A Generally Configurable Multigrid Implementation for Transputer Networks

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

October 16, 1992

Abstract

This paper describes the performance of a multigrid method implemented on a transputer-based architecture. We show that the combination of fast floating-point hardware, local memory and fast communication links between processors provide an excellent environment for the parallel implementation of multigrid algorithms. The gain in efficiency obtained by increasing the number of processors is shown to be nearly linear and comparisons are made with published figures for a parallel multigrid Poisson solver on an Intel iPSC 32-node hypercube.

1 Introduction

We discuss the design and implementation, in occam, of a highly efficient, parallel, multigrid algorithm. The emphasis of this implementation is on simplicity and our program does not require the complexities of, for example, the H-Gray codes of Chan and Saad [3] or the shuffle operations of McBryan and Van de Veld [4] to obtain efficiency.

We show that there is very little loss of efficiency due either to processors becoming idle when coarser grids are being processed or in communication of data between processors. The latter is due to the ability of transputers simultaneously to compute and transfer data at a high rate (10 Mbits/sec).

Finally we present results for the solution of Poisson's equation on progressively finer grids using a Meiko Computing Surface and utilizing networks of up to 16 T800 processors. These results are compared with those obtained for a similar problem by Briggs et al. [2] using an Intel iPSC 32$-$node hypercube.

2 A Brief Introduction to Multigrid Methods

We consider Poisson's equation on a unit square with Dirichlet boundary conditions as our model problem, i.e.,

$$\begin{aligned} u_{xx} + u_{yy} &= -f \quad \text{in} \quad \Omega = [0,1] \times [0,1] \\ u &= g \quad \text{on} \quad \delta\Omega \end{aligned} \tag{1}$$

We define a regular grid of points $(x_i, y_j) = (i\Delta x, j\Delta y)$ with $\Delta x = 1/(N+1)$ and $\Delta y = 1/(M+1)$; for simplicity we also assume that $M = N$. At each internal point on this grid we define an approximation, u_{ij}, to the exact solution $u(x_i, y_j)$. The derivatives in (1) are then replaced by the simple, second-order, five point difference operator to give a linear system of the form

$$\frac{u_{i+1,j} - 2u_{ij} + u_{i-1,j}}{(\Delta x)^2} + \frac{u_{i,j+1} - 2u_{ij} + u_{i,j-1}}{(\Delta x)^2} = -f_{ij} \tag{2}$$

with $u_{ij} = g(x_i, y_j)$ for $i = 0$ or N or $j = 0$ or N, $1 \le i, j \le N-1$ and $f_{ij} = f(x_i, y_j)$. A suitable ordering of the unknowns leads to a system of the form

$$\begin{bmatrix} A & -I & & & \\ -I & A & -I & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & -I \\ & & & -I & A \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ \vdots \\ u_{N-1} \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ \vdots \\ f_{N-1} \end{bmatrix}$$

where $u_i = (u_{i1}, u_{i2}, \ldots, u_{i,N-1})^T$, $f_i = (f_{i1}, f_{i2}, \ldots, f_{i,N-1})^T$, I is a unit matrix of order $N-1$ and

$$A = \begin{bmatrix} 4 & -1 & & & \\ -1 & 4 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & -1 \\ & & & -1 & 4 \end{bmatrix}.$$

The system is symmetric and block tridiagonal with tridiagonal diagonal blocks and diagonal off-diagonal blocks.

Methods for solving such systems are divided into two categories; direct and iterative. Direct solvers determine the solution to machine precision in a finite number of steps. Examples of direct methods are Gaussian elimination and cyclic reduction. Iterative methods, on the other hand, start with an initial guess at the solution and by use of a simple updating procedure generate a sequence of approximations which, ideally, converge to the exact solution. Examples of iterative methods are Gauss-Seidel and successive over-relaxation (SOR).

