Kron's Method of Tearing on a Transputer Array

K. BOWDEN

Computer Centre, Polytechnic of East London, Longbridge Road, Dagenham, Essex RM8 2AS

Kron's Method is a means of decomposing physical systems into subsystems with the particular property that the subsystems overlap on their boundaries. It is found that the solution of typical engineering problems is facilitated by this decomposition or tearing. Computational time and memory requirements are both improved, and the technique has the advantage that the individual subsystems can be resolved under change of input conditions or structure and the overall solution updated without the need for a global resolution. The technique is efficient on sequential machines but is ideally suited to MIMD (multiple-instruction multiple-data) machines such as those that can be constructed from the Inmos transputer. This paper discusses the implementation, in Occam, of direct, non-iterative methods of solution of torn systems on a network of pipelined transputers. In particular the software has been used to solve Laplace's equation on a finite difference mesh.

Received September 1989, revised May 1990

1. INTRODUCTION

A number of workers have recently been solving field problems on transputer arrays using variations on the finite difference and finite element methods. The former method, and sometimes the latter, traditionally use the iterative (indirect) method of solution called relaxation. The iteration converges stably towards an exact answer and thus takes an amount of time related to the required accuracy of the solution. Conversely (if the equations are linear) direct methods involve either the solution of a set of simultaneous equations by, for example, the Gauss–Seidel method, or a matrix inversion. These methods are exact and 'finite' in that the solution time is not related to the accuracy required, which depends only on the wordlength of the machine in use and the numerical stability of the algorithm. Matrix inversion is, in general, slower than Gauss–Seidel and involves the storage of a (very large) matrix inverse – hence it is almost never used in practice – but it does have the advantage that, once the inverse is known, solutions can be obtained for any number of sets of boundary conditions (right-hand sides). Newer direct methods such as the Marching Algorithms[1,2,6] are quicker and require less storage but are numerically unstable.

To solve these problems on a distributed array of processors the technique used by most investigators is to decompose the problem geographically into a number of adjacent regions which are then mapped on to adjacent processors in the computational array. A small area of overlap is defined between the information held on adjacent processors (for an n-dimensional problem this region may be n-1-dimensional and will normally constitute at least the boundary of the region). This, in fact, is a special case of the technique known as Kron's method of tearing. A global iteration is then performed, each processor calculating a solution for the region it represents, then passing the solution for the area of overlap to its neighbours, receiving in turn their version of the solution. The process is repeated until, hopefully, it converges to a solution. Thus this is an indirect global solution. This paper investigates the possibility of using a direct global method, based on Kron's method of tearing, on a transputer array, and suggests variations of it which may overcome the problems encountered. The processor topology does not map directly on to the physical decomposition of the system itself but to the form of the algorithm. This is known as structural decomposition. It is interesting that Wait,[4] champion of the iterative schemes, comments that 'iterative methods can never be as robust as direct methods'. Also that he finds that the iterative solution converges better if the overlap between subsystems is more than one element thick.

Kron's method of tearing, or Diakoptics, as he called it,[3] was invented and developed by Gabriel Kron in the 1950s and 1960s. It was intended both as a general philosophy and as a method of solution of general physical systems. Kron showed that by splitting up a system into a number of parts, solving the problem on each part separately, and then combining the subsystem solutions into an overall solution, an exact answer could be obtained, with a saving both of computational time and storage space over a direct matrix inversion, even on a sequential computer. Kron died in 1968, and interest in his research died out in the following decade. This author has been involved in a study of Kron's method for a number of years, and realised that with the advent of MIMD (multiple-instruction, multiple-data) distributed machines such as the Transputer array, Kron's method was the ideal form of solution for these architectures, as a direct mapping from the decomposed, or torn, problem to the processor array can be achieved. Moreover, a further saving in computational time can be achieved of just under n times, where n is the number of processors. A particular feature of Kron's method is the area of overlap between adjacent regions, which he called the intersection network (the method was originally aimed at the solution of torn electrical networks). The solution of the intersection network has to be achieved before the subregions can be solved, thus an increase in speed of n can never quite be achieved. Kron's method would normally involve the direct solution of a set of matrix equations, including the inversion of the subsystem matrix for each region and the inversion of the system matrix of the intersection network. The solution method, however, was not specifically defined by the method of tearing (in order to cater for nonlinear systems), so that

the indirect global methods of solution mentioned in the previous paragraph, such as Wait's,[4] are actually special cases of Kron's method.

