Parallel Algorithms for the Iterative Solution to Linear Systems

R. H. Barlow and D. J. Evans
Department of Computer Studies, Loughborough University of Technology, Loughborough, Leicestershire, UK

In this paper, parallel algorithms suitable for the iterative solution of large sets of linear equations are developed. The algorithms based on the well known Gauss Seidel and SOR methods are presented in both synchronous and asynchronous forms. Results obtained using the M.I.M.D. computer at Loughborough University are given, for the model problem of the solution of the Laplace equation within the unit square.

INTRODUCTION

The traditional methods of solving linear systems iteratively are Gauss-Jacobi, Gauss-Seidel and SOR, each of which has a greater rate of convergence than its predecessor. To further increase the rates of convergence of the Gauss-Seidel and SOR methods, researchers have developed block forms as opposed to the original pointwise forms of these methods.[1,2] The Gauss-Seidel and SOR methods have traditionally imposed a certain constraint on the ordering of the updates of point/blocks within an iterative cycle. The importance of the ordering scheme and the structure of the linear system to the convergence power of these methods was summarized in precise rules by the results of Young[2] and Varga.[1] For systems not satisfying these rules, little theory concerning convergence can be determined.

The advent of parallel computers poses a challenge because when equations or subsets of equations are updated in parallel then there is no guaranteed update ordering scheme possible. Thus, is it possible to find parallel schemes that converge with the full power of sequential schemes?

Lambiotti and Voigt,[3] considered tridiagonal linear systems, and gave parallel algorithms that reproduced exactly the results of the sequential schemes. These algorithms required some synchronization to achieve this. Here we extend their work first to arbitrary linear systems and then to examining the consequences of using no synchronization to coordinate the processors. We will show that it is possible to construct algorithms, that using no synchronization, reproduce almost exactly the results of the original sequential algorithm.

Algorithms that require no synchronization, called asynchronous algorithms, have achieved importance in parallel computing because the coordination of processors is an overhead in the execution of algorithms and in some algorithms/system combinations destroys any gains from having multiple processors co-operating on the algorithm execution.[4,5]

The next section presents the usual Gauss-Jacobi, Gauss-Seidel and SOR methods, outlines the rules of Young and Varga and then develops parallel synchronous schemes that produce the exact same results as their sequential counterparts. The middle section discusses asynchronous variants of the schemes that violate or perturb slightly the ordering rules. Results for both types of scheme, obtained using a MIMD parallel computer at Loughborough University are given in the final section.

ITERATIVE METHODS: SYNCHRONOUS VERSION

Given a linear system

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

one can write without loss of generality,

$$\mathbf{A} = \mathbf{I} - \mathbf{L} - \mathbf{U} \tag{2}$$

with matrices L and U consisting only of elements below and above the diagonal respectively. Then, the standard iterative form of the Gauss-Jacobi method is

$$\mathbf{x}^{(n+1)} = (\mathbf{L} + \mathbf{U})\mathbf{x}^{(n)} + \mathbf{b} \tag{3}$$

whilst the Gauss-Seidel method is,

$$\mathbf{x}^{(n+1)} = \mathbf{L}\mathbf{x}^{(n+1)} + \mathbf{U}\mathbf{x}^{(n)} + \mathbf{b} \tag{4}$$

and the SOR method is

$$\mathbf{x}^{(n+1)} = \mathbf{x}^{(n)} + \omega(\mathbf{L}\mathbf{x}^{(n+1)} + \mathbf{U}\mathbf{x}^{(n)} + \mathbf{b} - \mathbf{x}^{(n)}) \tag{5}$$

where ω is the over-relaxation parameter chosen to maximize the convergence rate of the method, and n is the iteration number.

In the forms written above, one sees clearly that all new components of the vector x in the Gauss-Jacobi method are calculated from all the old component values of the x within an iterative cycle. All component updates can therefore be carried out in parallel with synchronization being required only between iterative cycles.

In contrast, the Gauss-Seidel and SOR methods use new values within an iterative cycle in a systematic manner that, in Eqns (4) and (5), demands a strictly sequential evaluation of the components.

