Parallel Overrelaxation Algorithms for Systems of Linear Equations

Rudnei Dias da Cunha *
Computing Laboratory, University of Kent at Canterbury, U.K.
Centro de Processamento de Dados,
Universidade Federal do Rio Grande do Sul, Brasil

Tim Hopkins
Computing Laboratory, University of Kent at Canterbury, U.K.

Abstract. Parallel versions of the Jacobi and Successive Overrelaxation methods to solve systems of linear equations on a network of transputers are discussed. The Jacobi method is easily parallelizable, by distributing the data in a rowwise fashion among the processors. The SOR is a less obviously parallelizable method due to data-dependency issues, although some computation can still be distributed. An important feature of our implementations is the parallel computation of matrix norms, which allow a significant speed-up over the serial versions of the algorithms. We show a significant improvement in the efficiency of our implementations over other previously published parallel iterative linear equations solvers.

1 Introduction

Iterative methods to solve systems of linear equations attempt to find a solution of a given system by successively refining an initial guess to that solution.

Let

$$Ax = b \tag{1}$$

be a non-singular system of linear algebraic equations of order n, and $x^{(0)}$ an initial approximation; this approximation may be improved by means of the iteration

$$x^{(i+1)} = x^{(i)} + u, \quad i = 0, 1, \ldots$$

where u is an update vector.

Parallel iterative methods to solve systems of linear equations must deal with two main issues:

- how to partition the data *and*
- how to parallelize the iteration.

*This work was supported by CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico, Brasil) under grant 204062/89.6

Usually, the matrices and vectors are partitioned in one of three ways: by rows, by columns, or by blocks. Some partitionings are more suitable to particular parallel architectures, e.g., column partitioning allows programs written in Fortran to achieve efficient use of vector processors. On a distributed memory architecture, a rowwise or blockwise partitioning is more usual.

Given an iterative formula, it is possible that it will only provide a low degree of parallelism due to data dependency issues. In this case, some methods can be modified to allow an efficient parallel version to be implemented.

As an example, consider the Gauss-Seidel iteration to solve (1)

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right), k = 0, 1, \ldots \tag{2}$$

The parallelization of (2) may only provide a small speed-up over the sequential computation, as the values produced, $x_i^{(k+1)}$, are needed inside the iteration. This could means processors having to wait for updated values. In a distributed memory architecture, this would imply a very high rate of communications, slowing down the algorithm (in some cases, making it slower than a serial implementation).

A review of some iterative methods to solve systems of linear equations is presented by Hadjidimos [3]. One method, Successive Overrelaxation (§3.3) has been implemented on a number of parallel architectures with moderate success. Evans [2] uses a blockwise partitioning on a shared-memory architecture. Patel and Jordan [7] used a rowwise partitioning on the HEP computer. More recently, Niethammer [6] compares the Jacobi Overrelaxation (§3.2) method with the Successive Overrelaxation on a shared-memory architecture. Bonomo and Dyksen [1] present a new parallel iteration technique to solve a system of linear equations, based on pipelining successive iterations.

We will discuss the parallel implementations of the Jacobi and Successive Overrelaxation methods on a distributed processor architecture, using a network of transputers.

2 Parallel model used

2.1 Programming environment

Our algorithms were implemented in Occam 2 on a MEiKO Computing Surface[1]. We have used up to 16 T800 transputers, with 4KB of local memory and 4MB of extended memory.

2.2 Structure of processors

The processors were organized in a ring, with one of the processors designated as the *Master*; this processor controls the execution of the algorithm. The other processors are termed *Servers*, and they execute as requested by the master processor. The servers are responsible for the computations on the distributed data. The master processor also acts as a server, as part of the data is assigned to it. This ensures that all the processors are kept as busy as possible.

[1] MEiKO and the Computing Surface are trade-marks of Meiko Limited.

3 Parallel iterative methods for systems of linear equations

3.1 *Jacobi Overrelaxation (JOR)*

Given (1), the Jacobi overrelaxation (JOR) method is given by

$$x^{(k+1)} = x^{(k)} + \omega D^{-1}(b - Ax^{(k)}), \quad k = 0, 1 \ldots, \tag{3}$$

where D is the diagonal of A, and ω is the relaxation parameter.

3.2 *Parallel version of the JOR*

We note that the JOR method presents no data-dependency between elements of the x vectors during each iteration; thus, the JOR iteration is easily parallelizable. Using a row partition, iteration (3) can be distributed among P processors in the following way

$$x^{(k+1)}_{r_p:q_p} = x^{(k)}_{r_p:q_p} + wD^{-1}_{r_p:q_p,*}(b_{r_p:q_p} - A_{r_p:q_p,*}x^{(k)}), \quad p = 1, 2, \ldots, P \tag{4}$$

where the subscript $r_p{:}q_p$ means that processor p will update elements $x_i, i = r, r+1, \ldots, q$. The $*$ denotes all the columns.

