The Parallel Solution of Linear Elliptic Equations using Hopscotch Algorithms

Osama El-Giar and Tim Hopkins
Computing Laboratory
University of Kent
Canterbury
Kent, CT2 7NF
U.K.

Abstract

We consider a distributed solution of linear elliptic partial differential equations in two and three dimensions using hopscotch algorithms on rings of transputers. An occam implementation of these methods is reported. The gain in efficiency obtained by increasing the number of processors is shown to be nearly linear.

1 Introduction

We consider the numerical solution of elliptic equations of the form

$$Lu = -g(\mathbf{x}) \tag{1}$$

On a given region R where $\mathbf{x} = (x_1, x_2, \ldots, x_d)^T$ and L is a general linear elliptic differential operator in d-space dimensions subject to boundary conditions of the general form

$$a(\mathbf{x})u - b(\mathbf{x})u_{x_i} = c(\mathbf{x})$$

on ΩR

The Hopscotch method was first introduced by Gordon [2] and was generalized in a series of papers by Gourlay and his co-workers (see Gourlay [4] and references therein for a good review of this work). The method consists approximately the solution to 1 at the intersection points of an orthogonal grid system on R. We approximate L by a finite difference approximation, L_h, on $R_g \cup \Omega R_g$ where R_g consists of the internal grid points and ΩR_g the points of intersection of the grid lines with the boundary, ΩR_g.

The first step in the Hopscotch procedure is to split R_g into two distinct subsets P_1 and P_2 such that $R_g = P_1 \cup P_2$. The choice of P_1 and P_2 is designed to limit the degree of implicitness in the resultant method and hence limit the amount of computation required.

We report on the implementation of several different Hopscotch algorithms in occam on a ring of transputers. Results are presented for problems in 2 and 3 dimensions for both Poisson's equation and the more general case where L contains mixed derivatives. The efficiency of distributing the computational load over an increasing number of processors is discussed.

2 Two-dimensional hopscotch schemes

For the sake of clarity we will initially restrict our discussion to the solution of Poisson's equation subject to Dirichlet boundary conditions using a grid $R_g = \{(ih, jh) : i, j = 1, 2, \ldots, n\}$ on the unit square R. Both the method and implementation will generalize to more general shaped regions, non-square grids and more complex equations.

Using the usual five-point difference approximations we obtain, L_h, the finite difference replacement of the operator L where

$$L_h u_{ij} = (-4u_{ij} + u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1})/h^2$$

where $u_{ij} \approx u(ih, jh)$. Applying this approximation at each internal grid point would lead to a system of block tridiagonal equations of order n^2. As an example of the Hopscotch precess we consider the odd-even method. Following Gourlay [3] we use a two level iterative scheme of the form

$$u_{ij}^{m+1} = u_{ij}^m + \omega(L_h u_{ij}^m + g_{i,j}^m)$$

at the points $P_1 = \{(ih, jh) : i + j \text{ even}\}$ followed by

$$u_{ij}^{m+1} = u_{ij}^m + \omega(L_h u_{ij}^{m+1} + g_{i,j}^{m+1})$$

at the points $P_2 = \{(ih, jh) : i + j \text{ odd}\}$.

Note that each step there is only one unknown, u_{ij}^{m+1}, in each of these equations, since in the second equation $u_{i\pm1,j}^{m+1}$ and $u_{i,j\pm1}^{m+1}$ were determined using the first level iteration. We may replace this pair of equations with a single formula which defines the algorithm locally at all the internal grid points.

$$u_{ij}^{m+1} - \omega\theta_{ij}^{m+1}[L_h u_{ij}^{m+1} + g_{ij}^{m+1}] = u_{ij} + \omega\theta_{ij}^m[L_h u_{ij}^m + g_{ij}^m]$$

where

$$\theta_{ij}^m = \begin{cases} 1 & \text{if } m + i + j \text{ odd} \\ 0 & \text{if } m + i + j \text{ even} \end{cases}$$

If we consider two successive steps we find that for $(i+j+m)$ even

$$u_{ij}^{m+2} = 2u_{ij}^{m+1} - u_{ij}^{m}.$$

This is the *fast step* of the hopscotch algorithm.

