Parallel Versions of Stone's Strongly Implicit Algorithm

J.S. Reeve*, A.D. Scurr and J.H. Merlin
Department of Electronics and Computer Science
University of Southampton
Southampton SO17 1BJ, UK
email jsr@ecs.soton.ac.uk

February 21, 2001

Abstract

In this paper, we describe various methods of deriving a parallel version of Stone's Strongly Implicit Procedure (SIP) for solving sparse linear equations arising from finite difference approximation to partial differential equations (PDE's). Sequential versions of this algorithm have been very successful in solving semi-conductor, heat conduction and flow simulation problems and an efficient parallel version would enable much larger simulations to be run. An initial investigation of various parallelising strategies was undertaken using a version of High Performance Fortran (HPF) and the best methods were reprogrammed using the MPI message passing libraries for increased efficiency. Early attempts concentrated on developing a parallel version of the characteristic wavefront computation pattern of the existing sequential SIP code. However, a red-black ordering of grid points, similar to that used in parallel versions of the Gauss-Seidel algorithm, is shown to be far more efficient. The results of both the wavefront and red-black MPI based algorithms are reported for various size problems and number of processors on a sixteen node IBM SP2.

1 Introduction

Our aim is to develop a parallel version of the SIP algorithm[12] capable of effectively solving problems that would otherwise be intractable on a single processor. The SIP method is an iterative method, although as subsequent sections will show, it has much in common with direct methods with its emphasis on the efficient factorisation of the original matrix. Stone attributes SIP's higher convergence rate to the more strongly implicit nature of SIP against the Alternating Direction Implicit (ADI) methods, which in turn are more strongly implicit methods than SOR and point-Jacobi[12, 1, 8]. The SIP method is used extensively in computational fluid dynamics, heat conduction and semiconductor device simulators, and is a very efficient solver of discrete systems derived

*Corresponding Author

from Poisson's equation[1, 3, 10, 12]. This paper describes attempts to parallelise the SIP solution algorithm of an existing semiconductor device simulation program. This simulation program uses the basic semi-conductor equations, but simplifies the problem by assuming Poisson's equation is solved (i.e. the electric field is assumed to be unaltered by the minority carrier flow) and therefore solves the remaining Drift-Diffusion continuity equations for electrons and holes[10]. These parabolic partial DE's are discretised using a five-point mid-point finite difference scheme[3, 4, 10, 11] and the resultant system of linear equations solved by a sequential SIP algorithm described below.

2 The Strongly Implicit (SIP) algorithm

Given a rectangular grid with natural ordering of points (i, j) as shown below [7],
a finite difference approximation of sides $I+2$ and $J+2$ (where $i = 2, \ldots. I+1$

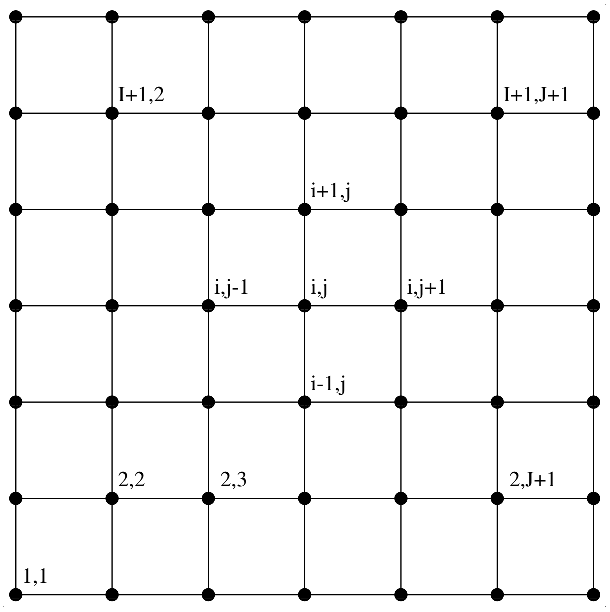

Figure 1: The finite difference stencil

and $j = 2, \ldots. J+1$) generates one linear equation for each grid point (i, j), giving $K = (I+1)(J+1)$ linear equations in total. The basic SIP algorithm assumes a five-point difference approximation over a rectangular mesh for the partial differential equations. This results in an A matrix of pentadiagonal structure (tridiagonal matrix with two super-diagonals at a set distance $J+1$ from the main diagonal), with a bandwidth of $2(J+1)$ [12, 1, 11]. For large K (total mesh size) the standard LU decomposition fills in a very large number of the zeros between the super and tridiagonals. From the matrix form $Au = q$ the general equation at grid point (i, j) is:

$$B_{i,j}u_{i-1,j} + D_{i,j}u_{i,j-1} + E_{i,j}u_{i,j} + F_{i,j}u_{i,j+1} + H_{i,j}u_{i+1,j} = q_{i,j} \qquad (1)$$