The parallel algorithm, using (4), may be summarized as

1. Initialization: each processor receives ω and the set of rows $r_p{:}q_p$ of the matrix A and the elements $r_p{:}q_p$) of the vector b;
2. Iteration: each server processor p receives $x^{(k)}$, updates it and returns $x^{(k+1)}_{r_p:q_p}$ to the Master processor, which tests for convergence.

3.3 *Successive Overrelaxation (SOR)*

Given (1), the Successive Overrelaxation (SOR) method may be written as

$$x^{(k+1)} = \omega D^{-1}(b - Lx^{(k+1)} - Rx^{(k)}) + (1-\omega)x^{(k)} \tag{5}$$

where D, L and R are, respectively, the diagonal, left- and right-triangular matrices of A, and ω is the relaxation parameter. When $\omega = 1$, SOR reduces to the Gauss-Seidel method.

3.4 *Parallel version of the SOR*

The SOR method converges if, for a given value of ω, the spectral radius of the iteration matrix $\mathcal{L}_\omega = (I + \omega D^{-1}L)^{-1}((1-\omega)I - \omega D^{-1}R) < 1$ (see [4]). Unfortunately, there is a data-dependency between $x^{(k)}$ and $x^{(k+1)}$, imposing a low level of parallelism. A parallel implementation of (5), as it is, would require some processors to wait while values of $x^{(k+1)}$ are computed.

To overcome this problem, we rewrite (5) to obtain a suitable iteration to parallelize. After expanding and collecting terms, we have

$$(I + \omega D^{-1}L)x^{(k+1)} = \Big((1-\omega)I - \omega D^{-1}R\Big)\, x^{(k)} + \omega D^{-1}b. \tag{6}$$

Let

$$y = x^{(k+1)},$$
$$z^{(k)} = ((1-\omega)I - \omega D^{-1}R)x^{(k)} + \omega D^{-1}b.$$

Then (6) can be written as

$$(I + \omega D^{-1}L)y = z^{(k)}. \tag{7}$$

System (7) is a lower triangular system with unit diagonal, which can be solved by a forward-substitution scheme,

$$y_i = z_i^{(k)} - \sum_{j=1}^{i-1} a_{ij}y_j, \quad i = 0, 1, \ldots n. \tag{8}$$

The modified SOR method can then be stated as

1. $z^{(k)} \leftarrow ((1-\omega)I - \omega D^{-1}R)\,x^{(k)} + \omega D^{-1}b$,
2. Solve the system $(I + \omega D^{-1}L)y = z^{(k)}$ using (8),
3. $x^{(k+1)} \leftarrow y$,
4. Repeat steps 1-3 until convergence of x is achieved.

On parallelizing the above algorithm, we note that step 1 may be distributed (in a rowwise fashion), among the available processors in a similar fashion to the method used in section 3.2 for the JOR method. The solution of the lower triangular system, in step 2, is performed sequentially.

Using the notation introduced in 3.2, we can rewrite step 1 as

1. $z^{(k)}_{r_p:q_p} \leftarrow \left((1-\omega)I_{r_p:q_p,*} - \omega D^{-1}_{r_p:q_p,*}R_{*,r_p:q_p}\right)x^{(k)} + \omega D^{-1}_{r_p:q_p,*}b$

for $p = 1, 2, \ldots, P$, where the $*$ denotes the full rows or columns, accordingly.

3.5 Convergence test and parallelism

The convergence test is based upon the computation of the Euclidean norm,

$$\| u \|_2 = \sqrt{\sum_{i=1}^{n} u_i^2}$$

Our parallel implementation allows us to partition the computation of this norm among the processors using partial sums, i.e.,

$$S_p = \sum_{i=r_p}^{q_p} u_i^2, \quad p = 1, 2, \ldots, P, \tag{9}$$

$$\| u \|_2 = \sqrt{\sum_{p=1}^{P} S_p} \tag{10}$$

The calculation of the norm of y is straightforward. To compute the norm of the x vector, we recall that the y vector is an approximation to $x^{(k+1)}$. The vector $x^{(k)}$ is present in each processor, as it is used to compute $z^{(k)}_{r_p:q_p}$. Thus, we can compute the partial sums using the vectors $y_{r_p:q_p}$ and $x^{(k)}_{r_p:q_p}$.