We now have the following algorithm for the odd-even hopscotch method

initialize Provide an initial estimate to the solution of 1 at all the internal grid points. Set $m = 0$

separate For all interior points with $(m+i+j)$ odd overwrite u_{ij}^m with

$$u_{ij}^{m+1} = u_{ij}^{m} + \omega[L_h u_{ij}^m + g_{ij}^m] \tag{2}$$

fast-step For all interior points with $(m+i+j)$ even define u_{ij}^{m+1} by

$$u_{ij}^{m+1} - \omega L_h u_{ij}^{m+1} = \omega g_{ij}^{m+1} + u_{ij}^{m} \tag{3}$$

then overwrite u_{ij}^m by

$$u_{ij}^{m+2} = 2u_{ij}^{m+1} - u_{ij}^{m}. \tag{4}$$

Set $m = m+1$

in-phase If the approximation is required over the whole grid at a particular step, m, e.g., to terminate the iteration or for an intermediate viewing of the approximation, then for all $m+i+j$ even overwrite u_{ij}^m with

$$u_{ij}^{m+1} - \omega L_h u_{ij}^{m+1} = u_{ij}^{m} + \omega g_{ij}^{m+1} \tag{5}$$

and either continue with the iteration by returning to the **separate** step or terminate the computation.

Otherwise return to **fast-step**

We note here that at each fast step all the points involved may be computed simultaneously.

The above computational procedure may be generalized in terms of the two sets of points P_1 and P_2 where P_2 is the complement of P_1 to give

initialize Set up the initial vector. Set $m = 0$ and $k = 1$,

separate For all points $\in P_k$ overwrite u_{ij}^m using (2),

fast-step For all points $\ni P_k$ define u_{ij}^{m+1} using (3) and overwrite u_{ij}^m using (4). Set $k = 3-k$ and $m = m+1$

in-phase To bring all points back in phase; for all points $\ni P_k$ overwrite u_{ij}^m using (5) and to continue return to the **separate** step, otherwise return to **fast-step**

We may now define a number of different hopscotch methods based on our choice of P_1, for example,

odd-even (Gourlay [3])

$$P_1 = \{(ih, jh) : i + j \text{ odd}\}$$

block (Evans & Danaee [1])

$$P_1 = \left\{(ih, jh) : \left\lfloor \frac{i+1}{2} \right\rfloor + \left\lfloor \frac{j+1}{2} \right\rfloor \text{ odd}\right\}$$

line (Gourlay & McGuire [5])

$$P_1 = \{(ih, jh) : i \text{ odd}\}.$$

This is known as the x-line method. A second line ordering, known as the y-line ordering, is possible using $P_1 = \{(ih, jh) : j \text{ odd}\}$.

Figure 1 shows how the points are distributed within the grid for each of these methods.

Note that the equations (3) and (5) require the solution of sets of linear systems of equations (for a linear operator L) of order 4 for the block grouping and order n (tridiagonal) for the line grouping. The amount of computation increases with the amount of implicitness although there is generally a compensatory gain in accuracy and/or speed of convergence (see Evans & Danaee [1] for details of some operations counts and accuracy comparisons).

