Instituto de Ciências Matemáticas de São Carlos

ISSN - 0103-2569

IMPLEMENTATION OF A DIRECT POISSON SOLVER ON THE KSR1.

ARMANDO DE O. FORTUNA

Nº 50

RELATÓRIOS TÉCNICOS DO ICMSC

São Carlos
Dez./1996

Implementation of a Direct Poisson Solver on the KSR1.

Armando de O. Fortuna
Department of Computer Science
University of Manchester
Oxford Road
Manchester M13 9PL

Abstract

A direct solver for the five-point finite difference approximation to the Poisson equation on a rectangular grid is described. The suitability of the solver for parallel implementation is analysed, and the results in terms of execution times and accuracy are compared against two iterative methods.

1. Introduction

Over past years, much work has been done in direct solution methods for elliptic equations and in particular, for the Poisson equation. The *Poisson equation*

$$-\nabla^2 u = f \qquad \textbf{1.I}$$

(where ∇^2 is the laplacian operator), frequently occurs in engineering problems such as heat conduction, incompressible fluid flow and electronic design [1] [2] [11]. In fact, it occurs so frequently that special techniques have been developed for its solution. Some of these algorithms include the solution of more general elliptic equations and domain partitioning for non-trivial geometries [2]-[13]. These methods are in general based on the fast Fourier transform (FFT) or are purely algebraic (*i.e.* cyclic reduction) [18] [19].

Compared to iterative methods, direct solution methods for the Poisson equation have smaller execution time and better accuracy. On the other hand, they require more computer storage, something that could be a real problem up to a few years ago. Today computers with large memory capacities have become available at affordable costs, thus practically eliminating memory as one of the constraints of program and data size. This allows one to explore the advantages of direct solution methods when necessary, such as in some fluid flow problems, where a Poisson equation for the pressure (or for the stream function) has to be solved at every time step. Direct methods then provide small execution times and increased accuracy [14].

The availability of parallel computers in recent years has placed a powerful tool in the hands of scientists and engineers. Direct solution algorithms which can be parallelised have thus become attractive, better exploiting the potential of these new parallel machines.

This report describes one FFT-based algorithm, the *Matrix Decomposition* (MD) [7] [13] and its implementation on a parallel computer, the Kendall Square 1 (KSR1), available at the University of Manchester [15]. While the parallelising technique employed is general enough so the code is portable to other computers of different architectures without difficulty, the performance figures of the code show great improvement when compared to commonly used iterative methods.

The rest of this report is divided as follows. Section 2. describes the MD algorithm. Sections 3. and 4. discuss the general features of the KSR1 and the implementation of a parallel version of the MD algorithm, respectively.

Section 5. describes the iterative methods used, and section 6. shows the result of the code applied to a sample problem and discusses the results obtained. The code which implements the MD algorithm is listed at the end of this report.

2. The Matrix Decomposition Algorithm

This implementation of the MD algorithm will focus on the solution of a 5-point discretisation of a Poisson equation in a $N \times M$ rectangle R, subject to Dirichlet boundary conditions (*i.e.* $u = g(x,y)$ on the boundary ∂R). The computational domain is shown in figure 1 below.

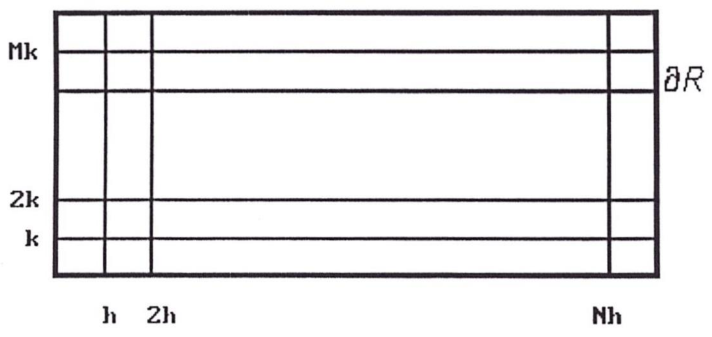


Figure 1
Domain where the solution of the Poisson equation is being computed.

The MD algorithm is more general than this implementation report would initially suggest, being applicable to all elliptic equations of the form

$$\frac{\partial}{\partial x}\left(a(x)\frac{\partial u}{\partial x}\right)+\frac{\partial}{\partial y}\left(b(y)\frac{\partial u}{\partial y}\right)+cu=f(x,y) \qquad \textbf{2.I}$$

on a rectangle R with Dirichlet, Neumann or periodic boundary conditions [13].

Approximating equation 1.I by the standard 5-point, second order finite difference formula, one has

$$(-\nabla^2 u)_{i,j}\approx\left(\frac{-u_{i-1,j}+2u_{i,j}-u_{i+1,j}}{h^2}\right)+\left(\frac{-u_{i,j+1}+2u_{i,j}-u_{i,j-1}}{k^2}\right) \qquad \textbf{2.II}$$

where $u_{i,j}\equiv u(ih,jk)$ and h and k are the spatial increments in the horizontal and vertical directions, respectively. Equation 2.II can be written as

$$f_{i,j}=-u_{i-1,j}+2u_{i,j}-u_{i+1,j}+\left(\frac{h}{k}\right)^2\left(-u_{i,j+1}+2u_{i,j}-u_{i,j-1}\right)$$

or

$$\alpha u_{i,j}-u_{i-1,j}-u_{i+1,j}-\beta^2 u_{i,j+1}-\beta^2 u_{i,j-1}=f_{i,j} \qquad \textbf{2.III}$$

with $\beta = \frac{h}{k}$ and $\alpha = 2(1+\beta^2)$.