3.6 *Adaptive version of the SOR*

The rate of convergence of the SOR algorithm, for a given problem, is controlled by the value chosen for ω. The optimal value of ω, denoted by ω_b, is given in [9] and [4] as

$$\omega_b = \frac{2}{1+\sqrt{1-\lambda^2}} \tag{11}$$

where λ is the spectral radius of the associated Jacobi iteration matrix, $D^{-1}(L+R)$, provided that

1. the partitioning of A is consistently ordered *and*
2. the associated Jacobi iteration matrix is symmetrizable.

If conditions 1 and 2 above hold, then the adaptive version of the SOR converges faster than the SOR method with $\omega = 1$ (see [4]).

Our parallel version of the SOR was modified to incorporate the adaptive procedure proposed in [4]. This is possible because the rowwise partitioning being used in our algorithms ensures condition *1* above is satisfied. We denote this version by "A-SOR". As the value of ω is modified during the iteration, we modified our parallel version of the SOR algorithm in order to allow the value of $z^{(k)}$ (cf. 3.4) to be recomputed by each of the server processors upon receiving a new value of ω from the master processor.

4 Results

4.1 *Topology issues*

The processors were interconnected in a ring. This interconnection is similar to a pipeline [5], although data generated by a server processor is used only by the master processor (like a SIMD processor).

4.2 *Model problems*

The JOR, SOR and A-SOR implementations were tested using two model problems, one dense and the other sparse. The sparsity of a problem is an important issue on the performance of the SOR and A-SOR. We used a convergence tolerance of 10^{-10} in all the tests.

For the dense system, the matrix A and the vector b are defined as

$$\begin{aligned} a_{ij} &= \begin{cases} 0.6 \times n^2 & \text{if } i = j \\ n/2 & \text{otherwise} \end{cases}, \\ b_i &= n, \quad i = 1, 2, \ldots, n \end{aligned} \tag{12}$$

where n is the size of the matrix A.

Test matrices of order 100, 200 and 400 were used. The value of ω was chosen to be 1. The number of iterations needed by the JOR for convergence for each of these matrices was, respectively, 100, 99 and 96, and for the SOR, 13, 13 and 12.

The comparison between the parallel SOR, with fixed ω $(= 1)$ and parallel A-SOR was performed using the system generated by a simple finite-difference approximation to Laplace's equation, with Dirichlet boundary conditions [9],

$$A = \begin{bmatrix} B & I & & & \\ I & B & I & & \\ & \ddots & \ddots & \ddots & \\ & & I & B & I \\ & & & I & B \end{bmatrix}, B = \begin{bmatrix} 4 & -1 & & & \\ -1 & 4 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 4 & -1 \\ & & & -1 & 4 \end{bmatrix}, b = \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \\ 1 \end{bmatrix}. \quad (13)$$

System (13) was solved for mesh sizes of 16×16 and 32×32, i.e., the order of the resultant linear system was 256 and 1024 respectively. The corresponding number of iterations was 357 and 429 for the SOR, and 104 and 134 for the A-SOR.

The optimal value of w calculated by the A-SOR for system (13) were 1.4560772659 and 1.4560770641, which are within 10^{-6} of the value given by (11).

4.3 Number of processors

Each algorithm was run with different numbers of processors in the ring, namely 2, 4, 8 and 16. In the tables that follow, "Serial" refers to a serial implementation of the algorithm in Occam, running on a single processor.

4.4 Results of the parallel JOR algorithm

Using the system of equations (13), the execution times (in seconds) and the respective speed-ups for the parallel JOR algorithm are shown in Tables 1 and 2. From Table 2, we note that our parallel implementation presents the expected behaviour, achieving almost linear speed-up with respect to the number of processors, when

- the number of iterations is sufficiently large to keep the ring working, as with 2 and 4 processors, and
- the size of the matrix is sufficiently large.

4.5 Results of the parallel SOR algorithm

Tables 3 and 4 present the execution time (in seconds) and the speed-up values for the parallel SOR algorithm when run using the system of equations (13). The speed-up values for our parallel SOR algorithm show a peak at around sixteen processors; adding more processors

# P	100×100	200×200	400×400
Serial	4.7340	18.3928	70.6777
2	2.8214	10.9145	41.8543
4	1.4667	5.5470	21.0826
8	0.8508	2.8835	10.7274
16	0.8147	2.1304	6.3782

Table 1: Execution times (in seconds) of the JOR.

# P	100×100	200×200	400×400
2	1.6779	1.6852	1.6887
4	3.2277	3.3158	3.3524
8	5.5641	6.3786	6.5885
16	5.8111	8.6334	11.0811

Table 2: Speed-up of the JOR.