When considering mixed derivative terms we would like to be able to find approximations to the mixed derivative which do not increase the implicitness of the resultant linear

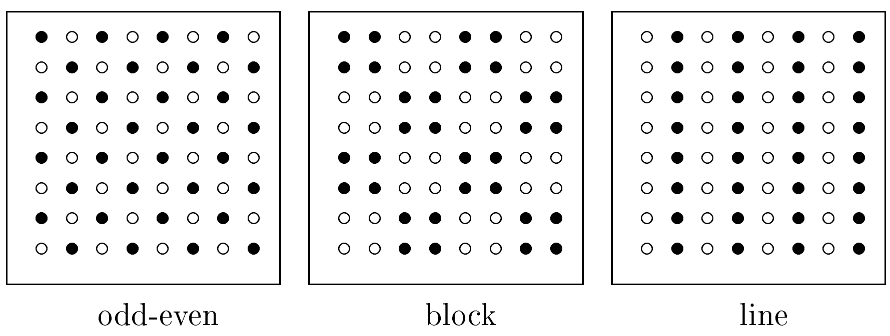

Figure 1: Point distribution for two-dimensional hopscotch methods

systems. Gourlay & McKee [6] give a method for extending the odd-even method but it depends upon computing the point approximations in a specified order. Thus, unlike the case without the mixed term, we cannot compute the approximation at all the active points simultaneously.

For both block and line methods it is possible to approximate the mixed derivative term without increasing the order of the intermediate systems of equations but at the expense of slightly more complex systems (see Gourlay & McKee [6] and Evans & Danaee [1] for details).

3 Three-dimensional hopscotch methods

Once again we begin by restricting our discussions to the simpler case of Poisson's equation on a unit cube subject to Dirichlet boundary conditions. We impose a regular cubic grid, $R_d = \{(ih, jh, kh) : i, j, k = 1, 2, \ldots, n\}$ and define our finite difference replacement operator, L_h, in terms of the approximation point and its six nearest neighbour points. Proceeding as before we identify four orderings

odd-even $P_1 = \{(ih, jh, kh) : i + j + k \text{ odd}\}$,

block $P_1 = \{(ih, jh, kh) : \lfloor \frac{i+1}{2} \rfloor + \lfloor \frac{j+1}{2} \rfloor + \lfloor \frac{k+1}{2} \rfloor \text{ odd}\}$,

line $P_1 = \{(ih, jh, kh) : i + j \text{ odd}\}$ two other line orderings are possible by taking $i + k$ odd or $j + k$ odd,

plane $P_1 = \{(ih, jh, kh) : i \text{ odd}\}$ two other plane orderings are possible by taking j odd and k odd.

as before P_2 is the complement of P_1. The computational procedure is as for the two-dimensional case although the degree of implicitness has changed for the block case which now requires the solution of sets of linear equations of order 8. For the plane hopscotch process the implicit step involves the solution of block tridiagonal systems of order n^2; this is equivalent to a direct solution to the two-dimensional problem.

If we are not to increase the implicitness of the approximation when mixed derivatives are present then the following restrictions apply

odd-even all mixed derivative terms may be approximated provided that the points are updated in a specific order,

block no restriction – all three terms may be approximated simultaneously,

line two mixed derivative terms may be approximated depending on which line method is chosen, for example, if the $(i+j)$ odd ordering is used the mixed terms u_{yz} and u_{xz} may be approximated. The third term may also be included if an ordering is imposed on the calculations,

plane two mixed derivative terms may be approximated depending on which plane method is chosen, for example, if the i odd ordering is used the mixed terms u_{xy} and u_{xz} may be approximated. The third term may also be included if we are willing to increase the complexity of the resultant linear systems from a maximum of 5 unknowns/equation to 9 unknowns/equation. The resultant system in this case is still block tridiagonal with each block now tridiagonal.

Since values need to be computed in a specific order, the odd-even, line and hopscotch schemes are not suitable for parallel implementation on transputers when all mixed derivative terms are present.

4 Transputer implementation

All the parallel algorithms discussed in this paper have been implemented in occam 2 on a Meiko Computing Surface using networks of up to 8 T800 processors.