The advantages of the Gauss-Seidel and SOR methods over the Gauss-Jacobi method is that firstly they require only 1 copy of the components and secondly that they converge much faster. In the case that matrix A possesses property A, viz.

$$\mathbf{A} = \begin{bmatrix} \mathbf{I} & \mathbf{B} \\ \mathbf{B} & \mathbf{I} \end{bmatrix} \tag{6}$$

i.e. A is of 2 cyclic form, and is solved in a consistent order,[2] then the Gauss-Seidel method converges twice as

CCC-0010-4620/82/0025-0056 $02.50

fast as Gauss-Jacobi and SOR converges $2/\varepsilon$ as fast with $0 < \varepsilon \ll 1$.

When property A is not satisfied very little about the convergence rates is known.

It is thus important to derive parallel forms of the Gauss-Seidel and SOR methods. Baudet[4] considers partially violated schemes wherein the components are grouped into blocks where blocks are computed in parallel to each other whilst within a block, equations are treated sequentially by the Gauss-Seidel or SOR methods. We should however point out that the main consideration of Baudet's paper is chaotic relaxation applied to non-linear systems.

The basis of our method is that Property A implies that the equations can be divided into two disjoint subsets, where the update equations for components in one set refer only to components in the other set. One possible consistent ordering scheme is obtained by first updating all the components in the first set and then those in the other set. Since Property A is satisfied, then, all updates within a set of components are independent and can be done in parallel.

Thus, we have a formulation of a parallel Gauss-Seidel and SOR method that proceeds in two steps that must alternate with each other within an iterative cycle. Thus they require synchronization between each set and between iterative cycles but each step has a level of parallelism equal to the number of members in its subset.

Thus, for example, if A has a tridiagonal form then it satisfies Property A since one can divide equations into two disjoint subsets consisting respectively of the even and the odd indexed components, where the update equation for a component in one set relates only to components in the other set. One consistent ordering, called red-black, is obtained by first updating all even-numbered components and then all odd-numbered components. The usual ordering obtained by considering the components in strictly increasing order is also a consistent ordering. Both orderings converge asymptotically at the same rate but no parallel method can be derived from the latter ordering.

THE ITERATION METHODS: ASYNCHRONOUS VERSION

We have seen that synchronization is required to ensure that components in each subset are all updated before updates in the other subset are started. This can result in loss of processor power due to fast processors waiting on slower ones.

For MIMD computers configured as multiple independent processing elements with no processor having overall control, the unique allocation of paths to processors, and the signalling of termination of a path, involves synchronization. This synchronization can result in a significant loss of processing power depending on the demand rate for synchronization from the program and the implementation time on a given system.

Kung[5] has therefore considered parallel iterative algorithms that proceed asynchronously: that is, processors are initially allocated to a parallel path and remain iterating that path until completion of the algorithm without any synchronization between the parallel paths. Data communication still takes place, with the most recent value of any particular data being used but without any concept of how many times that used data has been iterated.

Baudet[4] has applied these concepts to linear systems of equations and obtained results for the example of the model problem Laplace equation within the unit square with Dirichlet boundary values. As mentioned earlier, his methods are based on allocating a processor permanently to a fixed block of components, blocks being processed in parallel with respect to each other with sequential processing within a block following the standard Gauss-Seidel or SOR scheme. Clearly his method violates the consistent ordering scheme at the boundaries of the block; in linear systems with more complex dependence between components this violation may be much more extensive. Asynchronous versions are obtained by allowing processors to iterate blocks without waiting for other blocks to complete, and without the synchronization that insures that components in a block, used by another block are all from the same iterative cycle.

Let us now develop from the synchronous scheme given in the preceding section an asynchronous version. This asynchronous scheme may violate minimally, in a manner to be discussed, the consistent ordering scheme.

This asynchronous scheme will forgo the synchronization that insured: (i) once and once only uptake of a component each cycle; (ii) that all components updates in 1 subset are completed before the processing of the next subset commences.