Following [13], by writing the finite difference equation 2.III for all mesh points along a column $x = h$, one has

$$\begin{bmatrix} \alpha u_{11} & -\beta^2 u_{12} & & & \\ -\beta^2 u_{11} & \alpha u_{12} & -\beta^2 u_{13} & & \\ & \dots & & \dots & \\ & & -\beta^2 u_{1M-2} & \alpha u_{1M-1} & -\beta^2 u_{1M} \\ & & & -\beta^2 u_{1M-1} & \alpha u_{1M} \end{bmatrix} + \begin{bmatrix} -u_{21} & & & \\ & -u_{22} & & \\ & & \dots & \\ & & & -u_{2M-1} \\ & & & & -u_{2M} \end{bmatrix} = \tag{2.IV}$$

$$= \begin{bmatrix} f_{11} + \beta^2 g_{10} + g_{01} \\ f_{12} + g_{02} \\ \dots \\ f_{1M-1} + g_{0M-1} \\ f_{1M} + \beta^2 g_{1M+1} + g_{0M} \end{bmatrix}$$

Similarly, by writing the finite difference equation 2.III for all mesh points along the columns $x = Sh$, $S = 2, 3, \dots, N-1$:

$$\begin{bmatrix} -u_{S-1,1} & & & & \\ & -u_{S-1,2} & & & \\ & & \dots & & \\ & & & -u_{S-1,M-1} & \\ & & & & -u_{S-1,M} \end{bmatrix} + \begin{bmatrix} \alpha u_{S1} & -\beta^2 u_{S,2} & & & \\ -\beta^2 u_{S,1} & \alpha u_{S,2} & -\beta^2 u_{S,3} & & \\ & \dots & & \dots & \\ & & -\beta^2 u_{S,M-2} & \alpha u_{S,M-1} & -\beta^2 u_{S,M} \\ & & & -\beta^2 u_{S,M-1} & \alpha u_{S,M} \end{bmatrix} + \tag{2.V}$$

$$+ \begin{bmatrix} -u_{S+1,1} & & & & \\ & -u_{S+1,2} & & & \\ & & \dots & & \\ & & & -u_{S+1,M-1} & \\ & & & & -u_{S+1,M} \end{bmatrix} = \begin{bmatrix} f_{S,1} + \beta^2 g_{S,0} \\ f_{S,2} \\ \dots \\ f_{S,M-1} \\ f_{S,M} + \beta^2 g_{S,M+1} \end{bmatrix}$$

The equations for the last column $x = Nh$ are written as

2.VI

$$\begin{bmatrix} -u_{N-1,1} & & & & \\ & -u_{N-1,2} & & & \\ & & \cdots & & \\ & & & -u_{N-1,M-1} & \\ & & & & -u_{N-1,M} \end{bmatrix} + \begin{bmatrix} \alpha u_{N,1} & -\beta^2 u_{N,2} & & & \\ -\beta^2 u_{N,1} & \alpha u_{N,2} & -\beta^2 u_{N,3} & & \\ & \cdots & & \cdots & \\ & & -\beta^2 u_{N,M-2} & \alpha u_{N,M-1} & -\beta^2 u_{N,M} \\ & & & -\beta^2 u_{N,M-1} & \alpha u_{N,M} \end{bmatrix} =$$

$$= \begin{bmatrix} f_{N,1} + \beta^2 g_{N+1,0} + g_{N+1,1} \\ f_{N,2} + g_{N+1,2} \\ \cdots \\ f_{N,M-1} + g_{N+1,M-1} \\ f_{N,M} + \beta^2 g_{N+1,M+1} + g_{N+1,M+1} \end{bmatrix}$$

Defining $\boldsymbol{u_i}$ to be the M-dimensional vector

$$\boldsymbol{u_i} = \begin{bmatrix} u_{i1} \\ u_{i2} \\ . \\ . \\ . \\ u_{iM-1} \\ u_{iM} \end{bmatrix}$$

and $\boldsymbol{A}$ the M x M tridiagonal matrix

$$A = \begin{bmatrix} \alpha & -\beta^2 & & & \\ -\beta^2 & \alpha & -\beta^2 & & \\ & & \cdots & & \\ & & -\beta^2 & \alpha & -\beta^2 \\ & & & -\beta^2 & \alpha \end{bmatrix}$$

equations 2.IV, 2.V and 2.VI can be written as

$$\begin{gathered} \boldsymbol{A u_1 - u_2 = y_1} \\ \boldsymbol{-u_{S-1} + A u_S - u_{S+1} = y_S}, \quad S = 2, 3, \ldots, N-1 \\ \boldsymbol{-u_{N-1} + A u_N = y_N} \end{gathered} \qquad \textbf{2.VII}$$

with $\boldsymbol{y_i}$ being defined as the right hand side vectors of equations 2.IV, 2.V and 2.VI.

As the matrix $\boldsymbol{A}$ is symmetric, there exists a matrix $\boldsymbol{V}$ such that

2.VIII

$$V^T A V = \begin{bmatrix} \lambda_1 & & & & \\ & \lambda_2 & & & \\ & & \cdots & & \\ & & & \lambda_{M-1} & \\ & & & & \lambda_M \end{bmatrix} = D$$

with $VV^T = I$, where I is the identity matrix. The i-th column of V is the i-th eigenvector of A, with λ_i being the associated eigenvalue.

By multiplying each equation in 2.VII by V^T and inserting $I = VV^T$ on the left of each occurrence of A gives:

$$\begin{gathered} (V^T A V)(V^T u_1) - V^T u_2 = V^T y_1 \\ -V^T u_{S-1} + (V^T A V)(V^T u_S) - V^T u_{S+1} = V^T y_S, \quad S = 2, 3, \ldots, N-1 \\ -V^T u_{N-1} + (V^T A V)(V^T u_N) = V^T y_N \end{gathered}$$

or

$$\begin{gathered} Du_1^* - u_2^* = y_1^* \\ -u_{S-1}^* + Du_S^* - u_{S+1}^* = y_S^* \quad S = 2, 3, \ldots, N-1 \\ -u_{N-1}^* + Du_N^* = y_N^* \end{gathered} \qquad \textbf{2.IX}$$

where $u_i^* = V^T u_i$ and $y_i^* = V^T y_i$.

The system of equations 2.IX is now effectively *decoupled in a line-by-line basis* into M systems of linear equations. This can be seen by writing all the equations where λ_j, $1 \leq j \leq M$, occurs in 2.IX:

$$\begin{gathered} \lambda_j u_{1,j}^* - u_{2,j}^* = y_{1,j}^* \\ -u_{S,j}^* + \lambda_j u_{S,j}^* - u_{S+1,j}^* = y_{S,j}^* \quad S = 2, 3, \ldots, N-1 \\ -u_{N-1,j}^* + \lambda_j u_{N,j}^* = y_{N,j}^* \end{gathered} \qquad \textbf{2.X}$$

The system 2.X is tridiagonal and its unknowns depend only upon a single value of j. Hence it can be solved independently from the other $M - 1$ systems using the efficient *Thomas Algorithm* [16].