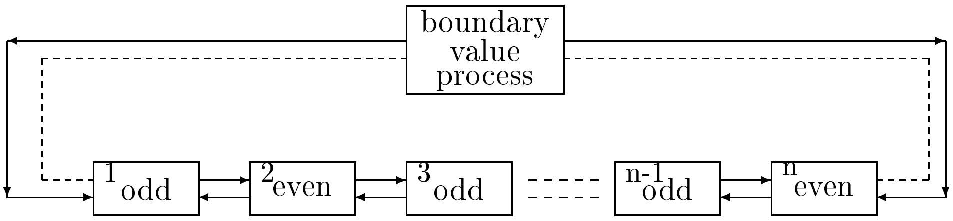

Figure 2: Process network

We start by considering the implementation of the line hopscotch method for approximating Poisson's equation in two dimensions using a ring of $n+1$ processes (where n is assumed even) – see Figure 2. The hopscotch approximations along each line of unknowns are computed using n of these processes; the one remaining is used to provide the boundary values along the edges parallel to the lines of unknowns. The parallel algorithm is then

1. Provide each process with an appropriate initial approximation,

2. All even processes (and possibly the boundary value process) simultaneously send their initial guesses to their neighbouring processes. On receipt all odd processes simultaneously execute the **separate** step and on completion send these newly computed values to their neighbouring even processes.

3. Alternating between odd and even processes, the main iteration cycle uses the latest transmitted values to compute the **fast-step** values and these in turn are passed to their nearest neighbour processes.

4. When termination is requested (we use a fixed number of iterations but other schemes are possible) all the lagging processes simultaneously perform the **in-phase** step. The final values are then passed anti-clockwise through the processes to the output device.

At each implicit step each active process needs to solve a system of tridiagonal linear equations; it thus updates n values from the point set P_1 or P_2. The amount of data transmitted and received at each step consists of n floating-point numbers in each direction.

To obtain a balanced use of the transputers implementing this algorithm an even number of processes should be placed on each processor since half the processes are idle when the **fast-step** computation takes place. The boundary value process is included with the first group.

For the three dimensional case we may implement the plane method in exactly the same way; each process now has to solve a block tridiagonal system at each implicit step and the amount of data transmitted to neighbouring processes rises to n^2 floating-point numbers. It may be possible to attain a higher degree of parallelism by using more than one transputer to solve these systems of equations. Currently we restrict the value of n to be of the form $2^t - 1$ and use matrix decomposition with fast Fourier transform on a single processor.

We implement the two-dimensional point method using the ring topology of Figure 2 by arranging for the points to be grouped in zigzag lines with the rth line containing the points $\{(rh, jh) : j = 1, 3, 5, \ldots \cup ((r-1)h, jh) : j = 2, 4, 6, \ldots\}$ see Figure 3. Note all points within a zigzag belong to the same (odd or even) group.

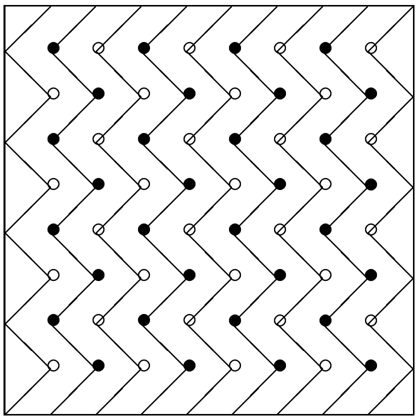

Figure 3: Zigzag grouping

To balance the implementation we use the boundary value process to compute the points in the zigzags adjacent to the boundaries. The remaining points are split into groups containing an even number of zigzags which are assigned to a processor. Note that, now the boundary value process has extra work to do, it is included with the first and last of these groups. Although the amount of data transmitted is the same as for the line algorithm; the amount of computation is reduced to explicit formulae evaluations.

The three-dimensional line implementation for Poisson's equation is accomplished in a similar fashion; each process deals with a set of lines consisting of points belonging to the same group. Thus if Figure 3 is interpreted as an x-y plane, then the $(i+j)$ odd method groups the zigzag of *lines* as shown. Each active process is now required to solve n systems of tridiagonal equations of order n as opposed to the single block tridiagonal system of order n^2 required by the plane method.

