A Generally Configurable Multigrid Implementation for the Solution of Three-Dimensional Elliptic Equations on a Transputer Network

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

Abstract

We report on the implementation of a generally configurable version of the full multigrid method on arrays of transputers. The design of a parallel algorithm to allow for the easy distribution of the computation over the available processors is discussed. We present results for the case of Poisson's equation on a unit cube with Dirichlet boundary conditions and show that the efficiency obtained by increasing the number of processors is close to linear.

KEYWORDS: Poissons equation, occam, multigrid methods, transputer networks, elliptic partial differential equations

1 Introduction

The three-dimensions version of the parallel multigrid algorithm presented here is an extension of the two-dimensional implementation reported in El-Giar and Hopkins [3]. As with the two-dimensional method we require neither the H-Gray codes used by Chan and Saad [2] or the shuffle operations of McBryan and Van de Velde [5] to obtain high efficiency. Rather we make the implementation as simple and straightforward as possible and allow the high speed communication links on the transputer to provide us with an efficient implementation. A major advantage of our implementation is that it may be very easily distributed over differing numbers of processors without changes to the code.

We begin with a brief description of the sequential $FMG(2,1)$ algorithm applied to our model problem; Poisson's equation on the unit cube. Thus we consider

$$u_{xx} + u_{yy} + u_{zz} = -f(x,y,z) \quad \text{in } \Omega = [0,1] \times [0,1] \times [0,1] \tag{1}$$

$$u = g(x,y,z) \quad \text{on } \partial\Omega$$

The unit cube is first discretized using a uniform grid of size $h = 1/(N+1)$ in all three directions. The resultant points, (x_i, y_j, z_k), are coloured red if $(i+j+k)$ is odd and black otherwise. By applying the standard seven-point finite difference approximation to the Laplacian at each point in the grid we obtain, with a suitable ordering of the unknowns, a system of linear equations of the form $Ax = b$, where A is of order N^3 and is highly structured.

We show how this system may be solved using the full multigrid $V(2,1)$ method (see also Briggs [1]). The implementation is in Occam using a Meiko Computing Surface, the TDS 700D Occam Compiler, 20 MHZ. T800 transputers, with 4-cycle external DRAM and 10 Mbaud link speeds.

2 The Multigrid Method

We use the following techniques to implement the multigrid method

2.1 Red-black relaxation

This iteration is given by

$$u_{red}^{new} = \frac{1}{6}\left[\sum_{i=1}^{6} u_{black}^{old} - h^2 b_{red}\right]$$

where the summation acts over the 6 nearest-neighbour, black points. The black points are updated using a similar formula.

A full relaxation step consists of a red update followed by a black update. We use two full relaxation steps before coarse grid correction, R_1, and one after coarse grid correction, R_2.

2.2 Fine to coarse grid residual transfer

On the fine grid the residuals at all the black points are zero and those at the red points are approximately equal, when using red-black relaxation (see Holter [4]). All points which will be in the resultant coarse grid are coloured red in the finer grid. We thus use half injection for the fine to coarse grid residual transfer, i.e.,

$$R_c = \frac{1}{2}R_f$$

where R_f is the computed residual at the red points which are common to both the fine and coarse grids.

2.3 Coarse to fine grid interpolation

Two types of interpolation are required in the three-dimensional case

1. Bilinear interpolation and correction

   This is used when the error or, at the end of each V-cycle, the approximate solution is corrected. The choice of colour at which we interpolate within each V-cycle is arbitrary. We choose the points which will be black on the finer grid as this choice is a logical extension to our two-dimensional implementation. (In Holter [4] red points are chosen; we have used both approaches in sequential algorithms and found that there was little difference numerically.) The value at the new black points is approximated by half the sum of its two nearest-neighbour points on the coarse grid. There are black points which are not lying on coarse grid lines (at the centre of coarse grid cubes) and their values are computed as

   $$\frac{1}{8}\sum_{k\in P} u_k$$

   where P consists of the eight corner points of the coarse grid cube.