Once all the MN values of u^* have been computed, the values of unknowns u can be determined from:

$$u_i = Vu_i^*, \quad 1 \leq i \leq N \qquad \text{2.XI}$$

In summary, the MD algorithm can be broken into the following sequence of steps:

1) determine matrices $\boldsymbol{V}$ and $\boldsymbol{D}$ given matrix $\boldsymbol{A}$;

2) compute $\boldsymbol{y}_i$ and $\boldsymbol{y}_i^* = \boldsymbol{V}^T\boldsymbol{y}_i, \quad 1 \leq i \leq N$;

3) solve the tridiagonal system of equations 2.X for $j = 1, 2, ..., M-1, M$;

4) compute $\boldsymbol{u}_i = \boldsymbol{V}\boldsymbol{u}_i^*, \quad 1 \leq i \leq N$.

One can now see the potential for parallelisation of the algorithm. Given the matrices $\boldsymbol{V}$ and $\boldsymbol{D}$, all $\boldsymbol{y}_i^*$ can be computed in parallel as they are independent of each other. The same applies to the M tridiagonal systems 2.X and to the computation of $\boldsymbol{u}_i$. In this work, only steps (2)-(4) were implemented, as the elements of matrices $\boldsymbol{V}$ and $\boldsymbol{D}$ were computed analytically from:

$$V_{i,j} = \left[\frac{2}{M+1}\right]^{1/2} \sin\left(\frac{ij\pi}{M+1}\right)$$

and

$$\lambda_j = \alpha - 2\beta^2 \cos\left(\frac{j\pi}{M+1}\right)$$

as given in [13]. Thus no routines for the determination of matrices $\boldsymbol{V}$ and $\boldsymbol{D}$ given $\boldsymbol{A}$ were required.

The computational work of the MD algorithm is dominated by steps (2) and (4), where NM^2 multiplications are required at each of those steps. Thus it is desirable to have $M \leq N$. If $M > N$, then one can implement MD along the M *lines* of the computational domain, instead of along the N columns. Step (3) only requires about $O(N)$ multiplications.

The minimum storage requirements are NM entries for matrix $\boldsymbol{V}$, M for matrix $\boldsymbol{D}$, and NM for each of the vectors $\boldsymbol{u}^*$, $\boldsymbol{y}^*$, $\boldsymbol{u}$ and $\boldsymbol{y}$, totaling $5NM + M$ floating point values.

3. The Kendall Square Research 1

The Kendall Square Research 1 (KSR1) installed at the University of Manchester is a shared-memory, scalable parallel computer. It is a double-ring, 64-processing node machine. Each processing node has 32 Megabytes of local memory, a 500 Kilobyte sub-cache and has a processing unit rated at 40 Megaflops and 20 MIPS peak performance. Total installed memory is 2 Gigabytes.

The operating system KSR OS is based on the OSF/1 OS, which is compatible with Berkeley Standard Distribution (BSD) 4.3 & 4.4, and AT&T System V.2 and V.3.

Parallelisation of FORTRAN codes can be done automatically by the compiler through the use of specialised software tools. In general, though, it is more efficient for the programmer himself to determine parallelisation techniques, based on the algorithm employed, data structure and machine architecture.

The KSR1 *virtual shared memory architecture* provides the programmer with the familiar shared-memory programming model. This in itself enhances the portability of existing codes which originate from sequential machines.

Parallel codes for the KSR1 do not require user-written data communication procedures for exchanging data between processing nodes [15]. The memory architecture and the operating system handle that automatically, bringing data to a requesting node from any other node in the system. As in most other parallel systems, data communication should be reduced to a minimum, due to the communication latency between nodes (access time one order of magnitude greater than if the data were in the local node's memory). Therefore, data partitioning among processing nodes should be made wisely.

4. Description of the code

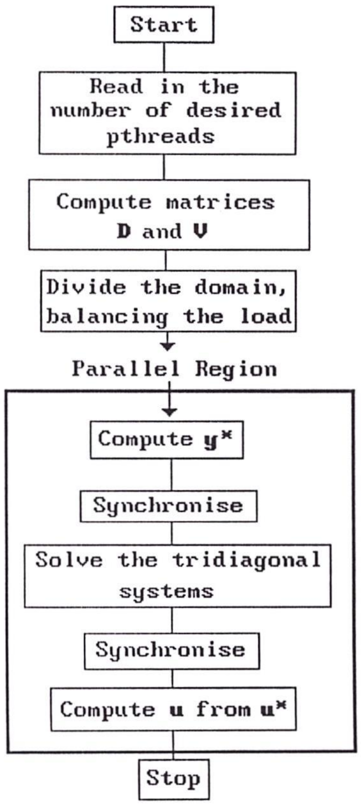
When implementing steps (2)-(4) on the KSR1, the main objective was to achieve load balance between pthreads with as small communication between processing nodes as possible. The chosen implementation strategy was the definition of a *parallel region*, which contains subroutine calls implementing steps (2)-(4). Proper synchronization between processes at the end of each step is enforced with the use of *barriers* [15].

Other parallelisation strategies, such as *parallel sections* and *tile statements* [15] were considered but discarded. *Parallel sections* do not allow for a *variable* number of threads to execute code in parallel, as the number of sections is *fixed.*

Tile statements could in principle, be used, and they would basically perform the same task as it has been implemented with parallel regions, but with slightly less flexibility as far as explicit synchronisation control by the user is concerned.

In all examples shown, only one pthread was being executed per processing node. This eliminates the overhead of context-switching within a single processing node, which on the KSR is rather high.

The block diagram of the parallel implementation of the MD algorithm is shown in figure 2 below.



Start
Read in the number of desired pthreads
Compute matrices D and V
Divide the domain, balancing the load
Parallel Region
Compute u*
Synchronise
Solve the tridiagonal systems
Synchronise
Compute u from u*
Stop

Figure 2

Block diagram of the implementation of the MD algorithm.