The crux of the method is to modify the A matrix with a matrix N such that in factorising the $(A+N)$ matrix, the L and U resultant matrices each have only three non-zero elements in each row, thus avoiding matrix fill and achieving a limited computational build-up that approaches $\Theta(n)$.
The forms of the L and U matrices are shown below:-

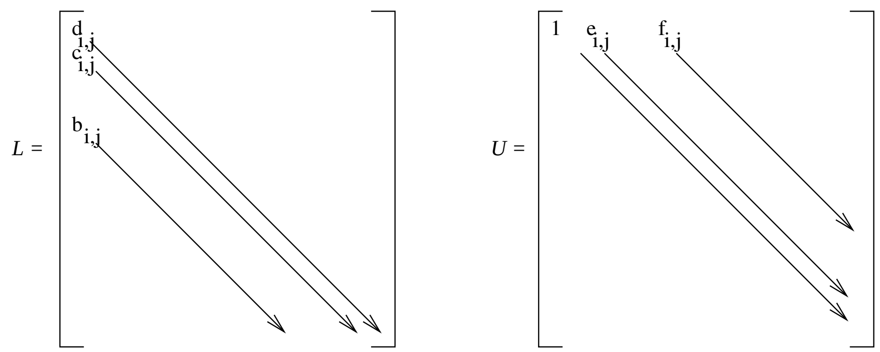

Figure 2: The form of the L and U matrices.

The nonzero elements of lower triangular matrix L are in diagonals $d_{i,j}$, $c_{i,j}$ and $b_{i,j}$ corresponding to the $B_{i,j}$, $D_{i,j}$ and $E_{i,j}$ diagonals of A. Similarly, the upper triangular matrix U has nonzero elements in diagonals corresponding to $E_{i,j}$, $F_{i,j}$ and $H_{i,j}$ diagonals of A , where the main diagonal corresponding to $E_{i,j}$ is unity. The product LU (the form of which is shown in figure 3) gives a matrix with seven nonzero diagonals, five corresponding to the diagonals (B, D, E, F, H) of A and two (C and G) which fall just inside the outer diagonals B and H, and which make up N. The algorithm is designed so that the terms Nu from the augmented matrix are small.

The complete set of equations relating LU to $(A+N)$ is:-

$$
\begin{aligned}
b_{i,j} &= B_{i,j} && (2)\\
c_{i,j} &= D_{i,j} && (3)\\
d_{i,j} + b_{i,j} f_{i,j-1} + c_{i,j} e_{i-1,j} &= E_{i,j} && (4)\\
d_{i,j} e_{i-1,j} &= F_{i,j} && (5)\\
d_{i,j} f_{i,j-1} &= H_{i,j} && (6)\\
b_{i,j} e_{i,j-1} &= C_{i,j} && (7)\\
c_{i,j} f_{i-1,j} &= G_{i,j} && (8)
\end{aligned}
$$

For grid point (i,j), there are five unknowns $(b_{i,j}, c_{i,j}, d_{i,j}, e_{i,j}, f_{i,j})$ but seven equations and therefore the system of equations is over-determined with a family of possible solutions. The simplest definition of $(A+N)$ which can be factored into LU comes from ignoring equation7 and equation8 and solving the five variables $b_{i,j}$ to $f_{i,j}$ using equations 2...6 by elimination, starting with $b_{1,1}$ (all zero subscripted variables are zero).

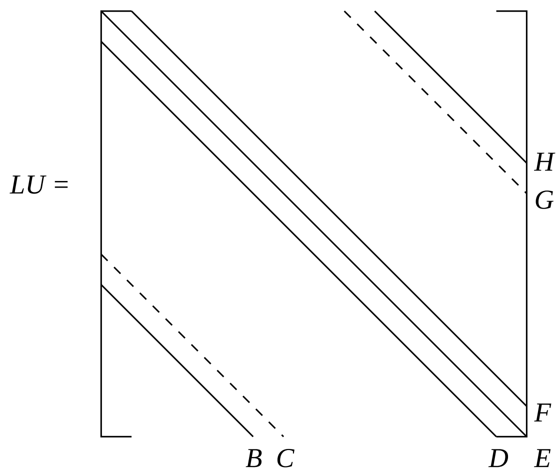

Figure 3: The form of the LU product matrix.