# P	100×100	200×200	400×400
Serial	0.6877	2.7116	9.9473
2	0.7129	2.7847	10.2578
4	0.5978	2.3208	8.5296
8	0.5336	2.0309	7.4376
16	0.5313	1.8963	6.8550

Table 3: Execution times (in seconds) of SOR.

to the ring is then unproductive due to communications overheads. It should be noted that the optimum number of processors is problem dependent and is linked to the sparsity of the coefficient matrix.

4.6 Comparison between JOR and SOR

The JOR implementation is fully distributable among p processors, and the greater the size of the problem, the better it performs. However, it converges more slowly than the SOR method. For the JOR, we note that *the larger the system and the higher the number of iterations, the better it performs*, because the amount of computation required on the data must take longer than the data transfers. Our parallel implementation of the JOR scales very well, with respect to the number of processors, according to the size of the problem. Although the largest matrix we could use was of order 400, due to the memory restrictions, we can expect that, for a much larger problem, it would be possible to use more processors efficiently.

The SOR implementation, even with fixed $w = 1$, generally converges in less iterations than the JOR; however, the parallel version does not present the same scalability as the paral-

# P	100×100	200×200	400×400
2	0.9647	0.9737	0.9697
4	1.1504	1.1684	1.1662
8	1.2888	1.3352	1.3338
16	1.2944	1.4299	1.4511

Table 4: Speed-up of SOR.

# P	256	1024
Serial	13.9148	54.8243
2	8.9197	32.8245
4	3.9307	15.6049
8	1.9299	7.6588
16	1.4375	5.5344

Table 5: Execution times (in seconds) of SOR.

lel JOR does. This is because the triangular system (7) is dense for this example and is solved sequentially at each iteration on a single processor. Thus, as the problem size increases, the other processors remain idle for longer. This situation changes when the problem is sparse.

4.7 Results of the parallel A-SOR

The parallel versions of the SOR and A-SOR were used to solve the system (13), for systems of order 256, 1024. The results are presented in Tables 5 and 6.

The A-SOR is about three times faster than the SOR, and this improvement is due to the reduction in the number of iterations obtained from using an increasingly optimal value of ω. This more than offsets the additional overhead in communication between the processors when ω is changed.

Tables 7 and 8 shows the efficiency, as defined in [8], of the SOR and A-SOR methods for solving system (13). Although the A-SOR is less efficient than the SOR (because the adaptive part of the algorithm is run on just one processor), it is able to compute the solution in less time.

The solution of a sparse system such as (13) improves the performance of our parallel SOR and A-SOR implementations compared to a dense problem. The sparsity of (13) of orders 256 and 1024 is 1.9% and 0.48%, respectively, and this leads to a significant reduction in the amount of time used in sequential computation to solve the triangular system (7). We need to strike a balance between the amount of computation each processor is required to perform and the amount of data it needs to communicate. Thus, the efficiency factors start to decrease for more than about eight processors when the order of (13) is increased further. Here the *sparsity pattern* is dictating how many processors can be used effectively.

# P	256	1024
Serial	4.3093	16.9787
2	3.0347	10.1689
4	1.2348	4.8572
8	0.6640	2.3891
16	0.4993	1.8453

Table 6: Execution times (in seconds) of A-SOR.

# P	256	1024
2	0.7800	0.8351
4	0.8850	0.8783
8	0.9013	0.8948
16	0.6050	0.6191

Table 7: Efficiency (E_p) of SOR.

# P	256	1024
2	0.7100	0.8348
4	0.8725	0.8739
8	0.8112	0.7341
16	0.5394	0.5751

Table 8: Efficiency (E_p) of A-SOR.

5 Comparison with other implementations

In this section, we compare the performance of SOR and A-SOR, with the parallel implementations due to Evans [2] and Bonomo and Dyksen [1]. These implementations use the same Laplace's equation with Dirichlet boundary conditions (13) as a model problem, although the approach used to parallelize the methods is very different to ours.

Evans uses the red-black partitioning (see [4]) to subdivide the original system into smaller problems which may be executed in parallel. The processors use shared memory and synchronization flags to control the update of the variables. The hardware used is the NEPTUNE system, a four processor partial MIMD machine [2].

Bonomo and Dyksen use a shared-memory multiprocessor machine, the Symmetry computer, with 28 Intel 80386 processors. They use a synchronization scheme similar to the one used by Evans.

