Spiteri, A new class of asynchronous iterative algorithms with order intervals, *Math. Comp.* **67** (1998), 237–255.
22. D. A. Reed and R. M. Fujimoto, "Message Based Parallel Processing," MIT Press, Boston, 1988.
23. S. A. Savari and D. P. Bertserkas, Finite termination of asynchronous iterative algorithms, *Parallel Computing* **22** (1996), 39–56.
24. Y. Shi, Program scalability analysis, *in* "Proceedings of Ninth IASTED International Conference on Distributed and Parallel Processing, Georgetown University, Washington, DC, 1997," pp. 451–456.
25. R. S. Varga, "Matrix Iterative Analysis," Prentice–Hall, Englewood Cliffs, NJ, 1962.