The code is written in KSR FORTRAN. Double precision is used for all floating point operations. The `-r8` compiler switch is set, forcing all double precision floating point numbers to be 64 bits long.

For the solution of a problem, the number of pthreads executing the parallel region is defined by the user at run time. By allocating the corresponding number of cells with the `allocate_cells` command, the execution of one pthread per allocated processor is enforced. The work is then divided as evenly as possible between the executing pthreads. This division implies allocating a fraction of the N multiplications from step (2) for the pthreads. This same strategy applies to step (3) and step (4). A listing of the code which implements the MD algorithm is provided in section 9.

5. Iterative methods

Two iterative methods for the solution of a linear system of equations were implemented using parallel constructs on the KSR1. Those methods were the Point Gauss-Siedel with Successive Over-Relaxation (PSOR) and Line Gauss-Siedel with Successive Over-Relaxation (LSOR) [17]. Those are commonly used iterative methods for the solution of the Poisson equation.

Starting from equation 2.III, these two methods can be written as:

$$\text{PSOR: } u_{i,j}^{k+1} = u_{i,j}^{k} + \frac{\omega}{\alpha}(h^2 f_{i,j} + u_{i+1,j}^{k} + u_{i-1,j}^{k+1} + \beta^2 u_{i,j+1}^{k} + \beta^2 u_{i,j-1}^{k+1} - \alpha u_{i,j}^{k})$$

$$\text{LSOR: } -\omega u_{i-1,j}^{k+1} + \alpha u_{i,j}^{k+1} - \omega u_{i+1,j}^{k+1} = \alpha(1-\omega)u_{i,j}^{k} + \omega(h^2 f_{i,j} + \beta^2 u_{i,j+1}^{k} + \beta^2 u_{i,j-1}^{k+1})$$

where k is the iteration number and ω is the over-relaxation factor ($0 < \omega < 2$). These methods were parallelised using *red/black* strategy [18], with the domain divided evenly between pthreads as in the MD algorithm.

As for the initial trial values for the iterative methods, zero everywhere was assumed. Even when one assumes a "good" initial solution, PSOR and LSOR are still generally surpassed by direct methods, at least in terms of accuracy [1].

In the sample problems, the execution time of the MD algorithm was compared against the execution time of iterative methods using optimum values of ω. Some of the values of ω were found by numerical trial-and-error experiment, and are accurate up to the 5th decimal place.

The minimum storage requirements of these iterative methods are small: just *NM* floating point numbers or approximately ⅕ of MD's storage requirements.

6. Results

As a test of the MD algorithm, the following boundary value problem was solved on a rectangle *R*, shown in figure 3:

$$\begin{cases} \nabla^2 u = -f = -(2y^2 - 2yy_t + 2x^2 - 2xx_r) & 0 < x < x_r,\ 0 < y < y_t,\ x_r = 10,\ y_t = 10 \\ u = g = xy(x - x_r)(y - y_t) & \text{on the boundary } \partial R \end{cases}$$

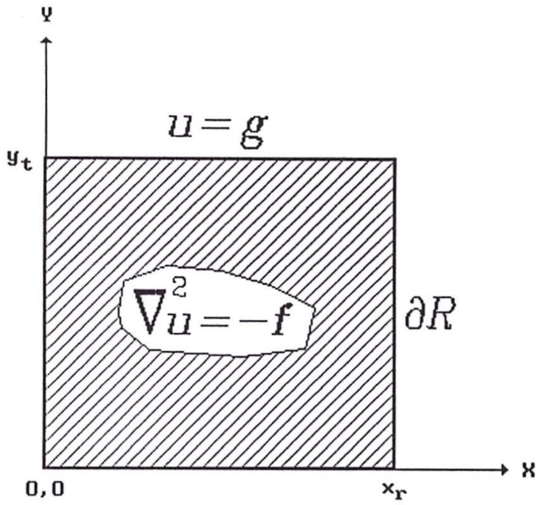


Figure 3
Computational domain for sample problems.

Three different values of *N* and *M* were used for the discretisation of the domain: 50 *x* 50, 100 *x* 100 and 300 *x* 300. The iterations of PSOR and LSOR were terminated after the residual *E* at iteration *k* satisfied

$$E = \frac{1}{NM} \sum_{i=1}^{N} \sum_{j=1}^{M} \left| u_{i,j}^{k} - \bar{u}_{i,j} \right| \leq 10^{-5} \qquad \textbf{6.I}$$

where $\overline{u}$ is the exact solution of the problem.

The figures below graph the *performance* of the parallel implementation of the three solution algorithms against the number of CPUs. Performance is defined here as

$$\text{Performance} = \frac{1}{\text{Execution Time}}$$

with the execution time being given in seconds.

Figure 4 below gives the performance of the MD algorithm for the 50 x 50 discretisation.

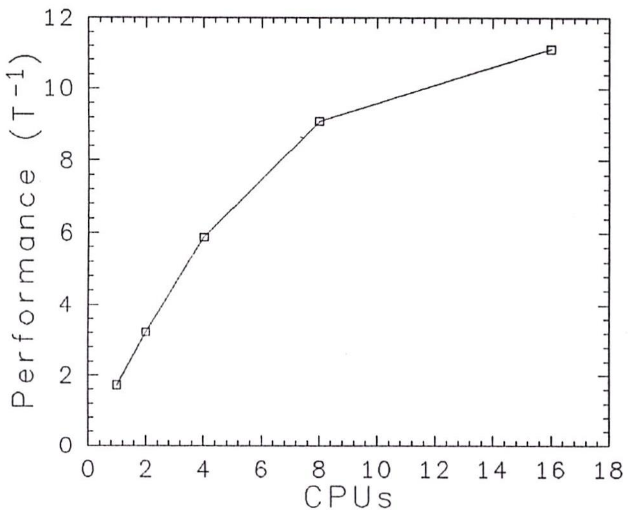


Figure 4
Performance of the MD algorithm, 50 *x* 50 discretisation.

There is a small decrease in performance after the 8-CPU mark is reached, due to synchronisation overheads and small data sizes. This decrease in performance is not observed in the other two sample problems with much greater number of data points.

The main point to notice is that when compared to the performance of PSOR and LSOR (figure 5), MD is almost *an order of magnitude faster*. The accuracy is approximately four orders of magnitude greater, as for MD, E (eq. 6.I) is smaller than 10^{-9}.