As stated above, Kron's method was designed to be a general method of decomposition and solution of physical systems (i.e. ones with a topology of some sort); however, this paper looks at both the general case and the solution of a particular field problem, that of the Laplacian on an area shaped like a Maltese cross. The problem will be divided into four regions – four identical ones and a square central region. The latter overlaps (or touches) the four surrounding regions on each of its edges. The solution will be by direct solution of the finite difference equations, involving discretising the problem by overlaying it with a regular Cartesian grid. We will not go into a discussion of computational times, as they can easily be estimated for direct methods for any particular problem, but will concentrate on a discussion of how the problem was implemented on a five-transputer array (obtained as part of the 1988 SERC loan scheme), and how the method could be developed to the stage where it could become a practical solution scheme competing with the currently more efficient iterative methods, and the advantages that would obtain if this were possible.

2. THE ALGORITHM

It is well known that the solution of Laplace's equation involves the inversion of a matrix of the following form (or the solution of the associated set of equations).

$$A=\begin{bmatrix} 4 & -1 & 0 & & \dots & \\ -1 & 4 & -1 & & \dots & \\ 0 & -1 & 4 & -1 & \dots & \\ & & -1 & 4 & -1 & \dots \\ & & & \dots & & \\ & & & \dots & -1 & 4 \end{bmatrix}$$

For a region decomposed into subproblems the A matrix becomes reordered. Instead of just scanning along the points of the grid to choose the order in which the equations are written down, points in each subregion are grouped together. This results in a block diagonal system matrix, where each block has the form of A above. Finally, we group the elements of the intersection network, the set of all subsystem overlaps, as the last group of equations with system matrix B. The interconnections between this group and the subsystem equations are given by the elements of the subsystem matrices shown as C below. The system equations of a problem decomposed into n subproblems are as shown below.

$$\begin{bmatrix} A(1) & & & & & C(1) \\ & A(2) & & & & C(2) \\ & & A(3) & & & C(3) \\ & & & \dots & & \dots \\ & & & & A(n) & C(n) \\ C(1)' & C(2)' & C(3)' & \dots & C(n)' & B \end{bmatrix} \begin{bmatrix} x(1) \\ x(2) \\ x(3) \\ \dots \\ x(n) \\ y \end{bmatrix} = \begin{bmatrix} b(1) \\ b(2) \\ b(3) \\ \dots \\ b(n) \\ c \end{bmatrix}$$

This form can be obtained for any linear system which can be physically decomposed into adjacent subsystems and where the equations are essentially local. We term this a physical system. These conditions are related to Huygens' principle and to holography and are necessary and sufficient for the applicability of Kron's method. They will be discussed in detail elsewhere. Such problems are found in all branches of engineering and physics and even in economics. Kron derived a very efficient direct solution for these as follows. Writing the equations as

$$\begin{bmatrix} A & C \\ C' & B \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} b \\ c \end{bmatrix}$$

where A, b, x and C are the appropriate block matrices, the intersection matrix y is given by

$y=/B(c-C'x)$ where

$x=/A(b-Cy)$

and $/A$ is the normal matrix inverse, from which it is easy to show that

$y=/(B-C'/AC)(c-C'/Ab)$ and

$x=/A(b-Cy)$ as above.

So by calculating the intersection first, we are led to a solution for each $x(i)$ in terms of y. Expanding back out we get

$$y=/\left(B-\sum_{i=1}^{n} C(i)'/A(i)C(i)\right)\left(c-\sum_{i=1}^{n} C(i)'/A(i)b(i)\right)$$

and

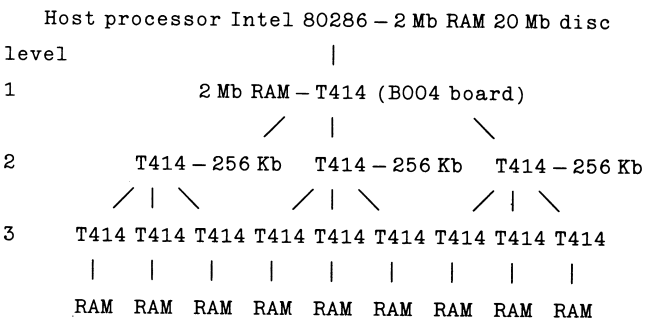
$x(i)=/A(i)(b(i)-C(i)y)$ for $i=1,\dots,n$,

which are the equations we shall implement on our transputer array.