Stone[12], found that the resultant $(A+N)$ matrix did not give rapid convergence on his heat conduction test problems and re-defined the N matrix so that the $C_{i,j}$ and $G_{i,j}$ components are made smaller by subtracting a closely equivalent expression from them, using simple Taylor series expansions and an acceleration parameter $\alpha \ni 0 < \alpha < 1$ (which may be a function of the chosen grid). However, the sequential version of the SIP algorithm in the semi-conductor device simulator uses the simpler equations 2…6 to calculate the LU coefficients, with apparently satisfactory convergence properties for this particular problem. A SIP iteration can be defined from $Au = q$ by adding Nu to both sides of the equation and $(Au - Au)$ to the rhs giving:

$$(A + N)u^{(n+1)} = (A + N)u^{(n)} + (q - Au^{(n)}) \tag{9}$$

Substitute LU for $(A+N)$ in equation 9 and let $d^{(n)} = u^{(n+1)} - u^{(n)}$ and $R^{(n)} = (q - Au^{(n)})$, then a complete iteration cycle consists of solving: $LUd^{(n)} = R^{(n)}$ followed by $u^{(n+1)} = d^{(n)} + u^{(n)}$.
The SIP algorithm consists of four steps:-

1. An initial factorisation of the A-matrix into lower and upper triangular matrices L and U
2. A Forward substitution step $v^{(n)} = L^{-1}R^{(n)}$
3. A Backward substitution step $d^{(n)} = U^{-1}v^{(n)}$
4. A maximum relative error check and new solution value update $u^{(n+1)} = d^{(n)} + u^{(n)}$, plus update of residual $R^{(n)}$ from $(q - Au^{(n)})$

Steps 2 , 3 and 4 are repeated until the maximum relative difference between the old and new solution values $d^{(n)}$ are within the defined tolerance or the maximum number of iterations has been reached.

3 Analysis of the sequential SIP code

The SIP algorithm is contained in a Fortran 77 routine 'sip' which takes the A matrix, plus some problem definition parameters and produces a matrix u of the solution value at each grid node. The 'sip' routine has been highly optimised, both to minimise memory usage and to speed-up execution. The A matrix is stored as a $6 \times (I+2) \times (J+2)$ three-dimensional array, with each of the five non-zero coefficients per row plus the right-hand-side stored in the first dimension, for each grid point i, j. After some temporary array initialisation, the routine is divided into the three stages of initial LU factorisation, followed by iteration through the forward and backward substitution steps.

3.1 The LU factorisation

The code uses the simplest definition of the elements of L and U as defined in equations 2…6 of section 2 above. Only the main diagonal of the L matrix ($d_{i,j}$) is stored and the corresponding A matrix coefficients and $d_{i,j}$ elements are substituted for the remaining L and U elements as needed. Thus the code defines the factorisation step as storing the elements of $d_{i,j}$ from equation 4 above, giving:

$$d_{i,j} = E_{i,j} - B_{i,j}.H_{i,j-1}/d_{i,j-1} - D_{i,j}F_{i-1,j}/d_{i-1,j} \tag{10}$$

The reciprocal of the L diagonal $d_{i,j}$ is stored for the forward step in the array holding the original A matrix in place of $E_{i,j}$

3.2 The forward substitution step.

The forward substitution step is derived from the equation:

$$b_{i,j}V_{i,j-1} + c_{i,j}V_{i-1,j} + d_{i,j}V_{i,j} = R_{i,j}^{(n)} \tag{11}$$

where $b_{i,j}, c_{i,j}, d_{i,j}$ are the diagonals of L and $V_{i,j}$ is defined by $LV = R^{(n)}$ the residual at iteration n.
This gives the forward step of calculating the $V_{i,j}$ node array values from:-

$$V_{i,j} = (R_{i,j}^{(n)} - b_{i,j}V_{i,j-1} + c_{i,j}V_{i-1,j})/d_{i,j} \tag{12}$$

where $R_{i,j}^{(n)}$ is defined by:-

$$R_{i,j}^{(n)} = q_{i,j} - (B_{i,j}u_{i,j-1} + D_{i,j}u_{i-1,j} + E_{i,j}u_{i,j} + F_{i,j}u_{i+1,j} + H_{i,j}u_{i,j+1}) \tag{13}$$

(The u and V values are those for the nth iteration) The actual equation implemented is:-

$$\begin{aligned} V_{i,j} = q_{i,j} \quad & - \quad B_{i,j}V_{i,j-1}/d_{i,j-1} + D_{i,j}V_{i-1,j}/d_{i-1,j} \\ & - \quad B_{i,j}F_{i,j-1}u_{i+1,j-1}/d_{i,j-1} + D_{i,j}H_{i-1,j}u_{i-1,j+1}/d_{i-1,j} \end{aligned} \tag{14}$$

where $q_{i,j}$ is the original rhs element.

3.3 The backward substitution step.

From the basic equation $Ud^{(n)} = v^{(n)}$, where $v^{(n)}$ is calculated in the forward step above, the required equation is:-

$$u_{i,j} = V_{i,j} - e_{i,j}u_{i+1,j} - f_{i,j}u_{i,j+1} \tag{15}$$

Using $e_{i,j} = F_{i,j}/d_{i,j}$ and $f_{i,j} = H_{i,j}/d_{i,j}$ results in

$$u_{i,j} = V_{i,j} - (F_{i,j}/d_{i,j})u_{i+1,j} - (H_{i,j}/d_{i,j})u_{i,j+1} \tag{16}$$

which is the equation implemented.