2. Fourth order interpolation

   This is necessary when we interpolated the first approximation on a finer grid, i.e., on our first visit to a finer grid. It is applied at the red points only and is followed by a half relaxation step at the black points. The 13-point interpolation formula is given by

   $$u_{new} = \frac{1}{32}\left[6\sum_{k\in P_1} u_k + \sum_{k\in P_2} u_k\right] - 0.5h^2 b$$

   where P_1 consists of the four nearest-neighbour points to new in the plane of interest and p_2 the eight nearest-neighbours in adjacent planes of the coarse grid. An example is shown in Figure 1 and more details may be found in Holter [4].

3 Distributing the computation

We assume that the finest grid discretization has $N = 2^m - 1$ internal points in each direction. The region is then divided into $p = 2^k$ equal parallelepipeds (the leftmost has an extra plane), where p is the number of processes to be used. In the multigrid hierarchy (i.e., for all grids) all the grid points inside each process region are assigned to

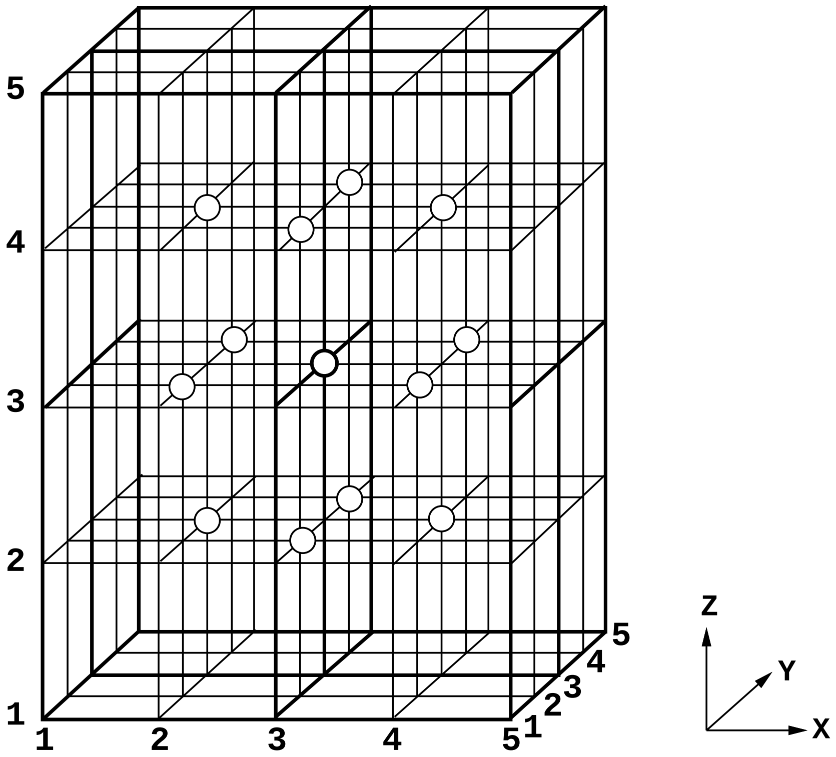

Figure 1: Points obtained in fourth-order interpolation

that process. Grid points which lie on the boundaries of process region are assigned to the left-hand process.

All processes are computationally identical and communicate with their nearest-neighbour processes to form a pipeline. This pipeline of processes is then mapped onto the available transputers. To ensure that the implementation is generally configurable (i.e., we may alter both the number of processes – and hence the number of transputers – we utilize, and also the number of levels of grid within the multigrid hierarchy without changing the code defining each process) we define three types of process which make up the pipeline

- a left-hand end process,
- a middle process,
- a right-hand end process.