Although the LSOR iterative scheme converges in fewer iterations than the PSOR, its execution time is slightly greater, indicating a greater amount of work per iteration when compared to the PSOR scheme. Its performance is thus, poorer.

As shown in figure 5 below, both iterative methods also suffer from a decrease in performance after the 8-CPU mark is reached for a problem this size.

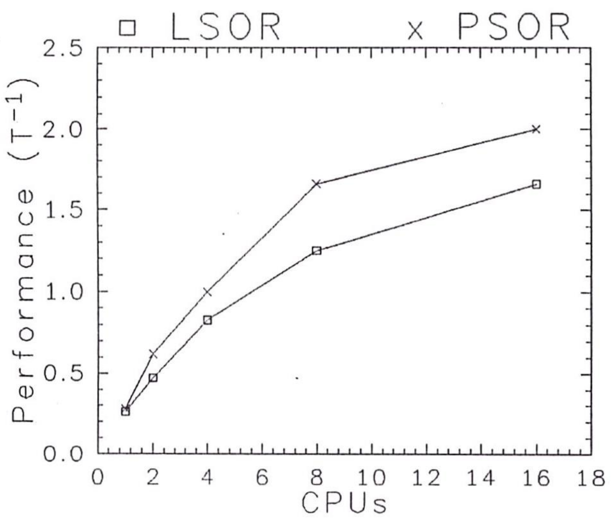


Figure 5
Performance of LSOR and PSOR, 50 *x* 50 discretisation.

Although all of the three methods perform much better in the 100 *x* 100 discretisation, MD's performance is the best, showing less loss in performance when the number of CPUs goes up, than the LSOR or PSOR.

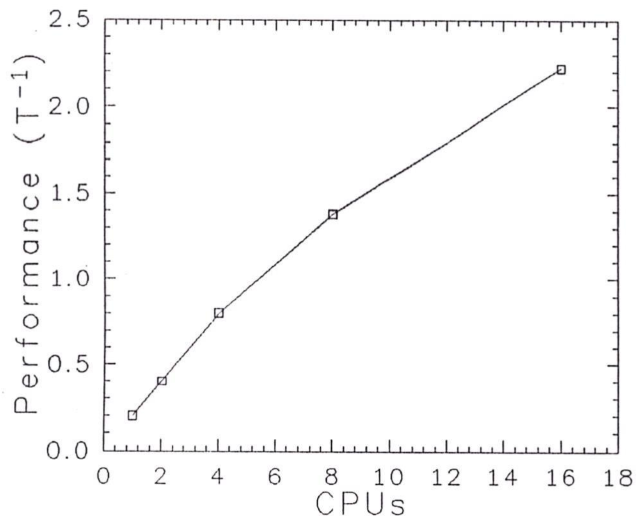


Figure 6
Performance of the MD algorithm, 100 *x* 100 discretisation.

The order-of-magnitude difference in execution time between MD and both PSOR and LSOR remains, even with the increased problem size.

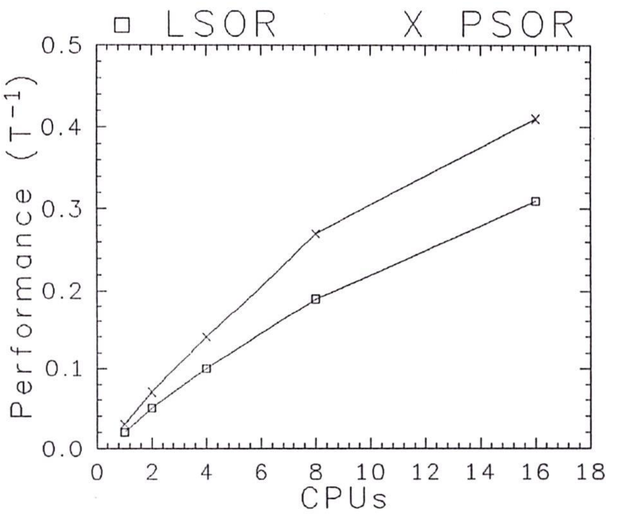


Figure 7
Performance of PSOR and LSOR, 100 *x* 100 discretisation.

For the 300 *x* 300 discretisation, the results follow the previously observed trend:

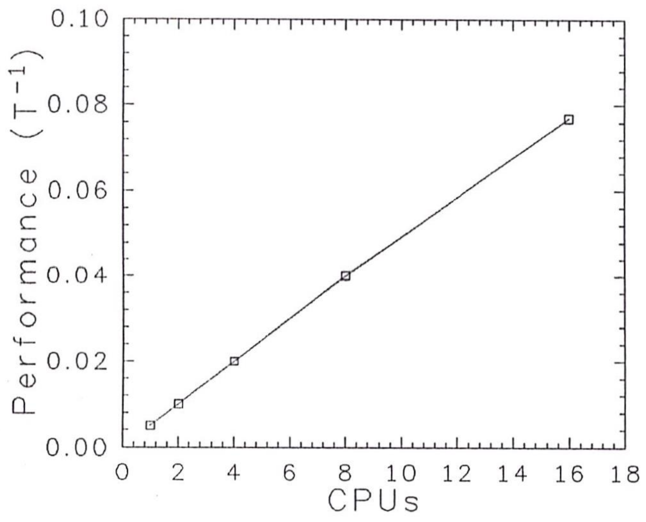


Figure 8
Performance of the MD algorithm, 300 *x* 300 discretisation.

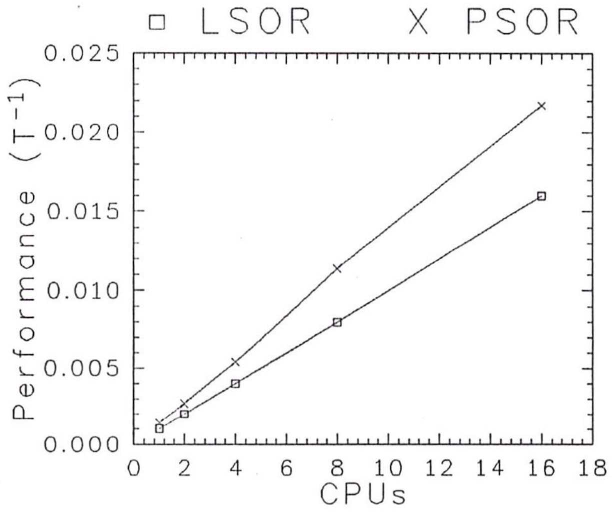


Figure 9
Performance of PSOR and LSOR, 300 *x* 300 discretisation.