3.4 The termination test

During the backward substitution step, a reduction is performed over the solution matrix u to find the maximum relative change in solution values. If this is less than or equal to an input parameter, or if the maximum number of iterations is exceeded, the SIP program terminates with the current u solution values.

4 Approaches to parallelising the SIP algorithm.

4.1 The dependency analysis

An analysis of the original sequential SIP code yields the following dependency graphs for steps 1. to 3.

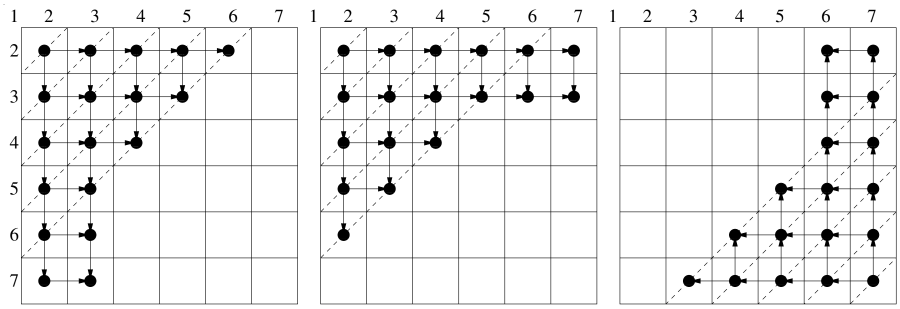

Figure 4: Step 1. Factorisation, 2. Forward Elimination, 3. Backward Substitution

Each iteration space dependency graph shows a wave-front diagonal pattern in which the next grid point value depends on the previous row and column grid point values. This form of dependency has been analysed by Wolfe[13] and others and is inherently sequential as neither the inner or outer do loops may be executed in parallel. The vertical arcs for the row index and the horizontal arcs for the column index show this in the dependency graphs above. However, the wavefront method creates a "wave" that passes through the iteration space, such that iterations on a single wavefront line (shown by successive dashed lines in figure 4) can be executed in parallel. Without support from particular language constructs (e.g. the FORALL construct in Fortran 95/HPF) parallel execution is achieved by skewing the index set of the original do loops creating a rhomboid iteration space out of the square iteration space. The columns of the rhomboid dependence graph now constitute the wavefront rather than the diagonals of the original graph, hence the elements in each wavefront column can be computed in parallel as there is no dependency (vertical arc) between them. SHPF, the Southampton University's version of HPF[6], was used to develop a prototype wavefront version of the SIP algorithm. In particular, it was used to find the best data distribution strategy i.e. row-wise verses block checkerboard partition, before re-implementation in MPI for increased efficiency.

4.2 Row-wise stripping

One matrix partitioning approach is to use row-wise block striping over a vertical vector of processors. The major problem with row-wise block striping for the wavefront method is that, as each column is partitioned between several processors, the value in the last row of each column must be communicated to the processor below (if there is one). Therefore, unless values are transmitted before the last processor column is reached, the algorithm becomes entirely sequential again. The problem, then, is to find the correct block size $k \ni 1 < k < J + 2$ of row values to send which minimises communication costs but which also gives a reasonable amount of parallelism over all processors.