3. THE IMPLEMENTATION

The system with which we were supplied consisted of an IBM PC AT clone made by Tandon hosting a Microway B004 board with one T414 and 2 Mb memory and a B003 board with four T414s each with 256 Kb. The TDS operating system with its folding editor along with Occam II, FORTRAN, C and Pascal compilers and harness software and manuals were also supplied. The algorithm was coded in Occam. As each transputer has four links which can be connected to its neighbours, the first problem was to decide on the best processor topology for implementing the above equations. The only constraint is that the host processor in the PC (which simply acts as a front end for input and output of information, i.e. screen, keyboard and disc handling) must be connected to the transputer on the B004 board, which can then be

hard-wired to the other processors in any topology desired by the user. Two versions of the software were implemented. In the first n was equal to 5 and each subsystem was mapped on to a different processor. This was inefficient in that some of the subsystem matrices are identical, and thus different processors were simultaneously doing identical jobs during the solution process. In the second version the user was allowed to specify that some of the subsystems were identical, and the system would only solve each individual system matrix once, thus leaving three processors unused in our example problem. The hardware we had available would thus solve any problem with up to five different subsystem matrices. The techniques described in this paper will



work on any MIMD processor array, but our description will be limited to our implementation on the one described above.

Topologies which are in common use for this sort of problem include trees, square meshes, pipelines and rings. The square mesh is the usual topology for processor arrays for the global indirect solutions mentioned above. The region itself is decomposed using a square mesh and then the subregions map naturally on to the Transputer network. Experience with tree structures (each subtree can have three branches) reportedly has not been very satisfactory. The decomposition topology for the region to be solved by Kron's method is arbitrary and need not map in a geographical kind of way on to the processor array. Investigation of the equations above show the following.

(1) A series of subsystem matrix inverses should be carried out in parallel if possible and the matrices $C(i)'/A(i)\,C(i)$ and $C(i)'/A(i)\,b(i)$ formed.

(2) The two summations must be carried out. The data for each subsystem are initially local to its own processor (MIMD systems have no global memory).

(3) The intersection vector y must be formed on some processor in the system and then transmitted to every other processor ready for the final stage (or alternatively calculated simultaneously on each processor).

(4) The subsystem solutions $x(i)$ are formed locally in parallel and then transmitted along with y to the host processor for output.

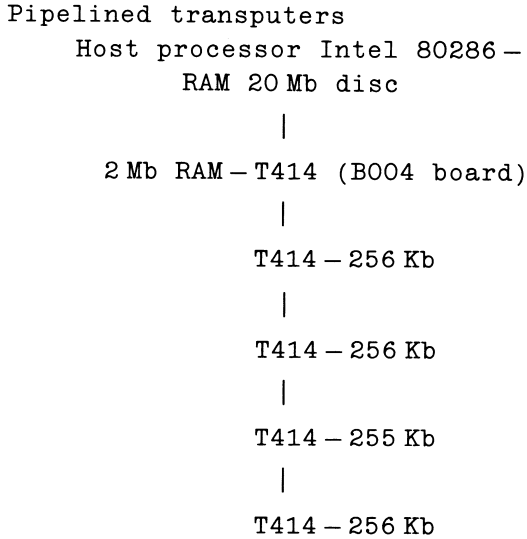
We could add an initial stage.

(0) The subsystem matrices and connection matrices must be set up and transmitted to the local processors.

It should be noted that as the connection matrices $C(i)$ consist largely of zeros and otherwise of ones, they were not stored explicitly but only as an incidence table. Thus multiplication by $C(i)$ consists of a reordering of the multiplicand, implemented simply by adding a level of indirection.

It would seem from first sight that a tree structure would be ideal for this problem. The main communication tasks involved in the above process are: getting the initial subsystem information out to the local processors; getting the matrix inverses back to the main processor (B004) for summation (if that is where we choose to do it); getting the intersection vector back to the local processors and getting the subsystem solutions back to the main processor. (Note that due to the shortage of processors it was decided to use the main processor as a subsystem solver as well.) A tree-structure topology would look like this (in the general case).