The asynchronous scheme is based on two components: (a) a shared list formed by the end to end concatenation of the two subset lists and a shared pointer into this list; (b) a process, executed by each processor, consisting of copying the index value to local data space, incrementing the shared index value and then iterating the component corresponding to the local index value.

One can note that if the accesses to the index were protected by a critical region/synchronization primitive then constraint (i) above would be satisfied.

If processors execute at the same speed and their start time is minimally offset with respect to each other then at any time the components being iterated will be consecutive components from the cyclic list of components of the system. Offsetting the start time is simply achieved by forcing processors to initialize via the update of a shared data item protected by a critical region/synchronization primitive.

The algorithm, written in a loose form of FORTRAN is,

```
C   Data shared by processors is size, components,
      subset list and pointer convergence flag
    $SHARED N, XCOMP, LIST, INDEX,
      BCONVF
    DIMENSION XCOMP(N), LIST(N)
C   Initialize components
    XCOMP(I) =
C   Initialize list of the order in which the components
      are to be taken up
    LIST(I) = permutation of I
C   Initialize index into list
    INDEX = 0
C   Initialize iteration counts
    ITER1 = 0, ITER2 = 0
```

```
C   Initialize local and shared convergence flags
    BCONV = TRUE
    BCONVF = FALSE
C   Set up process for each processor
    $DOPAR 100 IPROC = 1, NPROC
C   Loop on collecting component
1   CONTINUE
C   First test if result converged
    IF(BCONVF) GOTO 100
C   Collect next component: do not synchronize access
    ILOCAL = INDEX
    IF(ILOCAL . EQ . N) ILOCAL = 0
    ILOCAL = ILOCAL + 1
    INDEX = ILOCAL
C   Get component corresponding to local value of
      index
    ICOMP = LIST(ILOCAL)
C   Solve equation for component: if not converged set
C     BCONV = FALSE
    CALL SOLVE(LINE, BCONV)
    ITER1 = ITER1 + 1
C   Test if N iterations locally since last convergence
      test
    IF(ITER1 . LT . N) GOTO 1
    ITER2 = ITER2 + 1
C   Test convergence
    IF(BCONV) GOTO 10
C   Not converged
    ITER1 = 0
    BCONV = TRUE
    GOTO 1
10  BCONVF = TRUE
100 $PAREND
```

We briefly note that the convergence test may not be the best. Better ones have been utilized but involve a more detailed description which we shall forego here.

Let us examine the consequences of not satisfying constraints(i) and (ii) together with the possibility of unequal speed processors.

(1) No forced once and once only update of the index of the next component to be iterated. Clearly with processors executing at different speeds then two or more processors will eventually access and modify the index at almost the same time. Multiple processors will then execute the same component. The chance of this happening, on each uptake, is approximately

$$(\text{Time (index update)}/\text{Time (component iteration)}) * (\text{No. of processors} - 1)$$

which is usually very small. This will result in a waste of processor time, and a corresponding loss of speedup. Its pertubation of the iteration scheme values is likely to have only a minimal overall effect. Its effect can be eliminated by making processors first copy to local memory all components used and updated in the scheme. They then carry out the more time consuming update calculation. Now since processors started the component iteration at almost the same time then unless faster processors have started updates before slower ones have finished copying then multiple processors iterating the same component has no effect.

(2) No synchronization between disjoint subsets. It follows that components from both the original disjoint subsets may be iterated at the same time. If processors execute at the same speed then the component set in iteration at any time are a consecutive subset of the new cyclic list. If processors are executing N components then the chance of components lying in both subsets is $n/(N-1)$. However, even if components in both subsets are being iterated concurrently this will produce no effect in many cases.

Consider the case of a tridiagonal linear system: then the two original disjoint subsets consist respectively of all even-numbered and all odd-numbered components. If subsets are ordered by increasing component number then the ith component of one subset relates only to the ith and $i \pm 1$th component of the other subset. Since the subsets are concatenated end to end for the asynchronous scheme unless $n > (N/2 - 1)$ then no interference occurs in the concurrent iteration of components from both subsets.