4.3 Block checkerboard

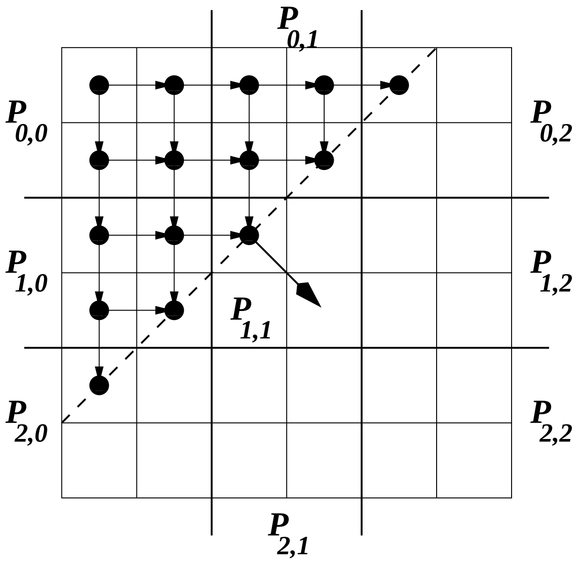

Figure 5: Block checkerboard partition

In general, 2D block checkerboard processor partitions tend to be more scalable (better area to perimeter ratio) than a 1D partition, such as row-wise stripping, unless the ratio $(I+2)/P$ (where P is the number of processors) is small. Note a 2D partition in the wavefront method involves the overhead of communicating with only two not four neighbours as in other methods such as the Jacobi algorithm. A 2D partition allows more control of the communications strategy, as it is essentially defined by the granularity of the matrix partition. Thus in the forward sweep, processor $P_{0,0}$ starts first and after completing the sub-matrix computations, would send the last row and column values to the below and right processor respectively. These processors would repeat the above procedure in parallel, gradually bringing in further processors as the computation sweeps through the grid to complete the forward step. The backward step is then computed in the reverse order of columns and processors to the forward step i.e a backward sweep through the processors.
The final step 4. involves a global reduction over all sub-matrices on all processors to find the maximum relative error. The matrix intrinsic function MAXVAL is used to determine the maximum absolute value of the relative error. This function, over a distributed array, takes at $\min(ts + tw)\log P[5]$ and so the code contains a parameter ITERCHECK (usually set to 10) which sets the iteration frequency with which the reduction operation is performed.

4.4 'Red-black' approach for SIP algorithm

The wave-front dependency graphs (figure 4) are similar to the graphs for Gauss-Seidel and other iterative methods. However as SIP is fully implicit, the current grid point value (i, j) only depends directly on the previous iteration values at $(i-1, j)$ and $(i, j-1)$. However, analysis of the Gauss-Seidel algorithm shows that the wave-front dependency arises out of the ordering of the problem equations, which in turn is determined by the order in which grid points are numbered. Thus different dependencies will arise from different equation orderings and hence grid point numbering schemes. The Gauss-Seidel algorithm and other methods can be effectively parallelised by renumbering the grid points based on red-black ordering, where alternate grid points in each row and column are numbered first (coloured red) in natural order followed by numbering all the remaining points (coloured black) in natural order (see [5] pages 430-432). This suggests that a similar 'red-black' strategy for the SIP algorithm should be possible and be capable of exhibiting more parallelism than wave-front based approaches.

5 Implementation of MPI versions of the parallel SIP algorithm.

5.1 Parallel wave-front algorithm

In the MPI implementation of the SIP algorithm extensive use is made of MPI communicators and topologies. This is both to simplify the code structure and to generalise it, so that any stripped or block partitioning scheme can be specified. Use is also made of the user defined data type facility for passing rows

of matrices from one processor to another. All problem parameters, Cartesian topology and communicator set-up (e.g. communicator names, processor grid ranks, neighbouring processor ranks, etc.) are all held together in a grid definition header file.
Most of the work in the MPI version was concerned with devising the correct message passing strategy. Analysis of the computation showed that the communications strategy had to follow the sweep through the matrix implied by the wave-front pattern. That is, computation starts at top left corner processor and sweeps to bottom right processor (assuming a 2D processor matrix) in the factorisation and forward SIP steps 1. and 2. described above. The preceding computational sweep follows the reverse course for the backward SIP step 3. Every process sub-matrix is defined to have a one row and column border all round for receiving data in both forward and backward sweep operations (see figure 6 below). This is necessary for the u matrix and simplifies indexing for other matrices.

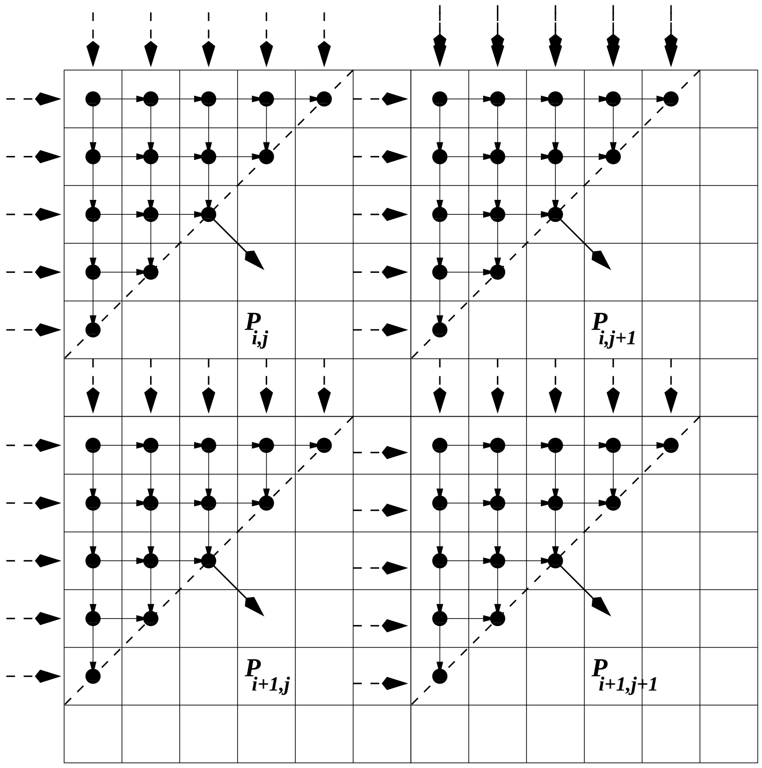

Figure 6: Communications involved in factorisation and forward sweep.