Multigrid is an iterative technique which attempts to improve upon deficiencies inherent in the classical relaxation methods. These classical methods tend to reduce the error in the approximation reasonably quickly for the first few iterations after which the error decreases far more slowly. This is due to the smoothing properties of the basic iterative scheme which is very effective at reducing the high frequency components in the error vector whilst leaving the relatively low frequency components unscathed. To combat this problem multigrid methods use a sequence of grids G_k (with spacing Δx), G_{k+1} (with spacing $2\Delta x$) etc. The coarsest grid, labelled G_0, may consist of just a single unknown. Note that at each stage the points in a grid are a subset of the points in any finer grids. For many iterative schemes it is more efficient to reduce the low frequency modes by computing on G_{k-1} rather than on G_k.

Assuming for simplicity that we decide to use just two grids G_k and G_{k-1}. We first generate an initial guess to the approximate solution at each of our internal grid points on G_k. A simple iterative method (e.g., Jacobi's method) is then applied to smooth out the high frequency error modes. Using a transfer function we generate an approximation at the points of G_{k-1}. This transfer function is known as an injection and in its simplest form consists of merely selecting the subset of values from G_k. Iterating on G_{k-1} then quickly reduces the lower frequency modes and a transfer function is then used to move back to the finer grid G_k. This interpolation function needs to be carefully chosen so as neither to introduce nor magnify any low frequency errors.

In practice we use a sequence of grids G_k, G_{k-1}, ..., G_0 where, in the implemented code, G_0 consists just of a single unknown which may be solved for trivially. Hybrid methods, which use iterative techniques down to a certain grid level and then a direct method to obtain a machine accurate solution, are also possible. After any transfer between grids some smoothing of the approximation thus obtained is performed.

Thus to specify a multigrid method we need to define

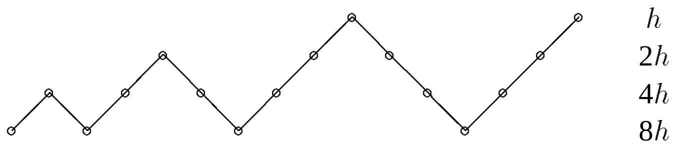

Figure 1: The $FMV(2, 1)$ schedule applied to four levels of grids

1. an injection to transfer from a fine to a coarser grid,
2. an interpolation formula to transfer from a coarse to a finer grid,
3. a smoothing technique to be applied iteratively at each step (this may be different at the interpolation and injection stages),
4. a schedule for transferring between grids.

There are many possible choices for these four stages (see Briggs [1] and Walsh [5] for examples). Our implementation uses

1. half injection,
2. bilinear interpolation,
3. red-black relaxation,
4. the $FMV(2, 1)$ schedule of grids (see Figure 1).

Details both of these terms and how the method is implemented in a sequential form may be found in the excellent introduction to multigrid methods [1].

It is important to realise that multigrid methods have been successfully generalized to more complex equations than (1); for example self-adjoint, non-linear and time dependent equations. In addition the multigrid idea has been applied to other problem areas; for example image processing, control theory and quantum electrodynamics.

3 Distributing the computation

The finest grid discretization is assumed to have $N = (2^m - 1)$ internal grid points in each direction i.e., $\Delta x = \Delta y = 1/2^m$. The region is then divided into $p = 2^k$, equal strips where p is the number of processes to be used. In the multigrid hierarchy, i.e., from the finest to the coarsest grid, all grid points inside a process region are assigned to that process. In addition grid points that lie on the boundaries of process regions are assigned to the process to the left of the boundary.

All processes are computationally speaking identical and communicate with their nearest neighbour processes to form a pipeline. This pipeline is then mapped onto the available transputers. During the initialization phase each process calculates its position within the pipeline and hence the subset of grid points it is to work with. Note that, except for the simple cases where the pipeline consists of either one or two processes, as the grid becomes coarser neighbouring points are not contained within neighbouring processes. Processes which contain no points for a particular grid are in a similar position to the sleeping nodes in the hypercube implementation described by Briggs et al. [2]. The difference is that these occam processes are required to transfer data between the processes containing active grid points – they daydream rather than develop catalepsy! Each process keeps track of the current level of calculation within the multigrid hierarchy and, since it has computed its position in the pipeline and knows the number of internal points in the finest grid, it can calculate whether it has active data or is in communication only mode.