The zigzag grouping may also be used to distribute the two-dimensional block scheme; the zigzags now connect blocks of the same coloured points. We note here that the level of concurrency is restricted since for an $n \times n$ grid this grouping will only allow a maximum of $\frac{1}{2}n$ processes to be used. The use of larger blocks as suggested by Evans & Danaee [1] would further restrict the distribution.

Finally we consider the point and block hopscotch schemes for the three-dimensional case. For the point case we assign all odd points in the first and last planes to the boundary value process. All even points in planes 1 and 2 are assigned to process 1, all odd points in planes 2 and 3 to process 2 etc. With the block implementation we consider planes of blocks (two of our point planes) and assign blocks in the same way. As with the two-dimensional case the use of blocks restricts the number of processors we can use to $\frac{1}{2}n$.

Any mixed derivative terms present may be accommodated into the above implementation without affecting the distribution of the computation provided that the block method is used for the two-dimensional case or the block or plane method for the three-dimensional problems (see the discussion in §2 and §3. Rectangular regions and grids and more general boundary conditions can also be included in a straightforward manner.

5 Results

We have solved a number of equations of the form (1), both with and without mixed derivative terms, using the methods described in the previous sections. In this section we present a few examples of the typical gains in performance obtained when the computation was distributed over an increasing number of transputers. We chose to measure the efficiency in distributing the algorithm over p transputers by

$$E_p = \frac{T_1}{pT_p}$$

n	E_2	E_4	E_8
8	0.85	0.63	0.43
16	0.99	0.92	0.48
32	0.99	0.97	0.84
64	0.99	0.96	0.93
128	0.98	0.95	0.90

Figure 4: Example 1 using odd-even hopscotch

where T_r is the execution time on a network of r transputers. The closer E_p is to unity the closer we are to a linear speed-up. Figures are also presented which compare the parallel implementation in occam with a sequential implementation in Fortran run on a VAX 8800.

All the results were obtained using a value of the iteration parameter ω of $1.7 \times h^2$ and applying 1000 iterations. These values are not optimal for all the problems solved and have been used only to compare the effectiveness of distributing the computational method. The domain of all the examples is the unit square or unit cube and a regular grid has been used throughout.

Example 1 – the two-dimensional Poisson's equation

$$u_{xx} + u_{yy} = -2\pi^2 \sin \pi x \sin \pi y$$

subject to Dirichlet boundary conditions

$$u(x, 0) = u(x, 1) = u(0, y) = u(1, y) = 0$$

which has an exact solution of $u(x, y) = \sin \pi x \sin \pi y$.

This problem has been solved using the odd-even method for various values of n and the results are given in Figure 4

The poor value of E_8 when $n = 8$ is because half the processors are idle at each **fast-step**. The values for E_4 for $n = 8$ and E_8 for $n = 16$ are due to the communication time dominating the computation time at each step. All other cases show an almost linear speed-up.

Example 2 – the two-dimensional equation with a mixed derivative term

$$(5x + 1)u_{xx} + 2yu_{xy} + 3yu_{yy} = y(7xy + 3x^2 + 5y + 2)u$$

subject to Dirichlet boundary conditions

$$u(x, 0) = u(0, y) = 1, u(1, y) = e^y, u(x, 1) = e^x$$

which has an exact solution of $u(x, y) = e^{xy}$.

	n	E_2	E_4	E_8	$\log_{10}$ (ARE)
line	8	1.00	0.55	0.51	−4.1
	16	1.00	0.97	0.53	−4.7
	32	1.00	0.98	0.95	−5.3
	64	1.00	0.98	0.98	−5.9
	128	0.99	0.97	0.96	−3.0
block	8	0.54	0.47	—	−4.0
	16	0.95	0.78	0.51	−4.5
	32	1.12	1.12	0.96	−5.1
	64	1.00	1.20	1.19	−5.7
	128	1.04	1.11	1.25	−2.9

Figure 5: Example 2 using line and block hopscotch

Figures 5 shows the results obtained using the line and block algorithms. The final column gives the log of the average relative error (ARE) over all the points in the grid and shows that typically the line method is slightly more accurate than the block method. Figure 6 gives the time, in seconds, for the block hopscotch method using 2 transputers (T_2) and running a sequential Fortran program on a VAX 8800 under VMS 4.7 (T_{vms}), and illustrates the floating-point processor power of the transputer.