The communications pattern for forward (and factorisation) steps is shown in figure 6. At the start of each processor sub-matrix iteration, the input row and column data must be present from the two neighbouring processors before computation can start and sub-matrix computation must be completed before the output row and column transfers can be made. Thus a forward substitution step consists of:-

- A non-blocking MPI_IRECV into the first (i.e. border) row and column from the processes above and left respectively of the current process. A wait is made until both messages are received before computation can start.
- Computation of the array elements in a forward sweep starting with the top left element.
- A blocking send (MPI_SEND) of the last but one (last row/column is the border) matrix row and column to the process below and to the right respectively of the current process.

The backward sweep has the same order of steps, except 'first' becomes 'last', 'left' becomes 'right', 'top' becomes 'bottom' and 'above' becomes 'below' and vise versa.
One of the attractive features of MPI is that the above communications pattern can be specified in a general way and the problem of handling edge conditions (i.e. there may not be a process to send data to) is taken care of by MPI itself. In order to get the maximum efficiency in message passing, non-blocking receives are used with blocking sends. Combined with a wait on the first (last for backward sweep) row and column, a process is ready to receive either transmission as soon as it is ready to be sent and as soon as both messages are received, computation can start. Thus messages can be received in any order or even simultaneously. A similar attempt was made to use non-blocking sends with non-blocking receives, but was found to give an unsafe programme. One further complication is that, because of the dependencies on u in equation 15, all four edge columns and rows have to be communicated to neighbours at the end of the backward iteration step. Processes are synchronised at specific points in the SIP algorithm using the MPI_BARRIER function. These coincide with the algorithmic steps at the start of factorisation, at the end of the forward step and the backward step. The maximum relative error is calculated on each process during the backward iteration step. MPI_ALLREDUCE function is used for the global determination of the maximum relative value over all processes, so that the maximum value is available on each process for the next iteration.

5.2 'Red-black' algorithm

The red-black version of the parallel SIP algorithm in MPI uses the same LU factorisation code as section 5.1, as the computation order dependencies in this step are such that red-black ordering doesn't work. However, the forward and backward substitution steps are amenable to red-black ordering. These steps are implemented in a two-pass iteration scheme where all the even grid points are computed followed by all the odd points, similar to the SOR implementation in[9]. The increased efficiency of this method over the wave-front method above is because half the grid point values can be computed on each pass concurrently over all processors. This parallel computation is followed by an exchange of row and column data between processes. Thus a forward substitution step consists of two passes (odd and even) of:-

- Computation of half the array elements of the residual matrix $R_{i,j}^{(n)}$ from equation 13 followed by calculation of the equivalent elements of matrix $V_{i,j}$ from equation 12.

- Transfer of the last but one matrix row and column of $R_{i,j}^{(n)}$ and $V_{i,j}$ to the process below and to the right respectively of the current process . Non-blocking receives are posted for the first (i.e. border) row and column from the processes above and left respectively of the current process before blocking sends are used of the last but one matrix row and column to the process below and to the right respectively.

Note, the above transfer pattern arises out of the dependencies in the equations and is the same as section 5.1 above, including the need to transfer all four edges of the u matrix. The backward step has a similar two-phase iteration structure, based on equation 15 but with the reverse computation and transfer structure. A complication of the red-black algorithm is the need to compute the starting column (forward step) or row (backward step) of the sub-matrix at the start of each pass, based on the distribution of the equivalent global matrix over all the processors.

6 Results

The following runs were carried out using up to 16 similar nodes on an IBM SP2. Problem sizes varied from 40×40 to 960×960 grids for I and J. The number of processors was chosen to give a square matrix array. The iteration check parameter was set at 10 for all runs and a least 500 iterations computed for each run. The execution time in seconds of ten iterations for different size problems and number of processors is given in the tables below. Graphs of the execution times, relative speedup and relative efficiency are shown below for the wavefront (table 1) and red-black (table 2) algorithms. Relative efficiency is defined as the fraction of execution time the processors spend doing useful work [2]. Thus

$$E_{relative} = \frac{\text{Execution time (secs) for 10 iterations on 1 processor}}{P * \text{Execution time (secs) for 10 iterations on } P \text{ processors}}$$

And relative speedup, $S_{relative} = P * E_{relative}$

6.1 MPI runs 'wavefront' algorithm

Execution time for ten iterations for various numbers of processors (secs), followed by relative speedup and relative efficiency computed for each case are shown in table 1 and figure 7.