Table 9 shows the efficiencies of SOR and A-SOR for problem (13) of order 256, with four

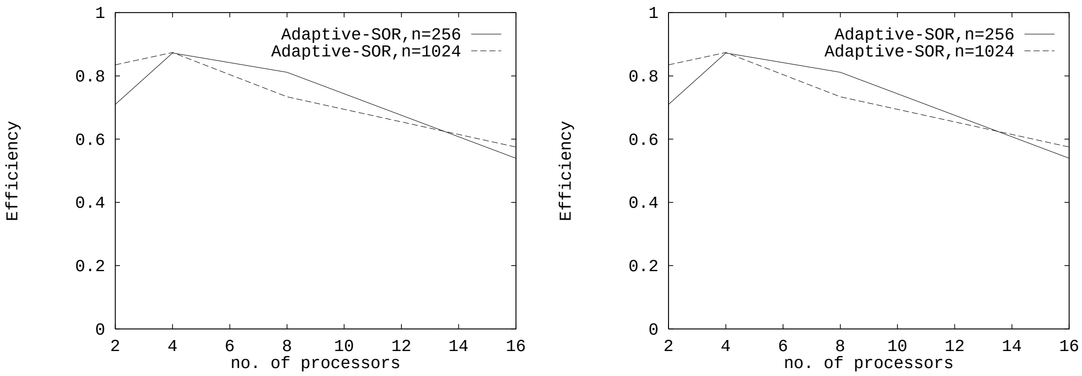

Figure 1: Efficiency graphs of SOR and A-SOR, $n = 256$ and $n = 1024$

Implementation	E_p
Evans ($\omega = 1$)	0.9859
SOR	0.8864
Evans ($\omega = 1.61$)	0.8864
A-SOR	0.8697

Table 9: Comparison between the implementations of Evans and da Cunha.

processors. The values for the Evans implementation on the NEPTUNE system are taken from [2], with $p = 4$. The values for our implementation were obtained using four processors (cf. Tables 5 and 6). Although the efficiencies presented by Evans' implementations are better than ours, our implementations run faster than those of Evans.

Table 10 compares the efficiencies of our SOR and A-SOR implementations with the PiSOR algorithm proposed by Bonomo and Dyksen. These results are also for the model problem (13). We have listed the best results obtained by Bonomo and Dyksen (B&D) when PiSOR is run with 16 processors; our implementations were run with the same number of processors.

Our algorithms performed better than the PiSOR as the system size increased.

6 Conclusion

We have presented parallel implementations of the JOR and SOR methods to solve systems of linear equations on transputer networks. For the JOR method the results show a high degree of efficiency for dense systems. The SOR methods are far more effective on sparse systems than on dense systems due to the bottleneck of having to solve triangular systems of equations on a single processor. We are currently developing a distributed triangular equation solver and first results indicate that integrating this into the SOR codes will provide an improved performance both on sparse and, more especially, on dense systems.

When solving sparse systems choosing the optimal number of processors requires balancing the amounts of computation and communication performed by each processor; we have not considered how this should be done for arbitrary sparsity patterns. Certainly for a fixed sparsity pattern, like (13), increasing the order does not mean that we can effectively use more processors.

n	*B&D*	*SOR*	*A-SOR*
20	0.5832	0.5446	0.5327
40	0.7343	0.6270	0.6235
60	0.7869	0.7715	0.7516
80	0.8352	0.8515	0.8388
100	0.8522	0.8696	0.8664

Table 10: Efficiencies of PiSOR, SOR and A-SOR.

7 Acknowledgements

We would like to thank the referees for their comments which helped improve this paper.

References

[1] J.P. Bonomo and W.R. Dyksen. Pipelined iterative methods for shared memory machines. *Parallel Computing*, 11:187–199, 1989.

[2] D.J. Evans. Parallel SOR iterative methods. *Parallel Computing*, 1:3–18, 1984.

[3] A. Hadjidimos. A survey of the iterative methods for the solution of linear systems by extrapolation, relaxation and other techniques. *Journal of Computational and Applied Mathematics*, 20:37–51, 1987.

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

[5] K. Hwang and F.A. Briggs. *Computer Architecture and Parallel Processing*. McGraw-Hill, New York, 1984.

[6] W. Niethammer. The SOR method on parallel computers. *Numerische Mathematik*, 56:247–254, 1989.

[7] N.R. Patel and H.F. Jordan. A parallelized point rowwise successive over-relaxation method on a multiprocessor. *Parallel Computing*, 1:207–222, 1984.

[8] U. Schendel. *Introduction to Numerical Methods for Parallel Computers*. Ellis Horwood, Chichester, 1984.

[9] D.M. Young. *Iterative Solution of Large Linear Systems*. Academic Press, New York, 1971.