7. Conclusions

For the sample problem and the three different discretisations, the Matrix Decomposition algorithm is consistently faster (by one order of magnitude) and more accurate (by four orders of magnitude) than the LSOR or PSOR iterative methods, even on one CPU. These numbers are in good agreement with the ones given in [13].

MD can be easily implemented and its potential for parallelisation is indeed great. Steps (2) and (4) are ideal for implementation on vector machines.

Even when using the best values of the over-relaxation factor, iterative methods are no match for direct methods. Moreover, it is sometimes difficult (if not impossible) to determine *a prori* the best value of ω which will gives huge savings in iteration count, as ω can be obtained from theory for some special cases only. Direct methods do not suffer from this problem, and the MD algorithm in particular is *no harder to program* than LSOR. Thus, if applicable, a direct method should be considered in place of the PSOR/LSOR method.

It is true that Conjugate Gradients-based methods are faster than LSOR or PSOR, and they should also be used. The objective of this work was to show that two different methods (LSOR/PSOR and MD), *roughly requiring the same programming effort*, give such different results for the model problem. If one wanted to compare these methods against Conjugate Gradients, then it would be only fair to include Multigrid Methods, which can be made to converge even faster than direct methods or Conjugate Gradients, if one uses the "Full Multigrid" (FMG) strategy.

Some possible extensions to this code are the investigation of the implementation of other general elliptic equations or discretisations for the Laplacian operator (such as the 9-point discretisation), and the implementation of domain partitioning, so that non-square geometries can be treated.

8. Bibliography

[1] Fletcher, C.A.J., *Computational Techniques for Fluid Dynamics.* Second edition, Springer-Verlag 1991.

[2] Miki, K., Takagi, T., *Numerical Solution of Poisson's Equation with Arbitrarily Shaped Boundaries Using a Domain Decomposition and Overlapping Technique.* Journal of Computational Physics 67, 263-278, 1986.

[3] Chan, T.F., Resasco, D.C., *A Domain-Decomposed Fast Poisson Solver on a Rectangle.* SIAM J. Sci. Stat. Comput. 8, 1, s14-s26, 1987.

[4] Concus, P., Golub, G.H., *Use of Fast Direct Methods for the Efficient Numerical Solution of Non-Separable Elliptic Equations.* SIAM J. Numer. Anal. 10, 6, 1103-1120, 1973.

[5] Swarztrauber, P.N., *The Methods of Cyclic Reduction, Fourier Analysis and the FACR Algorithm for the Discrete Solution of Poisson's Equation on a Rectangle.* SIAM Review 19, 3, 1977.

[6] Buzbee, B.L., Dorr, F.W., George, J.A., Golub, G.H., *The Direct Solution of the Discrete Poisson Equation on Irregular Domains.* SIAM J. Numer. Anal. 8, 4, 1971.

[7] Buzbee, B.L., Golub, G.H., Nielson, C.W., *On Direct Methods for Solving Poisson's Equation.* SIAM J. Numer. Anal. 7, 4, 1970.

[8] Pares-Sierra, A., Vallis, G.K., *A Fast Semi-Direct Method for the Numerical Solution of Non-Separable Elliptic Equations in Irregular Domains.* Journal of Computational Physics 82, 398-412, 1989.

[9] Swarztrauber, P.N., *A Direct Method for the Discrete Solution of Separable Elliptic Equations.* SIAM J. Num. Anal. 11, 6, 1974.

[10] King, H.H., *A Poisson Equation Solver for Rectangular or Annular Regions.* Int. Journal for Numerical Methods in Engineering 10, 799-807, 1976.

[11] Schumann, U., Sweet, R.A., *A Direct Method for the Solution of Poisson's Equation with Neumann Boundary Conditions on a Staggered Grid of Arbitrary Size.* Journal of Computational Physics 20, 171-182, 1976.

[12] Hockney, R.W., *A Fast Direct Solution of Poisson's Equation using Fourier Analysis.* J. ACM 12, 95-113, 1965.

[13] Buzbee, B.L., *A Fast Poisson Solver Amenable to Parallel Computation.* IEEE Transactions on Computers C-22, 8, 1973.

[14] Davis, R.W., Moore, E.F., Purtell, L.P., Physics of Fluids 27, 46-57, 1984.

[15] *KSR1 Parallel Programming Manual.* Kendall Square Research, 170 Tracer Lane, Waltham, MA 02154-1379, USA, 1991.

[16] Chapra, S.C., Canale, R.P., *Numerical Methods for Engineers.* McGraw-Hill International Editions, 1990.

[17] Hoffmann, K.A., *Computational Fluid Dynamics for Engineers.* Engineering Education System, 1989.

[18] Modi, J.J., *Parallel Algorithms and Matrix Computation.* Oxford Applied Mathematics and Computing Science Series, 1990.

[19] Freeman, T.L., Philips, C., *Parallel Numerical Algorithms.* Prentice-Hall International Series in Computer Science, 1992.