If processors do not all execute at the same speed then the current set of iterating components will not always be consecutive: although the scheduling algorithm of processors to the next indexed component will always tend to restore the consecutive structure. Any effect this has depends on n and N. The worse effect in terms of components that this can have, can be assessed by noting that if processor speeds are s_i, $i = 1, n$ with respect to the slowest processor then this worst case is equivalent to having $m = \sum_{i=1}^{n} s_i$ processors of the same speed.

Obviously systems can have component relating structures more complicated than that of the tridiagonal one but similar principles will still apply.

To appreciate the importance of this point, let us again consider the tridiagonal system discussed above. If the system has 16 components then in the circular list the newer dependent components are seven components apart. Thus, if two processors are cooperating then one can be six times slower than the other without there being any interference between them. All the potential of both processors is realized whilst at the same time exactly the same result as the sequential algorithm is reproduced. With Baudet's method with processors locked to blocks,[4] the boundary values of the block of components iterated by the faster processor would only have stabilized six times later than the faster processor had thought its subset of components had converged.

RESULTS

Consider the solution of the Laplace equation with Dirichlet boundary values discretized on a square mesh to yield the 5-point equation for each point

$$\mathbf{x}_{i,j-1} + \mathbf{x}_{i-1,j} - 4\mathbf{x}_{i,j} + \mathbf{x}_{i+1,j} + \mathbf{x}_{i,j+1} = 0 \quad \text{for } i, j = 1, 2, \ldots, N,$$

which relate the update formula for any point (i, j) to only its nearest neighbours. Grouping points on rows together, one obtains a block equation that relates the update of components on 1 row only to the values of components on its nearest neighbour rows. Thus, updates to even indexed rows use only values from odd indexed rows and the system satisfies Property A.

Table 1. Synchronous and asynchronous versions of Gauss-Seidel and SOR

Method	Mesh	Over-relaxation parameter	Speed-up two processors	Shared memory access rate (cycles)	Synchronization access rate (cycles)	Shared memory loss		Synchronization loss	
						Access	Contention	Access	Contention
Synchronous GS	16 × 16	1.0	1.71	1:115	1:24K	1%	0.3%	5.5%	5.0%
SOR	16 × 16	1.60	1.71	1:115	1:24K	1%	0.3%	5.5%	5.0%
GS	32 × 32	1.0	1.82	1:120	1:50K	0.9%	0.3%	3.2%	2.5%
SOR	32 × 32	1.77	1.82	1:120	1:50K	0.9%	0.3%	3.2%	2.5%
Asynchronous GS	16 × 16	1.0	1.97	1:120	—				
SOR	16 × 16	1.60	1.95	1:120	—				
GS	32 × 32	1.0	1.97	1:120	—				
SOR	32 × 32	1.77	1.95	1:120	—				

Any scheme that alternates the update of first, all the even-components and then all the odd-components is consistently ordered. Thus, the asymptotic rate of convergence of the SOR scheme is $2/\varepsilon$ with respect to the Gauss-Jacobi scheme. In order to minimize the overlap of processing of dependent components in the asynchronous scheme the sublists are constructed in order of increasing row number viz. (1, 3, 5, . . .) and (2, 4, 6, 8, . . .), known commonly as σ_2 ordering.[2]

The resultant SOR block method was programmed as indicated in the preceding two sections and run on a two processor MIMD system at Loughborough University. The results are indicated in Table 1. We should note that the comparison is with the corresponding sequential algorithm with no synchronization and no shared data overhead. We note also that in the parallel results, the mean number of iterations was identical with that of the sequential algorithm. Examination of the end grid values showed that except for the first row in the asynchronous version, they were identical to the sequential results. The discrepancy in the asynchronous version arises because the convergence test that stops the iteration is done by any processor that terminates a given iterative cycle: the other processors continue iterating components testing the 'STOP' flag only on taking up a component.