Within each process all the approximate solution values and the values of the function f in (1) for all the grids are stored in one-dimensional vectors. The values are stored starting with the coarsest grid and within each grid in increasing y then z then x order. The relationship between the position in the vector and the coordinate position (ih, jh, kh) within grid l is given by

$$vector.element = N_l^2(i-1) + N_l(k-1) + j + vector.start[l]$$

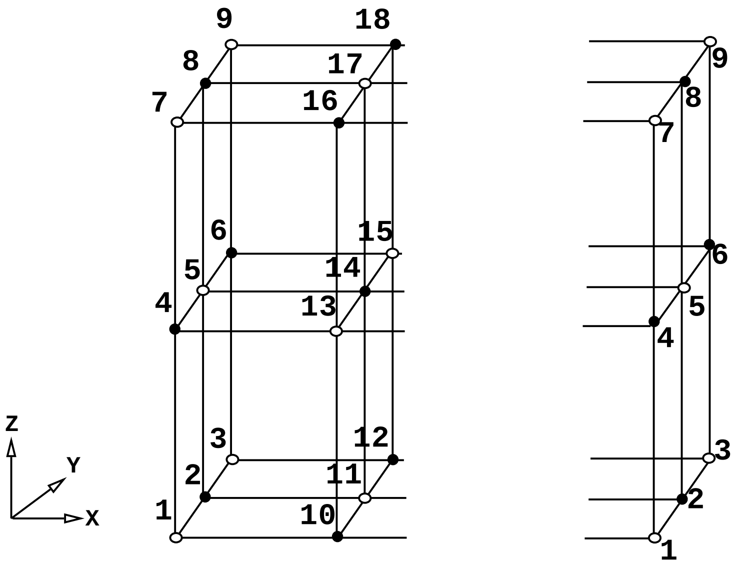

Figure 2: Ordering of points within two processors

where *vector.start* is a vector of integers giving the starting element of each grid segment within a process and $N_l = 2^l + 1$ where N_l is the number of grid points in the lth grid including the boundary points; its elements are defined by

$$vector.start[l+1] = vector.start[l] + N_l^2 H_l$$

where there are p processes in the pipeline, $H_l = \lfloor N_l/p \rfloor + 1$ for the extreme case, and for the normal and limiting cases H_l is given by $\lfloor N_l/p \rfloor$ for the right-hand and middle processes and $\lfloor N_l/p \rfloor + 1$ for the left-hand process. $vector.start[l] = 0$ for the left- and right-hand end processes. For the middle process, the initial values depends on the position of the process p_i within the pipeline.

The ordering of the points in the case of two processors and using two grid levels is shown in Figure 2.

4 The three process types

We describe briefly the main features of each of the three process types which go to make up the pipeline of processes

1. The left-hand end process

   This is always the first process in the pipeline and administers one more plane of points than the other processes. To operate, it needs to know only the number of levels of grids in the multigrid hierarchy and the number of processes in the pipeline. It has two channels to communicate with its right-hand neighbour and

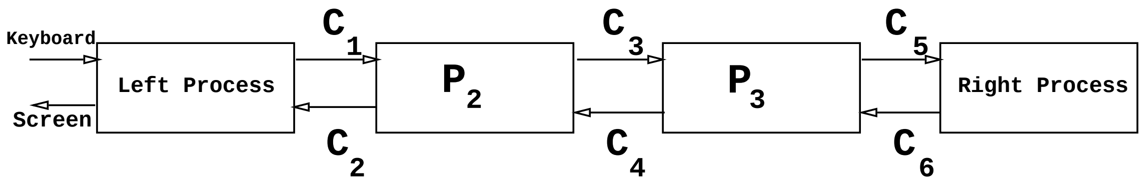

Figure 3: Processor pipeline

two to communicate with the user via the screen and keyboard. This process also collects residual and error information from the other processes at the end of each V-cycle and displays this to the user.

2. The right-hand end process

   This process always terminates the pipeline and, like the left-hand process, needs to know only the number of grid levels and the length of the process pipeline. It has only two channels with which to communicate bidirectionally with its left-hand neighbouring process.

3. The middle process

   This represents any process which is not an end process. To operate, it needs to know, in addition to the number of grid levels and the process pipeline length, its position within the pipeline. It has four channels to provide bidirectional communication with its left- and right-hand nearest-neighbour processes.

Figure 3 illustrates the communication channels and process types for a pipeline of length four; P_2 and P_3 are both middle processes. The processes within the pipeline execute each step of the multigrid algorithm in parallel. For each process and for each step of the algorithm we may consider a number of different process states depending on the current grid being worked on and the number of processes in the pipeline.