The system shown has three levels and 13 transputers; our system would consist of the first two levels of this network. It can be seen that getting data to and from the local processors would be fairly efficient with the high degree of connectivity in this system. In fact it can be seen that the bottleneck in this system is getting data in and out of the main (B004) processor. Extrapolating from this it can be seen in fact that during the data communication periods of the algorithm there is no advantage in having any topology with a higher degree of connectivity than a pipeline. The main processor churns out information. The others either take it in or pass it on. Provided the algorithm for making this decision is extremely simple a pipeline is a good topology. It also turns out that a pipeline is an excellent topology for carrying out the summations, which is the other non-local procedure that we must perform.



As an aside it should be pointed out that there is another way in which the approach taken in this work is very different from that of other algorithms implemented on transputer networks. The conventional approach to all this multiprocessor computing and information transmission is to have a number of processes running on each processor. So in the pipeline we could have one processor reading data, one deciding what to do with it, one sending data off to the next processor in the chain and others performing computations. This approach is very necessary for non-finite, iterative algorithms in which the order of computation is often influenced by the data itself. It is also extremely robust and proof against a number of different kinds of error, if possibly rather

inefficient. However Kron's method is finite, with a well-defined order of operations. Thus, provided there are no errors in the program or data, the code will always be executed in the same order and we may use only one strictly defined process on each processor. Of course if anything goes wrong with this system it will crash in the normal impenetrable way that transputers do, making debugging the software extremely difficult. However, once it works the code will be extremely efficient because there are not the overheads of running multiple processes. There are no decisions about which data to keep and which to throw away. Everything is known in advance.

Back to the pipeline and consider the summation process. We have to add a number of equally sized matrices on a pipeline of processors with local memory. It is easy to see that the following process is optimal.

(1) Pass the first element from the first processor to the second and add (to the first element).

(2) Pass the second element from the first processor to the second processor and add (to the second element). Simultaneously pass the first element (already a sum) from the second processor to the third and add.

(3) Pass the third element from the first processor to the second processor and add (to the third element). Simultaneously pass the second element (already a sum) from the second processor to the third and add. Simultaneously pass the first element (already a sum) from the third processor to the fourth and add.

(4) And so on down the pipeline until all the processors are working flat out simultaneously. Pass, add, pass, add. Continue until the (m,m)th element where $m \times m$ is the size of the arrays. The first processor will stop transmitting. The second processor will stop processing, followed by the third and fourth and so on down the line until finally the required sum is stored in the last processor in the chain.

In practice it was arranged that this process was performed with the processors in the reverse order to that described so that the sum ended up in the main processor. A similar procedure was then carried out to obtain the sum of the $C(i)'/A(i)\,b(i)$.

So the main loop on the main processor looks like this

```
getsubsysteminverse(z,m[4],n[4])
i:=0
WHILE i<ni
  SEQ
    j:=0
    WHILE j<ni
      SEQ
        chan0 ? zi[i][j]
        --get element from pipeline on
        --channel(0)
      IF
        c[i]=0
          SKIP
        TRUE
          IF
            c[j]=0
              SKIP
            TRUE
              zi[i][j]:=zi[i][j]-
                  z[c[i]-1][c[j]-1]
                  --build C'/AC
      j:=j+1
  i:=i+1
```

which is the last processor in the pipeline. A similar bit of code for $C(i)'/A(i)\,b(i)$ contains the line

```
bi[i]:=bi[i]+(z [c[i]-1][j]*b[j])
    --build C'/Ab
```

For the intermediate three processors the code looks like this

```
getsubsysteminverse(z,m[k],n[k])
i:=0
WHILE i<ni
  SEQ
    j:=0
    WHILE j<ni
      REAL32 zi:
      SEQ
        chanj ? zi
          --get element from
          --pipeline
        IF
          c[i]=0
            --if no entry in C
            --matrix
            chanjm1 ! zi
              --then pass straight on
              --down pipeline
          TRUE
            IF
              c[j]=0--ditto
                chanjm1 ! zi
              TRUE
                chanjm1 !
                zi-z[c[i]-1][c[j]-1]
                --else build C'/AC
        j:=j+1
    i:=i+1
```

where chanj is channel (j) and chanjm1 is channel $(j-1)$. The code for the tail (i.e. first) processor in the pipeline is

```
getsubsysteminverse(z,m[0],n[0])
i:=0
WHILE i<ni
  SEQ
    j:=0
    WHILE j<ni
      SEQ
        IF
          c[i]=0
            channm2 ! zi[i][j]
          TRUE
            IF
              c[j]=0
                channm2 ! z[i][j]
              TRUE
                channm2 ! (zi[i][j]-
                  z[c[i]-1][c[j]-1]
                  --build C'/AC
        j:=j+1
    i:=i+1
```

where channm2 is channel $(n-2)$. Fig. 1 shows the processor interconnections with the actual Inmos channel and link assignments. Fig. 2 shows the actual system