As is standard in our work[6,7] we give the mean rate of access to the shared data and to the synchronization tool. From these figures, one can predict parallel overheads for the algorithms running on any MIMD system given its unit access overhead to stored data and the synchronization tool. The actual overheads in accessing the shared data and synchronization tool on the Loughborough system which are given in the table are divided into two parts: one arising from accessing the resource and the other from the loss due to processors contending and being forced to wait on the shared resource. Contention figures are obtainable when the algorithm does the same amount of real work irrespective of the number of cooperating processors. The separation between contention for the shared memory and synchronization was possible because synchronization was controlled through software and thus waiting was measurable by the software.[6,8]

Following Baudet[4] we partitioned the components into two fixed subsets ($i = 1, 2, \ldots, N/2; N/2 + 1, \ldots, N$) and forced each of the two processors to permanently iterate within one subset. The results are given in Table 2. We have chosen to give only the number of iterations required to converge. The comparison is with the usual sequential Gauss-Seidel and SOR iterating components in increasing order 1,2,3, . . . , N. The results make it clear that in this method the results differ numerically depending on the number of cooperating processors. If one ignores the parallel overheads due to shared memory access the speedup is directly related to the number of iterations/number of processors. In some cases, speedup greater than the number of processors occurs.

Table 2. Asynchronous Gauss-Seidel and SOR using the method of Baudet (1978)

Mesh	Over-relaxation parameter	Mean number of iterations (sequential)	Mean number of iterations (parallel)
16 × 16	1.0	37	38
16 × 16	1.60	17	16
32 × 32	1.0	58	60
32 × 32	1.74	33	32

CONCLUSION

We have derived a parallel version of the Gauss-Seidel and SOR iterative schemes that in the case of a given system with Property A and consistent ordering gives the exact same results as the usual sequential method. Asynchronous algorithms were also derived that under weak conditions on the number and speed of processors relative to the size of the system also give the same results as the sequential methods.

The work in the paper can almost certainly be generalized to linear systems where A is of p-cyclic form.[1] In this case, parallelism within the p subsets would be possible while synchronization between the p subsets would be required. However, the exact form of the algorithm and in particular the conditions under which the asynchronous version works are under further investigation.

Acknowledgement

Research supported by SRC under its Distributed Computing Research Program.

REFERENCES

1. R. S. Varga, *Matrix Iterative Analysis*, Prentice Hall, Englewood Cliffs, New Jersey (1963).
2. D. M. Young, *Iterative Solutions of Large Linear Systems*, Academic Press, London (1971).
3. J. J. Lambiotti and R. G. Voigt, The solution of tridiagonal linear systems on the CDC star 100 computer. *ACM Transactions on Mathematical Software* **1**, 308–329 (1975).
4. G. M. Baudet, Asynchronous iterative methods for multiprocessors. *Journal of the ACM* **25**, 226–244 (April 1978).
5. H. T. Kung, Synchronous and asynchronous parallel algorithms for multiprocessors, in *New Directions and Recent Results in Algorithms & Complexity*, ed. by J. F. Traub, Academic Press, London (1976).
6. R. H. Barlow and D. J. Evans, Analysis of the performance of a dual minicomputer parallel computer system, in *Proceedings of Eurocomp 1978*, Online Conferences, Uxbridge, 259–276 (1978).
7. R. H. Barlow and D. J. Evans, A parallel organisation of the bisection algorithm. *Computer Journal* **22**, 267–269 (August 1979).
8. I. A. Newman and M. C. Woodward, *Reliable Sharing of Passive Resources in a Multiprocessor Environment*, Internal Report No. 45, Department of Computer Studies, Loughborough University of Technology.

Received November 1980

Note added in proof: We have run the above-mentioned parallel algorithms on a new 4-processor asynchronous parallel computer at Loughborough University.[1]

The speedups for the synchronous version of the algorithm are 1.91, 2.65 and 3.78 using two, three and four processors respectively, while the corresponding results for the asynchronous algorithm are 1.96, 2.93, 3.94 for a grid of 32×32 points when used in the solution of the Laplace Equation in the unit square. The two processor speedups are better than the corresponding results in Table 1 of the paper because the synchronization tool on the new system is more efficient.

1. R. H. Barlow, D. J. Evans, I. A. Newman and M. C. Woodward, *The NEPTUNE Parallel Processing System*, Report of the Department of Computer Studies, Loughborough University, UK (1981).