We denote by n the number of grid points in the x, y or z direction, including the boundary points, in the currently selected mesh.

1. Normal case ($\lfloor n/p \rfloor > 1$): each process region contains at least two planes of points in the current grid. The left-hand process contains at least three planes.
2. Limiting case ($\lfloor n/p \rfloor = 1$): each process region contains a single plane of points in the current grid. The same values are communicated to both neighbouring processes.
3. Extreme case ($\lfloor n/p \rfloor < 1$): some process regions contain no active points; others have a single plane of points. In this state the processes with no active points just pass data between active processes.

To illustrate the general configurability of our complementation we present an example of the main driver program. The processes *gen.left*, *gen.right* and *gen.middle* are all stored in a precompiled library.

```
-- Define method dependent constants
-- This sets up an FMG(2,1) method
VAL INT NY1 IS 2:                  -- number of relaxation steps before
                                   -- coarse grid correction
VAL INT NY2 IS 1:                  -- number of relaxation steps after
-- coarse grid correction
-- Define implementation dependent constants
VAL INT ng IS 5:                   -- maximum number of grid levels
VAL INT p IS 4:                    -- number of processes in pipeline

-- Declare all channels
CHAN OF ANY keyboard, screen:      -- to the outside world
[2p] CHAN OF ANY c:                -- for internal communications
SEQ
  PAR
    gen.left (keyboard, screen, c[1], c[2], NY1, NY2, ng, p)
    PAR i = 1 FOR (p - 2)
      PAR
        -- i + 1 denotes its position in the pipeline
        gen.middle (c[2i-1], c[2i], c[2i+1], c[2i+2], NY1, NY2, ng, p, i+1)
    gen.right (c[2p-3], c[2p-2], NY1, NY2, ng, p)
```

Array subscripts have been presented in pseudo-occam.

All pipeline processes start by calling a general set-up process which determines the points within the process region and initializes the local approximate solution and right-hand side function vectors as follows

- for boundary points set $u_{approx} = u_{exact}$,
- for internal points set $u_{approx} = 0$ (this is the initial guess at the solution) and $b = -h^2 f_{ijk}$ (these values are only computed once).

The general layout of the middle process is given as follows

```
PROC gen.middle (CHAN OF ANY outl, inl, outr, inr,
      VAL INT NY1, NY2, ng, p, position.pipe)
-- Declarations
SEQ
  ... Determine the starting positions of each grid within the
  -- one dimensional vector containing all the internal points
  -- (This depends upon ng, p and pos.in.pipe)
  ... set up u_exact, u_approx and b
```

```
SEQ lev = 1 for ng
  SEQ
  -- Descending procedures
    IF
      (lev > 1)
        ... Descending from fine grid to coarse grid
      TRUE
        SKIP
  -- Ascending procedures
    SEQ m = 1 FOR lev
      SEQ
        ... Ascending from coarse to fine grid
:
```

The ascending procedures perform

1. relaxation after coarse grid correction,
2. calculation and display of errors,
3. interpolation.

The descending procedures perform

1. relaxation before coarse grid correction,
2. residual transfer from fine to coarse grid (half-injection).

As we have mentioned previously each of these procedures needs to consider three cases, $n/p > 1$, $n/p = 1$ and $n/p < 1$. We finally present some details of the steps involved in implementing the relaxation step process.

1. $\lfloor n/p \rfloor > 1$

   Each parallel pipe contains at least one vertical y-z plane. The following steps are executed sequentially

   (a) Values at black points in the planes adjacent to the process region boundaries are exchanged with their nearest-neighbour processes,

   (b) Relaxation is performed on all red points in the process region,

   (c) Values are exchanged as in step (1a) but for the red points,

   (d) As step (1b) but for the even points

2. $\lfloor n/p \rfloor = 1$ As step (1) except that in step (1a) and (1c) the same values are passed both to the right and to the left

3. $\lfloor n/p \rfloor < 1$ Processes containing active points in their process region execute as step (1). Others just pass data from neighbouring left and right processes in the same direction.