9. Code listing

```
implicit double precision(a-h,o-z)

parameter (n=100,m=100)
parameter (xl=0.d0,yt=10.d0,xr=10.d0,yb=0.d0)
parameter (maxnprocs=64)

common /md/ y(n,m), v(m,m), d(m), u(n,m)
common /bound/ flr(2,m), ftb(n,2)
common /const/ hx, hy, r2, pi
```

Figure 10
Listing of CONTROL.F

The listing of the code which implements the MD algorithm on the KSR1 follows below. It consists of two include files:

» **control.f**, where mesh parameters are defined;

» **func.f**, where functions *f* and *g* are defined.

plus the program file, **popar.f**.

```
c *********************************************
c the test function
        function fu(xpar,ypar)

        include 'control.f'

        fu=(xpar-xr)*xpar*(ypar-yt)*ypar

        end
c *********************************************
c the (negative) laplacian of the above function
        function flapu(xpar,ypar)

        include 'control.f'

        d2dx2=2.d0*ypar*ypar-2.d0*ypar*yt
        d2dy2=2.d0*xpar*xpar-2.d0*xpar*xr
        flapu=-(d2dx2+d2dy2)

        end
c *********************************************
```

Figure 11
Listing of FUNC.F

```
c sample implementation of the MD algorithm for the
c solution of the Poisson equation
        program sample
        include 'control.f'

c xl, yt              xr, yt
c    +    +---------+    +
c         |         |
c mk |----|---------|----|
c    |    |         |    |
c    |    |         |    |
c k  |----|---------|----|
c         |         |
c    +    +---------+    +
c xl, yb              xr, yb
c         h    ....  nh

c pthreads variables
        common /parint/ iteam, nprocs, ibar1, istat
        common /parstp/ imstart(maxnprocs+1), imend(maxnprocs+1),
       +instart(maxnprocs+1), inend(maxnprocs+1)

c barrier data structures & variables
        common /pthread_defaults/ipthread_attr_default,
       +ipthread_mutexattr_default,
       +ipthread_condattr_default,
       +ipthread_barrierattr_default

c *** say hello ***
        print *
        print *,'MD - Direct Poisson Solver'
        print *,'v1.0 - 04/06/93'
        print *
        print *,'Problem size = ',n,' x ',m
        print *

c initialize the timers
        call eqtimer(1, 'vty')
        call eqtimer(2, 'trid')
        call eqtimer(3, 'vu')
        call eqtimer(4, 'MD')

c set the number os CPUs
        print *,'Number of processors?'
        read *, nprocs
        call ipr_create_team(nprocs, iteam)
        print *,'Iteam=',iteam
        print *
```

Figure 12

Listing of POPAR.F (I)

```
c create the barrier
        call pthread_barrier_init(ibar1,
       +ipthread_barrierattr_default,nprocs,istat)

        if (istat.ne.0) then
            print *,'Error creating the barrier! - Error code ', istat
            stop
        endif

c and balance the system
        print *
        print *,'Data partitioning - m space'
        call getload(m, imstart, imend, nprocs)
        print *
        print *,'Data partitioning - n space'
        call getload(n, instart, inend, nprocs)

c initialize values of md data structures
        call setup

c initialize the boundary values
        print *
        print *,'Initializing boundary values...'
        call initb

c initalize the RHS of poisson's equation
        print *
        print *,'Building RHS...'
        call initrhs

c finally, execute md
        print *
        print *,'Calling MD...'
        call mdr

c compute and print the error
        dif=0.d0
        do 140 j=1,m
            do 140 i=1,n
 140            dif=dif+dabs(u(i,j)-fu(hx*dble(i),hy*dble(j)))

        dif=dif/(dble(n)*dble(m))
        print *,'Difference = ',dif

c print the results
        call printet()
        end
```

Figure 14

Listing of POPAR.F (II)

```
c ***************************************************
c this routine does the actual solving...

        subroutine mdr
        include 'control.f'

        common /parint/ iteam, nprocs, ibar1, istat
        common /parstp/ imstart(maxnprocs+1), imend(maxnprocs+1),
       +instart(maxnprocs+1), inend(maxnprocs+1)

c multiply all vectors y by v transpose
        print *
        print *,'MD: Performing Vt.Y...'

        call eton(1)
        call eton(4)

c*KSR* parallel region(teamid=iteam,
c*KSR*&private=(me,me_start,me_end,i))

c see who is doing this instance (0 <= me <= maxnprocs-1)
        me=ipr_mid()

c get the indexes for this pthread (n dimension)
        me_start=instart(me+1)
        me_end=inend(me+1)

        do 10 i=me_start,me_end
 10         call mvtm(i)

c and check it into the barrier
        call pthread_barrier_checkin(ibar1, me, istat)
        call pthread_barrier_checkout(ibar1, me, istat)

c inform the user what we are doing
        if (me.eq.0) then
            call etoff(1)
            call eton(2)
            print *
            print *,'MD: Solving Tridiagonal systems...'
        endif

        call pthread_barrier_checkin(ibar1, me, istat)
        call pthread_barrier_checkout(ibar1, me, istat)

c get the indexes for this pthread (m dimension)
        me_start=imstart(me+1)
        me_end=imend(me+1)
```

Figure 14

Listing of POPAR.F (III)

```
        do 100 i=me_start, me_end

c solve this tridiagonal system
            call trdg(i)

c repeat for every tridiagonal system
 100    continue

        call pthread_barrier_checkin(ibar1, me, istat)
        call pthread_barrier_checkout(ibar1, me, istat)

c inform the user again...
        if (me.eq.0) then
            call etoff(2)
            call eton(3)
            print *
            print *,'MD: Performing V.U...'
        endif

        call pthread_barrier_checkin(ibar1, me, istat)
        call pthread_barrier_checkout(ibar1, me, istat)

c get the indexes for this pthread (n dimension)
        me_start=instart(me+1)
        me_end=inend(me+1)

c now multiply each vector u by v
        do 130 i=me_start,me_end
 130        call mvu(i)

c*KSR* end parallel region
        call etoff(3)
        call etoff(4)

        end
```

Figure 14

Listing of POPAR.F (IV)

```
c *******************************************************
c divides and balances the data space among cpus
        subroutine getload(ilenght, istart, iend, ncpus)
        implicit double precision (a-h,o-z)

        dimension istart(*), iend(*)
        integer ncpus, ilenght

c zero the process counter per cpu
        do 5 i=1, ncpus+1
 5          iend(i)=0

c distribute the processes among all the cpus
        j=1
        do 10 i=1, ilenght
            iend(j)=iend(j)+1
            j=j+1
            if (j.gt.ncpus) then
                j=1
            endif
 10     continue

c now find the end of each one
        istart(1)=1
        if (ncpus.gt.1) then
            do 15 j=2,ncpus+1
 15             istart(j)=iend(j-1)+istart(j-1)

            do 20 j=1,ncpus
 20             iend(j)=istart(j+1)-1

c if one 1 cpu available
        else
            iend(1)=ilenght
        endif

        print *, ' cpu      starts    ends'

        do 439 k=1, ncpus
            idif=iend(k)-istart(k)+1
 439        write (*,500) k-1, istart(k),iend(k),idif

 500    format(i3,'         ',i3,'         ',i3,'    n#=',i3)

        end
```