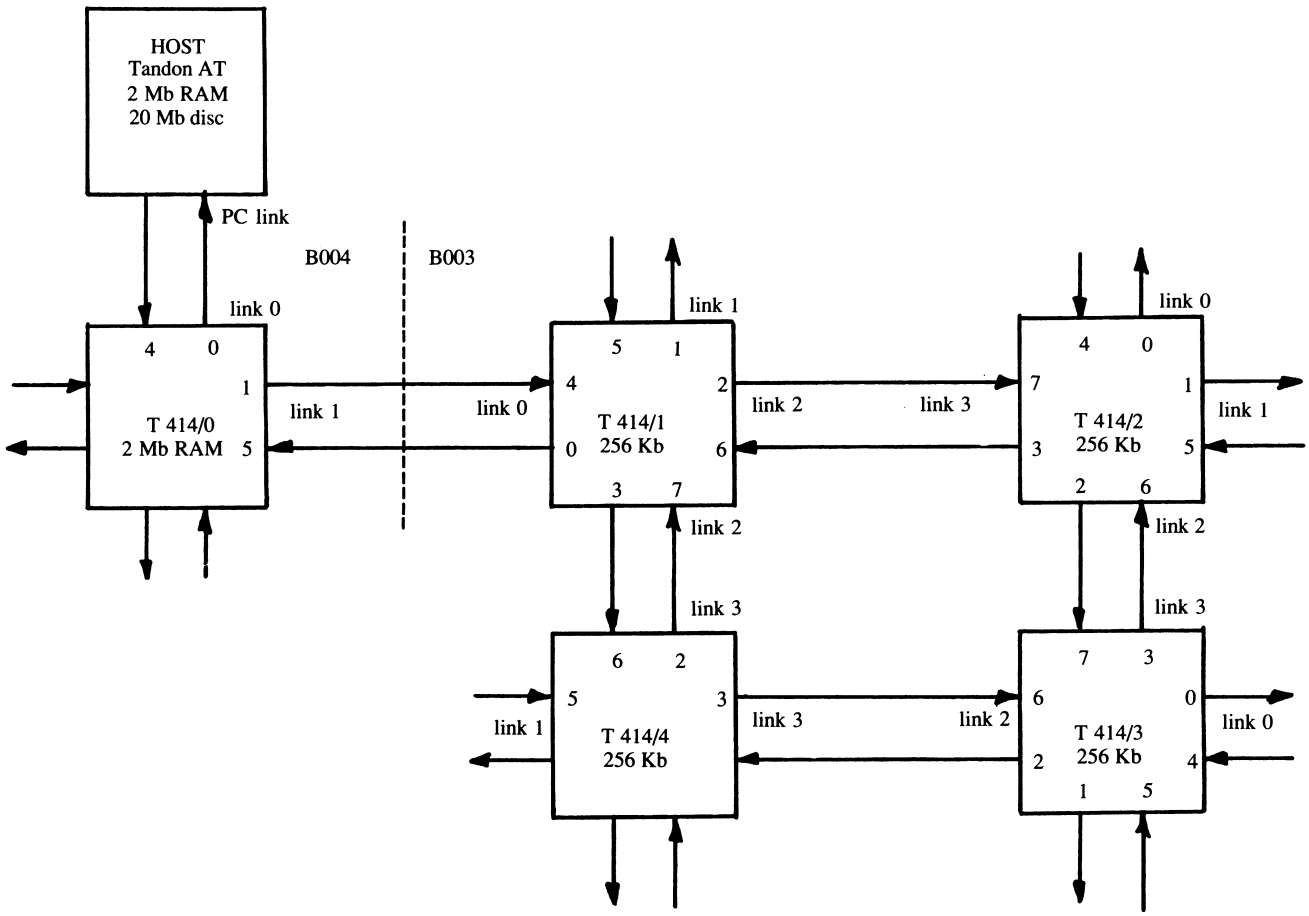


Figure 1. The actual hardware configuration used. The link between T414/1 and T414/4 was hard-wired on to the board but was not used by the algorithm.