5 Results

The resulting occam program has been tested on a number of examples of the form (1) for values of h ranging from $1/8$ to $1/32$ using transputer networks of size 2^k for $k = 0, 1, \ldots, 4$. We present just one example which is typical of the results obtained. For this example we have used

$$f(x, y, z) = -(x^2y^2 + x^2z^2 + y^2z^2)\exp(xyz).$$

The exact solution is $u(x, y, z) = \exp(xyz)$ and this has been used for the boundary values.

For a given value of N we plot the efficiency, E_{Np}, against the number of transputers in the network, p, where

$$E_{Np} = t_1/(p.t_p)$$

and t_j is the time taken to solve the problem when distributing the computation over j transputers. The left-hand graph in Figure 4 shows the results obtained when the residual and error norms are displayed at the end of each V-cycle. The right hand graph shows the effect of suppressing this intermediate output — only the final values were displayed. The increase in performance is explained by the fact that during the collection of data for displaying, processors idle after passing on all values to their right. This second graph provides the better picture of the effectiveness of the parallel implementation of the multigrid method itself.

The drop in efficiency for $N = 7$ when 16 transputers are used is to be expected since half of the processors are not doing any useful computational work; they are just passing data to their neighbours. This is a case where $n/p < 1$ for the whole computation.

The increase in efficiency for $N = 15$ and $N = 31$ when increasing the number of transputers from 4 to 8 while suppressing intermediate results is due to the fact that the amount of scalar workspace that fits into the fast on-chip memory within the transputer has increased. This means that, for example, array indexes, now reside on-chip and, for this particular problem, this provides us with a gain in performance.

Figure 5 shows the speed-up ratios for the cases $N = 7$, 15 and 31; the left-hand graph with intermediate results and the right-hand one without. The almost linear speed-up for the case of $N = 31$ is well illustrated; in fact we have speed up of approximately 13 for this case using 16 transputers and we would expect a further increase in performance

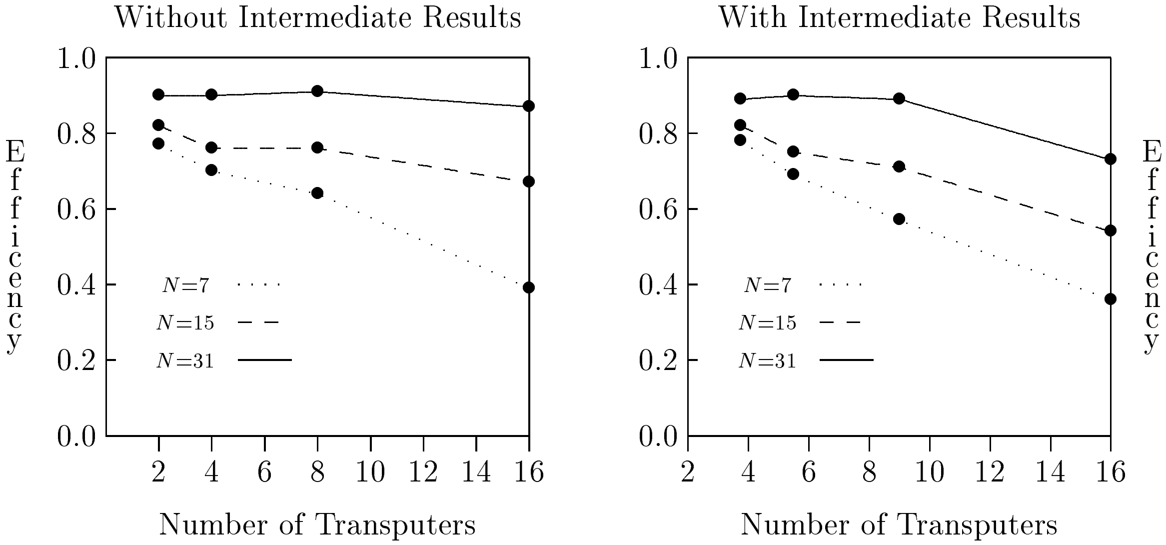

Figure 4: Efficiency plots for $N = 7, 15, 31$

for 32 transputers. On the other hand additional processors would be computationally wasted on the $N = 15$ case since they would only be acting as data passing processors with the current implementation.