The left-hand process is used for communication with the outside world via the two channels which are not connected to the pipeline. This process has its own subset of grid points and performs the relevant multigrid calculations. In addition, after each V-cycle this process receives error and residual information from all the other processes and display it on the user's terminal.

We present in some detail the steps involved in implementing the processes for the relaxation step and the computation and display of the error norms and residuals. The code is presented as pseudocode rather than occam for compactness.

PROC relax

- n denotes the number of grid points in the x- or y-direction including the boundary points in the currently selected mesh.
- p denotes the number of processes being used. The maximum useful value of p is N, the number of internal grid-lines in the finest mesh. If $p > N$ then some processes will idle throughout the computation.

  For each relaxation step we recognize two basically different cases

1. $(n/p \geq 1)$: each strip contains at least one vertical line of points. The following four steps are executed sequentially within each process

   (a) values adjacent to the boundaries of process regions are exchanged in parallel with their nearest neighbour processes. Note in the limiting case $(n/p = 1)$ the same values are sent both to the right and the left,
   (b) relaxation is performed on all even points including those adjacent to the boundaries of the process regions,
   (c) exchange as in step 1a,
   (d) as step 1b using odd points.

2. $(n/p < 1)$ some processes contain no active grid values whilst others are computing with a single vertical line of points.

   if $(n = 3)$ – we have reached the coarsest grid

   - middle process $(p_n = p/2)$
     - receives, in parallel, single values from processes $p_n \pm 1$,
     - solves directly for the single unknown.
   - processes to the right of centre $(p_n > p/2)$
     - receive boundary value from right-hand boundary and pass to left.
   - processes to the left of centre $(p_n < p/2)$
     - receive boundary value from left-hand boundary and pass to right.

   if $(n > 3)$

   - for processes working with active points as 1 above,
   - for processes containing no active points
     - in parallel receive from left and right and send to right and left.

PROC calc_and_display_errors

- we only present the case for $n/p \geq 1$ the extra logic required for the other case is similar to the relaxation process. The left-most processor requires some extra code as explained below. The following steps are performed in sequence

  1. Exchange values as in step 1a in the relaxation process.
  2. Compute residuals and error norms.
  3. Send required values to the left-hand process.