Figure 14

Listing of POPAR.F (V)

```
c ***********************************************
c solves a tridiagonal matrix in compressed form

         subroutine trdg(indx)

         include 'control.f'

         double precision tri(n,4)

c build the RHS vector and the (compressed) coefficient matrix
         do 110 j=1,n
              tri(j,1)=-1.d0
              tri(j,3)=-1.d0
              tri(j,4)=y(j,indx)
 110          tri(j,2)=d(indx)

c zero the two non-existent elements
         tri(1,1)=0.d0
         tri(n,3)=0.d0

c decomposition
         do 10 k=2, n
              tri(k,1)=tri(k,1)/tri(k-1,2)
              tri(k,2)=tri(k,2)-tri(k,1)*tri(k-1,3)
 10      continue

c forward substitution
         do 20 k=2,n
 20           tri(k,4)=tri(k,4)-tri(k,1)*tri(k-1,4)

c back substitution
         tri(n,4)=tri(n,4)/tri(n,2)
         do 30 k=n-1,1,-1
 30           tri(k,4)=(tri(k,4)-tri(k,3)*tri(k+1,4))/tri(k,2)

c copy the result back into u
         do 120 j=1,n
 120          u(j,indx)=tri(j,4)

         end
```

Figure 14

Listing of POPAR.F (VI)

```
c ***********************************************
c multiplies v by u, to obtain the final result
        subroutine mvu(indx)

        include 'control.f'

        dimension x(m)

        do 5 i=1,m
 5          x(i)=0.d0

        DO 1 j = 1,m
            DO 1 i = 1,m
                X(i) = X(i) + v(i,j)*u(indx,j)
 1      continue

        do 10 i=1, m
 10         u(indx,i)=x(i)

        RETURN
        END

c ***********************************************
c multiples v (transposed) by a vector y
        subroutine mvtm(indx)

        include 'control.f'

        dimension ytilde(m)

        DO 1 i = 1,m
            x=0.d0
            DO 2 j = 1,m
 2              X = X + v(j,i)*y(indx,j)

            ytilde(i)=x
 1      continue

c copy over vector
        do 7 i=1,m
 7          y(indx,i)=ytilde(i)

        RETURN
        END
```

Figure 14

Listing of POPAR.F (VII)

```
c ***************************************************
c sets up some values used by the md routine,
c including the matrix v and computes the
c eigenvalues
        subroutine setup

        include 'control.f'

c setup constants
        pi=4.d0*atan(1.d0)
        s=dble(sqrt(2.d0/dble(m+1)))
        hx=(xr-xl)/dble(n+1)
        hy=(yt-yb)/dble(m+1)
        rho=hx/hy
        r2=rho*rho
        sigma=2.d0*(1.d0+r2)
        sp=pi/dble(m+1)

c build matrix v
        do 10 i=1, m
            do 10 j=1, m
                v(i,j)=s*sin(dble(i)*dble(j)*sp)
 10     continue

c build vector d (eigenvalues)
        do 20 i=1,m
            d(i)=sigma-2.d0*r2*cos(dble(i)*sp)
 20     continue

        end
 c ***************************************************
c initialize boundary values
        subroutine initb

        include 'control.f'

        do 30 i=1,m
            flr(1,i)=fu(xl,dble(i)*hy)
            flr(2,i)=fu(xr,dble(i)*hy)
 30     continue

c setup the boundary values for the square
        do 35 j=1,n
            ftb(j,1)=fu(dble(j)*hx,yb)
            ftb(j,2)=fu(dble(j)*hx,yt)
 35     continue

        end
```

Figure 14

Listing of POPAR.F (VIII)

```
c ***************************************************
c initialize RHS
        subroutine initrhs

        include 'control.f'

c compute vectors y
        do 40 i=1,n
            do 40 j=1,m

c leftmost column
                    if (i.eq.1) then
                        if (j.eq.1) then
                            y(1,1)=hx*hx*flapu(hx,hy)
                            y(1,1)=y(1,1)+r2*ftb(1,1)+flr(1,1)
                        endif
                        if (j.ge.2.and.j.le.m-1) then
                            y(1,j)=hx*hx*flapu(hx,dble(j)*hy)
                            y(1,j)=y(1,j)+flr(1,j)
                        endif
                        if (j.eq.m) then
                            y(1,m)=hx*hx*flapu(hx,dble(m)*hy)
                            y(1,m)=y(1,m)+r2*ftb(1,2)+flr(1,m)
                        endif
                    endif

c middle columns
                    if (i.ge.2.and.i.le.n-1) then
                        if (j.eq.1) then
                            y(i,1)=hx*hx*flapu(dble(i)*hx,hy)
                            y(i,1)=y(i,1)+r2*ftb(i,1)
                        endif
                        if (j.ge.2.and.j.le.m-1) then
                            y(i,j)=hx*hx*flapu(dble(i)*hx,dble(j)*hy)
                        endif
                        if (j.eq.m) then
                            y(i,m)=hx*hx*flapu(dble(i)*hx,dble(m)*hy)
                            y(i,m)=y(i,m)+r2*ftb(i,2)
                        endif
                    endif
```

Figure 14

Listing of POPAR.F (IX)

```
c rightmost column
           if (i.eq.n) then
               if (j.eq.1) then
                   y(n,1)=hx*hx*flapu(dble(n)*hx,hy)
                   y(n,1)=y(n,1)+r2*ftb(n,1)+flr(2,j)
               endif
               if (j.ge.2.and.j.le.m-1) then
                   y(n,j)=hx*hx*flapu(dble(n)*hx,dble(j)*hy)
                   y(n,j)=y(n,j)+flr(2,j)
               endif
               if (j.eq.m) then
                   y(n,m)=hx*hx*flapu(dble(n)*hx,dble(m)*hy)
                   y(n,m)=y(n,m)+r2*ftb(n,2)+flr(2,m)
               endif
           endif

40      continue

        end

c ***************************************************

        include 'func.f'
```

Figure 14

Listing of POPAR.F (X)