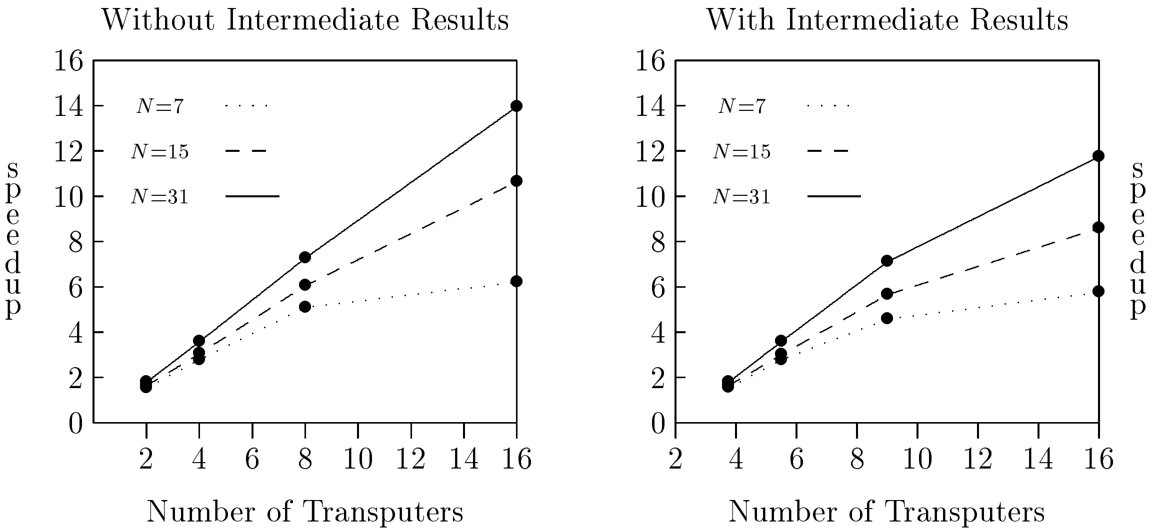

Figure 5: Speed-up for $N = 7, 15, 31$

Our experiments have been limited to small problems due to lack of memory on the transputers available to us. However, with an array of 32 transputers each with 4Mbytes of memory our implementation would allow us to solve (1) with $h = 1/128$. For a problem of this size we would expect our program to provide additional increases in performance up to pipelines of 128 processors.

6 Conclusion

We have shown that it is possible to provide a highly efficient implementation of the multigrid algorithm to solve Poisson's equation in three dimensions using arrays of transputers. This implementation is such that it is easily configurable over different numbers of processors without the need for changes to the code of the main procedures. The need for complex algorithms to combat problems with idling processors reported by McBryan and Van de Velde [5] for hypercubes and the rather poor performance reported by Shaw and Stewart [6] for two-dimensional multigrid methods in transputer arrays are avoided.

References

[1] W. L. Briggs. *A Multigrid Tutorial.* SIAM, Phildelphia, 1987.

[2] T. Chan and Y. Saad. Multigrid algorithms on the hypercube multiprocessor. *I.E.E.E. Trans. Comp.*, C-35(11):969 – 977, Nov 1986.

[3] O. El-Giar and T. R. Hopkins. A generally configurable multigrid implementation for transputer networks. In J. Wexler, editor, *Developing Transputer Applications*, pages 80–88, Amsterdam, 1989. IOS.

[4] W. H. Holter. A vectorized multigrid solver for the three-dimensional poisson solver. *Applied Math. and Comp.*, 19:127 – 144, 1986.

[5] O. McBryan and E. F. van de Velde. The multigrid method on parallel processors. In W. Hackbusch and U. Trottenberg, editors, *Proceedings of the 2nd Conference on Multigrid Methods*, Berlin, 1985. Springer Verlag.

[6] G. J. Shaw and A. Stewart. Implementation of a multigrid method on a transputer network. Technical report, Oxford University Computing Laboratory, 1989.