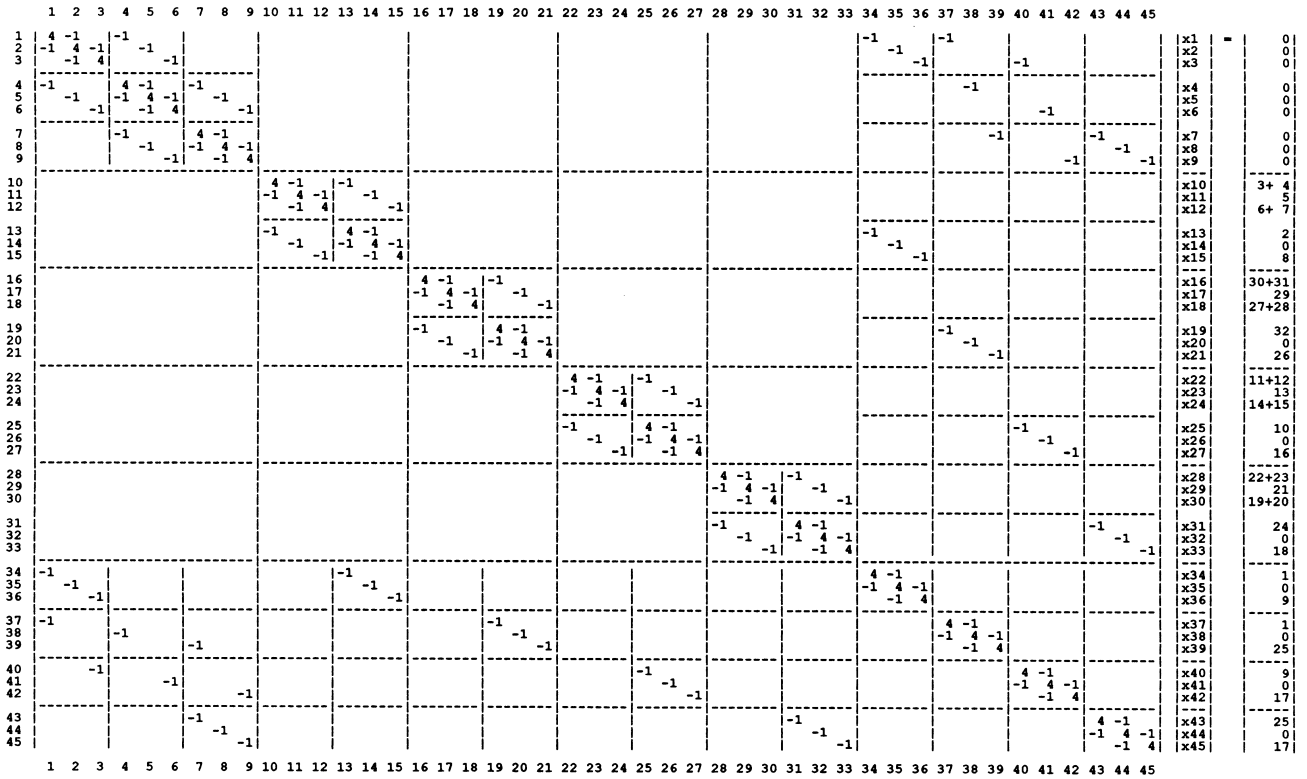


Figure 2. Finite-difference equations for the Laplacian on a 2-D cross.