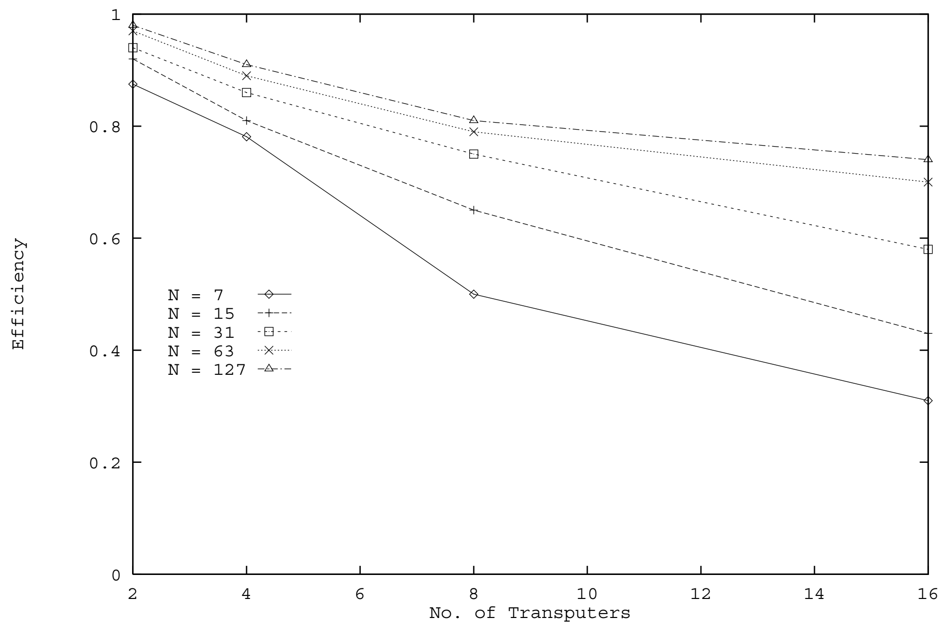

Figure 2: With intermediate results – $N = 7, 15, 31, 63, 127$

4. In parallel, receive data from the right hand process and send it on to the left-hand process, until all data computed to the right of the process has been transmitted.

The left-most process needs only two channels to communicate with the pipeline of processors; the other two are connected to the user's display and keyboard. This process receives all the error and residual values and computes the necessary maximum values for display.

The interpolation and restriction processes have a similar structure to the relaxation step above.

4 Results

The resulting occam program has been tested on a number of examples of the form (1) on grids with step sizes ranging from $1/8$ to $1/256$ 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 $f(x, y) = 2\pi^2 \sin \pi x \sin \pi y$ and $g = 0$ in (1) above. The exact solution is $u(x, y) = \sin \pi x \sin \pi y$.

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 execute the program when distributed over j transputers.

Figure 2 shows the results when the residual and error norms are displayed at the end of each V-cycle. Figure 3 shows the effect of suppressing this intermediate output – only the final values are 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. The second graph gives a better picture of the effectiveness of the parallel implementation of the multigrid method itself.

We note from these figures the dramatic decrease in efficiency when 16 transputers are used to solve the $N = 7$ and $N = 15$ problems. This is because some of the processors are idling for a large proportion of the computation. For the largest problem, $N = 127$, the curve is very flat showing the very high performance gains obtained by increasing the number of processors. It can be seen that the overhead of computing on the coarser grids is extremely small. The decline in performance caused by the idling of processors during the coarser grid computation can just be seen for the $N = 63$ grid using 16 transputers.

Figure 4 shows a comparison graph for a solution of Poisson's equation on the Intel hypercube taken from Briggs et al. [2]. No details of the actual problem solved are given in this reference. However it is clear that there is a far more marked degradation in performance as more nodes are introduced into the solution than for the transputer array.

5 Conclusion

We have presented an occam implementation of a multigrid algorithm which makes very efficient use of the available processors. It is especially efficient for larger problems when the increase in performance becomes almost linear with the number of extra processors.

The implementation of the multigrid algorithm as a general occam process means that the program is very easily adaptable to transputer pipelines of different sizes. Having defined the equation to be solved only the number of processors