6.2 MPI runs 'red-black' algorithm

Execution time for ten iterations for various numbers of processors (secs) and computed relative speedup and efficiency for each case are shown in table 2 and figure 8

The graphs clearly show that the wavefront algorithm yields only a modest speedup with increasing number of processors arranged in a block chequer board array. A maximum speedup of 2.8 was obtained with 16 processors on the 480 problem. Thus relative efficiency decreases with increasing number of processors. However, except for the smallest problem where communication overheads

Size	Execution time				$S_{relative}$			$E_{relative}$		
	Number of Processors				Number of Procs			Number of Procs		
$n \times n$	1	4	9	16	4	9	16	4	9	16
40	0.1	0.084	0.074	0.075	1.19	1.35	1.33	0.3	0.15	0.08
240	3.84	2.58	1.93	1.54	1.49	1.99	2.49	0.37	0.22	0.16
480	16.73	11.15	7.62	5.96	1.50	2.20	2.81	0.38	0.24	0.18
720	38.18	27.49	18.65	14.01	1.39	2.05	2.73	0.35	0.23	0.17
960	67.8	49.05	34.59	26.11	1.38	1.96	2.60	0.35	0.22	0.16

Table 1: The Wavefront method timings

Size	Execution time				$S_{relative}$			$E_{relative}$		
	Number of Processors				Number of Procs			Number of Procs		
$n \times n$	1	4	9	16	4	9	16	4	9	16
40	0.074	0.038	0.036	0.035	1.95	2.06	2.11	0.49	0.23	0.13
240	3.43	0.65	0.38	0.21	5.28	9.03	16.33	1.32	1.00	1.02
480	16.65	3.45	1.23	0.70	4.83	13.54	23.79	1.21	1.50	1.49
720	37.76	9.66	3.66	1.88	3.91	10.32	20.09	0.98	1.15	1.26
960	66.01	17.69	7.48	3.59	3.73	8.82	18.39	0.93	0.98	1.15

Table 2: The Red-black method timings

predominate, the red-black algorithm gives linear speedup with increasing number of processors and hence maintains a relative efficiency of 1.

7 Summary

The main project goal was to develop a parallel version of the SIP algorithm capable of efficiently solving larger problems. A successful but rather inefficient parallel version of the SIP algorithm has been developed in MPI and in SHPF. This version is based on the existing sequential algorithm and parallelises the forward and backward wavefronts over a block chequer board array of processors (with row or column striping as an option). It has been used to solve problems up to 960 $\times$ 960 grid elements and up to 16 processors on the SP2. Performance figures are given above. The poor relative speedup (a maximum of 2.8 achieved for the 480×480 problem) is mostly due to the wave-front nature of the method i.e. the inherently sequential way in which the computation flows from one processor to adjacent processors and the consequent poor load balancing obtained.

Substantially more parallelism is achieved using the red-black ordering method, as all processors can update half their values simultaneously in each pass before exchanging edge values. It was not possible to change the factorisation step from a wavefront based method to a red-black algorithm because of the ordering and dependencies inherent in the factorisation equations. However, the wave-front algorithm is satisfactory as the factorisation of the (A + N) matrix is only performed once. For the forward and backward iteration steps, the wave-front method is replaced by a two pass simultaneous computation on all processors.

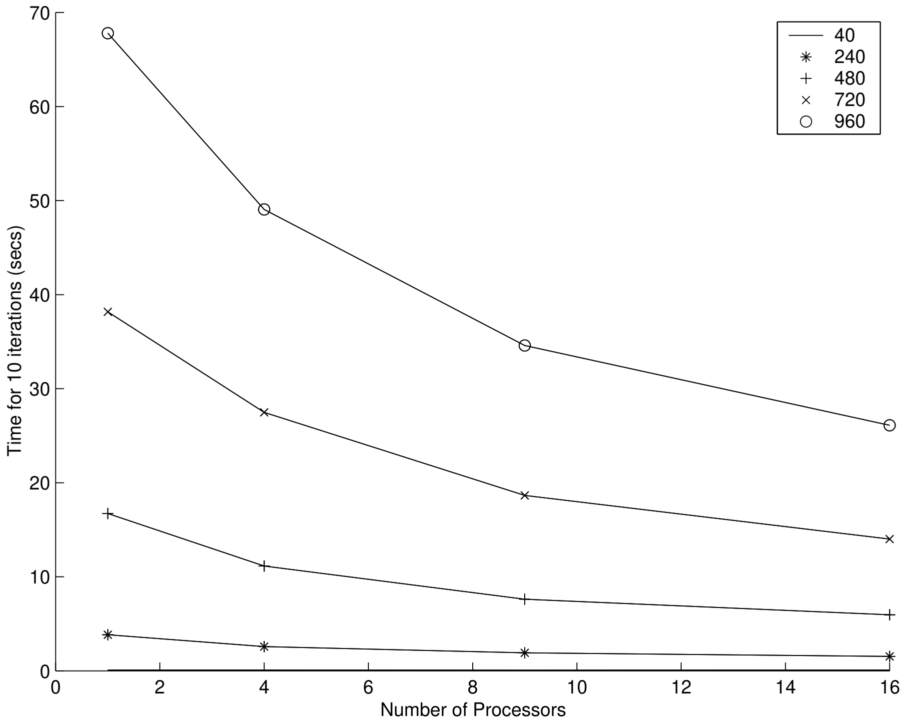

Figure 7: The wavefront method execution times

Results from running various size problems using an MPI version of 'red-black' ordering are shown in table 2. Maximum speedup obtained is approximately 24 on 16 processors, which is roughly 8 times better than the equivalent wave-front version in MPI. Super-linear speed-up is obtained for certain sized problems due to more effective cache utilisation and should be ignored. Thus, using a red-black algorithm for the forward and backward steps of the main iteration loop provides linear speed-up with increasing numbers of processors, except for very small problems where communication overheads dominate.