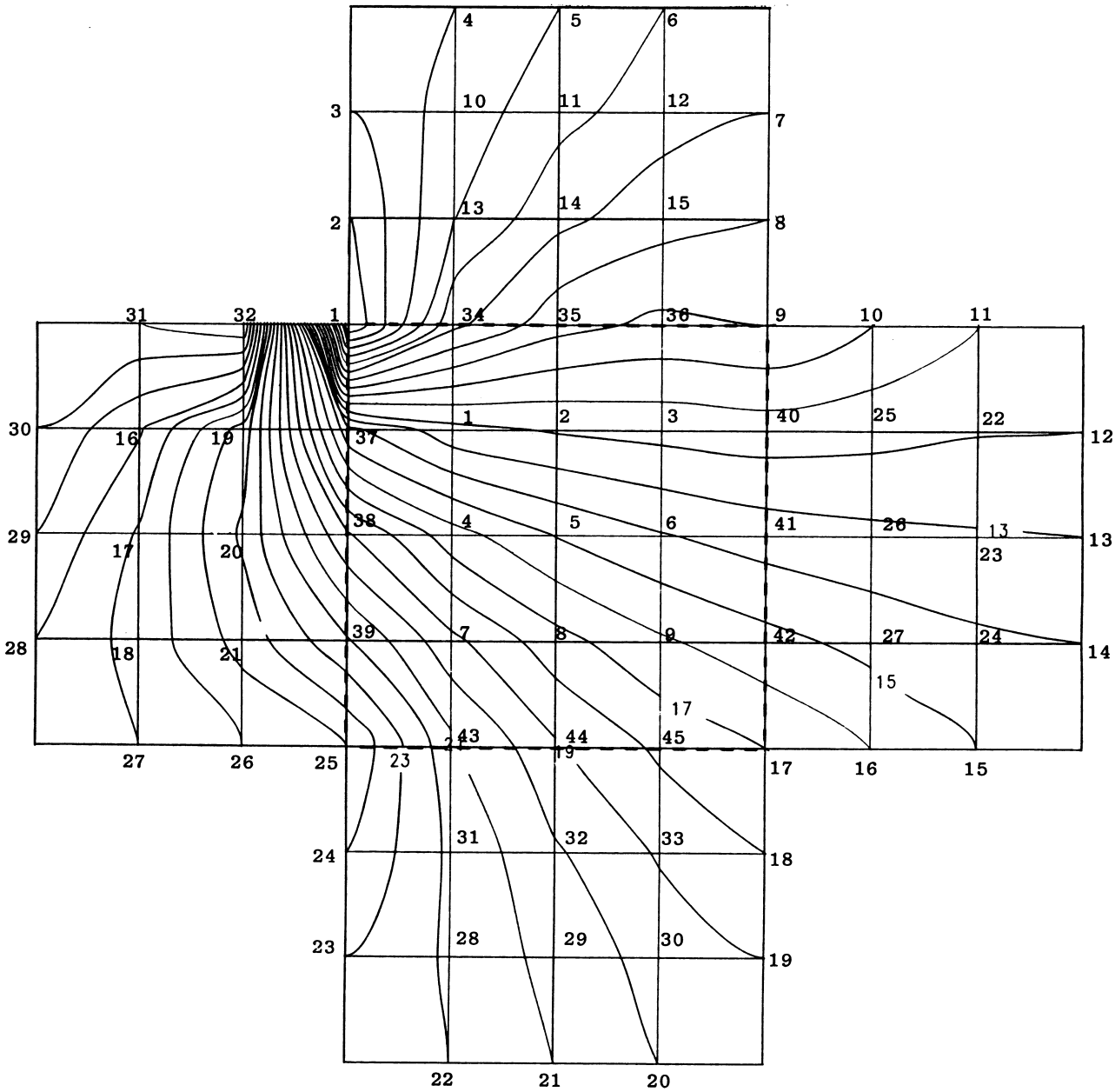


Figure 3. Topology and contoured potentials. The diagram shows the finite-difference grid with 45 nodes superimposed on a Maltese cross shape. The boundary conditions consist of a ramp potential rising clockwise round the cross. The system was torn into five pieces along the dashed lines. The solution grid is shown contoured by the NAG graphics library.

matrix for the Maltese cross and Fig. 3 shows the physical configuration and potential solution.

4. CONCLUSIONS

The program implements Kron's method of tearing on a five-transporter array connected in a pipeline. It currently solves Laplace's equation on a Maltese cross by the finite-difference method with 45 nodes. The boundary conditions consist of a linear ramp potential travelling around the boundary and ending in a discontinuity. The system is torn into five parts, each of which is solved separately and in parallel, and then the overall solution is reconstructed from the parts. The method worked well with an improvement in computational time of almost n over purely sequential code, as was expected. A second version of the program allows for the fact that some of the subsystems may be identical and, in the case of the Maltese cross solves the problem with only two different subsystems on two processors.