The superlinear speed-up exhibited by the block hopscotch method is thought to be due to the fact distributing the algorithm over an increasing number of transputers leads to more local (on-chip) memory being available for the storage of temporary variables used by the compiler (e.g., array indices).

Example 3 – the three-dimensional Poisson's equation

$$u_{xx} + u_{yy} + u_{zz} = -3\pi^2 \sin \pi x \sin \pi y \sin \pi z$$

subject to zero Dirichlet boundary conditions over the unit cube. This has an exact solution $u(x, y, z) = \sin \pi x \sin \pi y \sin \pi z$

Figure 7 shows the results for the odd-even, line and plane hopscotch methods. Only three values of n are available for the odd-even and line methods and two for the plane algorithm due to memory restrictions on the transputers used.

n	8	16	32	64	128
T_{vms}	3.58	15.2	60.5	245	1032
T_2	4.11	10.2	39.2	177	758

Figure 6: Example 2 comparison of times for block hopscotch

	n	E_2	E_4	E_8
odd-even	8	0.95	0.79	0.40
	16	0.99	0.99	0.70
	32	1.00	0.99	0.95
line	8	1.02	0.85	0.50
	16	1.00	1.02	0.88
	32	1.00	1.00	1.00
plane	7	0.90	0.80	0.48
	15	0.95	0.97	0.97

Figure 7: Example 3 using odd-even, line and plane hopscotch

6 Conclusion

We have shown that hopscotch methods in general provide an efficiently distributable algorithm for the solution of Poisson's equation in two and three dimensions using transputer networks. Several of these algorithms may be extended, without loss of efficiency, to deal with the case of mixed derivative terms. More complex boundary conditions may also be incorporated.

The same approach may be applied to parabolic equations of the form

$$u_t = Lu + g$$

where L and g are as defined for (1). For these equations hopscotch methods combine the stability conditions of totally implicit finite difference methods with the computational convenience of explicit methods.

Hopscotch methods also provide an efficient means of solving non-linear differential equations. Compared to totally implicit methods (e.g., Crank-Nicolson) hopscotch reduces the order of the resultant systems of non-linear equations whilst allowing them to be easily distributed over a number of transputers.

7 Acknowledgements

We would like to acknowledge the help, guidance and useful discussions we have had during this research with Peter Welch, Colin Willcock and Tony Curtis at UKC. Thanks also to Peter for improving a first draft of this paper when he had much more important things to do!

References

[1] D. J. Evans and A. Danaee. A new group hopscotch method for the numerical solution of partial differential equations. *Siam J. Numer. Anal.*, 16:588–598, 1982.

[2] P. Gordon. Nonsymmetric difference schemes. *SIAM J. Appl. Math.*, 13:667–673, 1965.

[3] A. R. Gourlay. Hopscotch: a fast second-order partial differential equation solver. *J. Inst. Maths Applics*, 6:375–390, 1970.

[4] A. R. Gourlay. Splitting methods for time-dependent partial differential equations. In D. A. H. Jacobs, editor, *The State of the Art in Numerical Analysis*, pages 757–796. Academic Press, 1977.

[5] A. R. Gourlay and G. R. McGuire. General hopscotch algorithm for the numerical solution of partial differential equations. *J. Inst. Maths Applics*, 7:216–227, 1971.

[6] A. R. Gourlay and S. McKee. The construction of hopscotch methods for parabolic and elliptic equations in two space dimensions with a mixed derivative. *J. Comput. Appl. Math.*, 3:201–206, 1977.