7.1 Acknowledgement

The authors wish to express their gratitude to Dr. J.S. Hamel of the Electronics and Computer Science department of the University of Southampton for providing the semiconductor simulation programme and for his help and advise on its use.

References

[1] D.A. Anderson, J.C. Tannehill, and R.H. Pletcher. *Computational Fluid Mechanics & Heat Transfer*. McGraw-Hill, 1984.

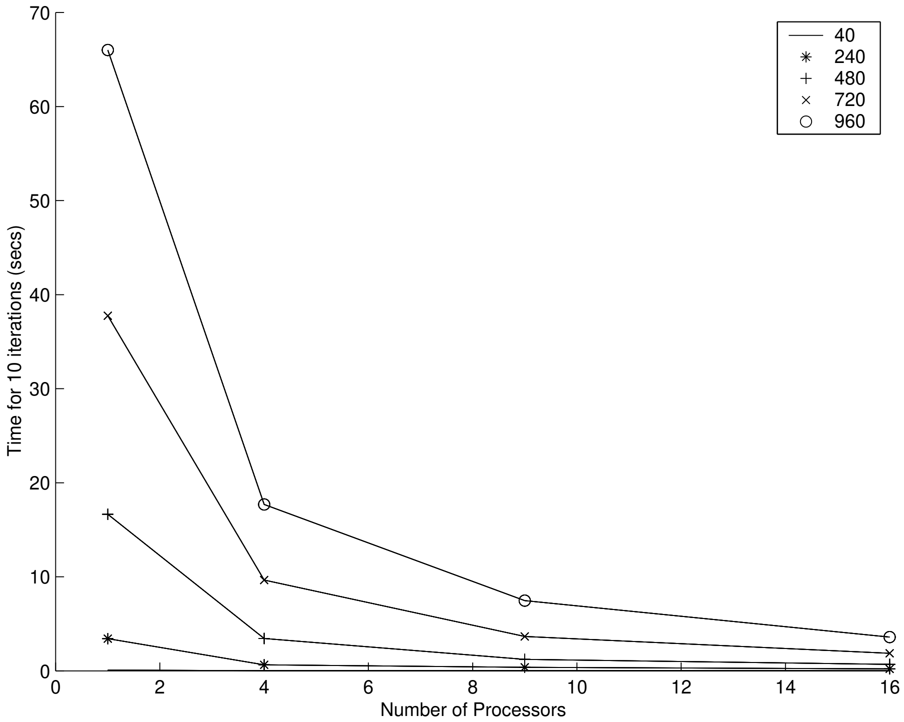

Figure 8: The red-black method execution times

[2] I.T. Foster. *Designing and Building Parallel Programs.* Addison-Wesley, 1994.

[3] C.R. Jesshope. *Bipolar transistor modelling.* Ph.d thesis, Department of Electronics, University of Southampton, 1976.

[4] C.R. Jesshope. SIPSOL: a suite of subprograms for the solution of the linear equations arising from elliptic partial differential equations. *Computer Physics Communications*, 17, 1979.

[5] V. Kumar, A. Grama, A Gupta, and G. Karypis. *Introduction to Parallel Computing.* Benjamin/Cummings, 1994.

[6] J. Merlin and A.J. Hey. An introduction to high performance fortran. *Scientific Programming*, 4:87–113, 1995.

[7] Ortega and Poole. *An Introduction to Numerical Methods for Differential Equations.* Pitman, 1981.

[8] W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling. *Numerical Recipes in Fortran.* Cambridge, 1989.

[9] W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling. *Numerical Recipes in Fortran 90*, volume 2. Cambridge, 1996.

[10] S. Selberherr. *Analysis and Simulation of Semiconductor Devices.* Springer-Verlag, 1984.

[11] G.D. Smith. *Numerical Solution of Partial Differential Equations: Finite Difference Methods.* Oxford: Claredon Press, 1985.

[12] H.L. Stone. Iterative solution of implicit approximations of multidimensional partial differential equations. *SIAM J. Numerical Analysis*, 5(3), 1968.

[13] M. Wolfe. Loop skewing: The wavefront method revisited. *International Journal of Parallel Programming*, 15(4), 1986.