The great problem with direct methods such as this is the creation and storage of large subsystem matrix inverses. This is alleviated by the existence of large numbers of identical subsystems as in our example, but exacerbated if this situation cannot be achieved, as in systems with complicated or irregular, non-symmetric boundaries. If it were possible to calculate just an element, or even a submatrix of the subsystem inverse

efficiently, the problem would be alleviated. In fact a formula for the ijth element of the inverse of the matrix

$$A=\begin{bmatrix} 4 & -1 & 0 & & & \dots & \\ -1 & 4 & -1 & & & \dots & \\ 0 & -1 & 4 & -1 & & \dots & \\ \dots & & -1 & 4 & -1 & \dots & \\ & & & \dots & & & -1 \\ & \dots & & & & -1 & 4 \end{bmatrix}$$

is not hard to find, although a solution of this form for

$$\begin{bmatrix} A & -I & & & \dots \\ -I & A & -I & & \dots \\ & & \dots & & -I \\ \dots & & & -I & A \end{bmatrix}$$

has not been found.

However there are a set of techniques published by this author and others, known as marching algorithms,[1,2,6] which have been found to help further as far as computational time is concerned. In essence they give a stepwise, direct, recursive solution to the block inverse above, based on Chebyshev-like polynomial matrices. The process works quite well for the solution of rectangular finite-difference blocks. Unfortunately, however, the numerical accuracy decreases dramatically with the number of steps, and more than twenty steps is impracticable with the word length of typical modern computers. This problem can be overcome by tearing the problem into strips twenty mesh points wide or less. The length of the strips (and hence the size of the submatrices) is not important. The algorithm described above has been modified to use the marching algorithm to solve the submatrices and found to work satisfactorily on systems torn into narrow sections as described.

The marching algorithm, however, still does not solve the problem of storing the subsystem matrix inverses. Furthermore, it is only applicable to finite-difference-type systems, and we are looking for a general algorithm suitable for any of the wide range of systems to which Kron's method is applicable. Kron's method essentially uses Schur's lemma to find the inverse of the partitioned-system matrix. Assuming matrix inversion takes longer than matrix multiplication, it is easy to show that for a square two-dimensional region the optimal number of subsystems to minimise solution time using Schur's lemma is in the order of $m^{\frac{1}{3}}$ where m is the total number of mesh points. (This is assuming that we can make the subsystems all identical.) So for a 1000×1000 grid the optimal number of subsystems is $1{,}000{,}000^{\frac{1}{3}} = 100$, i.e. a 10×10 array of subsystems of 99×99 points each (allowing 1 for the intersection). Remarkably this formula appears to be largely independent of the matrix inversion algorithm. It turns out, however, that rather than dividing the system into lots of little bits, it is more efficient to initially tear into a few large subsystems and then tear these up into smaller ones and so on. The optimal level of nesting depends in general both on the problem and on the hardware available to solve it. A (non-trivial) algorithm for n-level tearing has been derived and will be published elsewhere. It is hoped that future work will include the implementation of this on a transputer array.

Finally it is hoped that, in the future, techniques developed and described in this paper could form part of an analysis package linked to an engineering draughting (electrical, mechanical or otherwise) system. Many modern draughting packages build up complex assemblies from discrete system parts, rectangles, boxes, circles and spheres, etc. Stress analysis (for example) on these simple components is relatively easy. Component solutions can then easily be combined to give overall system solutions. Indeed, libraries of components with their solutions could be built up ready to assemble into products for which overall solutions would be immediately available.

Acknowledgement

Thanks to Smita Patel for so efficiently coding the marching algorithm.

REFERENCES

1. K. Bowden, A direct solution of the discretised form of Laplace's equation. *Matrix and Tensor Quarterly*, pp. 93–96 (March 1982).
2. K. Bowden, A direct solution to the block tridiagonal matrix inversion problem. *International Journal of General Systems* **15**, 185–198 (May 1989).
3. G. Kron, *Diakoptics: The Piecewise Solution of Large-Scale Systems*. Macdonald, London (1963).
4. R. Wait, Overlapping block methods for solving tridiagonal systems on transputer arrays. Thesis, University of Liverpool (December 1987).
5. T. Durham, Working out the algebra of algebras. *Computing* (19 March 1987).
6. R. E. Bank and D. J. Rose, Marching algorithms for elliptic boundary value problems. I. The constant coefficient case. *SIAM Journal of Numerical Analysis*, **14** (5) (1977).
7. K. Bowden, Homological structure of optimal systems. Thesis, University of Sheffield (December 1983).
8. P. Atkin, *Performance Maximisation*. Inmos Technical Note 17 (March 1987).