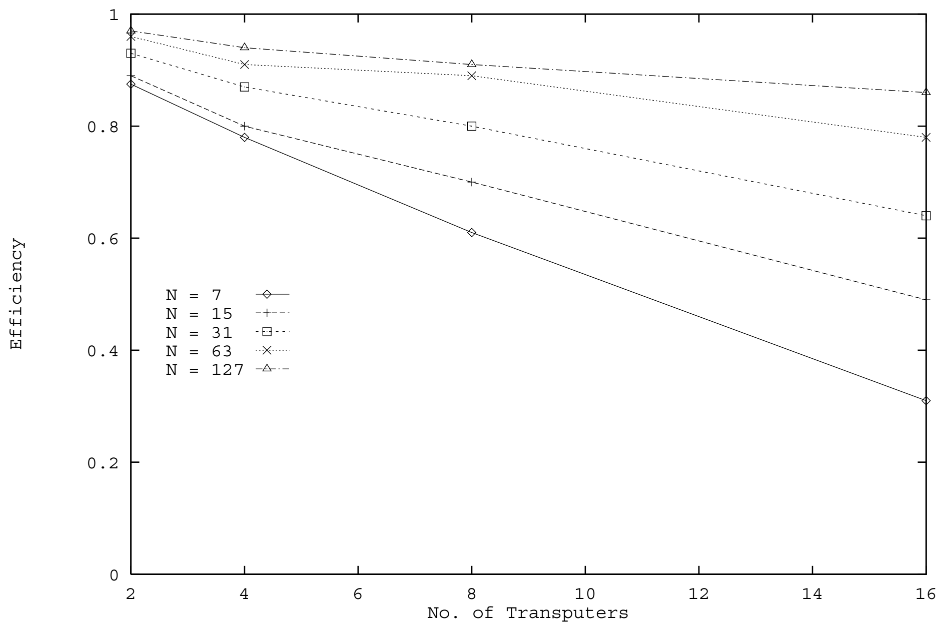

Figure 3: Without intermediate results for $N = 7, 15, 31, 63, 127$

available and the size of the finest grid are required to start the program running. No changes to the code are required.

With the large increases in the external memory now available on transputers the approach taken in this paper would appear to be ideally suited to the solution of three dimension problems. Code for this problem is close to completion.

6 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 thing to do!

References

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

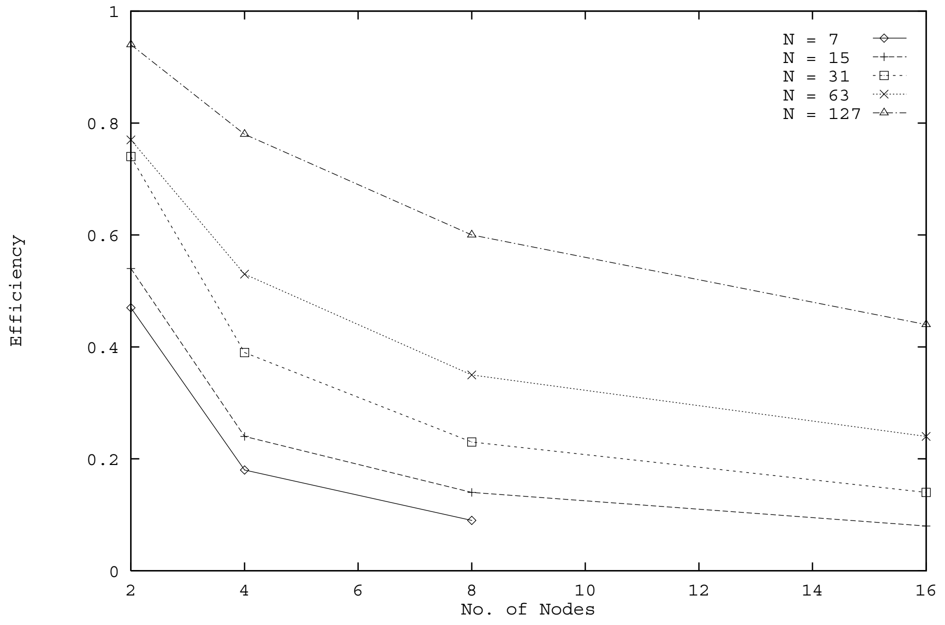

Figure 4: Efficiency vs. number of hypercube nodes for $N = 7, 15, 31, 63, 127$

[2] W. L. Briggs, L. Hart, S. F. McCormick, and D. Quinlan. Multigrid methods on a hypercube. In S. F. McCormick, editor, *Multigrid Methods – Theory, Applications, and Supercomputing*, pages 63 – 84. Marcel Dekker, Inc, Basel, 1988.

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

[4] 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.

[5] J. Walsh. Multigrid methods for elliptic equations. In A. Iserles and M. J. D. Powell, editors, *The State of the Art in Numerical Analysis*, Oxford, 1987. Clarendon Press.