Stanford Exploration Project, Report 70, December 18, 1997, pages 1–16

A finite element code on the Connection Machine

Dave Nichols[1]

keywords: *algorithm, finite element, boundary conditions, modeling*

ABSTRACT

I have written a finite element code for the Connection Machine (CM) that models heat flow in a circuit element. The code uses the conjugate gradient method to solve the global Galerkin matrix associated with the finite element problem. The design of the algorithm depends on the connectivity of the elements. For regularly connected elements, the application of the global matrix can be coded as a banded matrix multiply. This is very efficient on the CM. If a regular mesh is used the solution of the finite element system runs at 275MFlops on 4K CM. For irregularly connected elements the communication is more complicated and the global data transfer is much more difficult to implement efficiently on the CM. The basic design considerations that apply to this simple problem should be the same for more complicated problems such as solution of the wave equation.

INTRODUCTION

The finite element (FE) method (e.g., Johnson, 1987) is well suited to problems with irregular computational domains and problems that have solutions whose smoothness varies greatly from one part of the domain to another. It has these properties because the approximating solution is expressed as a superposition of basis functions with local support. Irregular domains can be accommodated because the domain is divided into elements that need not be laid out on a regular grid. A higher density of elements can be used where the solution is complex and a lower density can be used where the solution is smooth. This is in contrast to the finite difference method that uses a regular grid with the same spacing throughout the computational domain.

The FE solution is expressed as a function of the values at nodal points. The nodal values are obtained by solving a sparse set of linear equations. The structure of the sparse, symmetric operator is controlled by the layout of the elements and nodes. The dimension of the operator is the number of nodes in the problem. A row of the operator corresponding to one node will have non-zero entries in the columns

[1]**email:** not available

corresponding to nodes that share a common element with that node. In this paper I will show that the regularity of the layout of the elements is an important factor in the efficiency of the method on a CM.

I illustrate the method with a model problem. In the model problem I use rectangular elements with bilinear basis functions. The solution of this problem on a CM requires a different approach to that used for vector processors. I am able to make efficient use of the CM by using the conjugate gradient method (e.g. Golub & Van Loan, 1989). The speed of the solution appears to be controlled by the speed at which the CM can execute a dot product of two arrays.

If the elements are not laid out on a regularly connected mesh the operator is more expensive to apply on a CM. However, more spatially complex problems can be solved, as this is the type of problem that the FE method is designed to solve. The major challenge in implementing FE methods on the Connection Machine is to write efficient algorithms when the nodes are not regularly connected.

THE FINITE ELEMENT METHOD

Since the FE method has not been discussed much in SEP reports, I will now attempt to give a general introduction to the method. If you are a FE expert the generalizations and omissions in this description may offend you so you should skip to the next section.

The FE method for solving a partial differential equation starts by casting the P.D.E. as a variational problem. The differential problem

$$\begin{aligned} F(u) &= f \quad \text{on} \quad \Omega \\ u &= g \quad \text{on} \quad \partial\Omega_1 \\ \frac{\partial u}{\partial n} &= h \quad \text{on} \quad \partial\Omega_2 \end{aligned}$$

may be expressed as the variational problem, find $u \in V$ such that,

$$\begin{aligned} a(u,v) &= (f,v) + <h,v> \qquad \forall v \in V \\ (f,v) &= \int_\Omega f\, v\, dV, \qquad <h,v> = \int_{\partial\Omega_2} h\, v\, dS \end{aligned}$$

where the space V and the form of the function $a(u,v)$ depend on the differential operator F and the boundary conditions.

This is a variational problem in an infinite space, V. The approximate solution is obtained by restricting the search to a finite space of dimension k,V_k. The problem then becomes. Find u_k in V_k such that

$$a(u_k, v_k) = (f, v_k) + <h, v_k> \qquad \forall v_k \in V_k$$

$$(f, v_k) = \int_\Omega f\, v_k\, dV, \qquad < h, v_k >= \int_{\partial\Omega_2} g\, v_k\, dS$$

The accuracy of the solutions depends on the distance of the true solution u from the solution in the finite space u_k.

If we construct a set of basis functions, $\phi_i, i = 1, k$ that spans the space V_k, the problem can be rewritten as find $u_k \in V_k$ such that,

$$a(u_k, \phi_i) = (f, \phi_i) + < h, \phi_i > \qquad i = 1, \cdots, k$$

If the function u_k is written as a linear combination of the basis functions $u_k = x_j \phi_j$, we have,

$$a(\phi_j, \phi_i) x_j = (f, \phi_i) + < h, \phi_i > \qquad i = 1, \cdots, k$$

This is a linear problem for the scalars x_j of the form $Ax = b$ with,

$$A_{ij} = a(\phi_j, \phi_i), \quad b_i = (f, \phi_i) + < h, \phi_i >$$

This system of equations is know as the Galerkin system. If the basis functions are chosen to be orthogonal in the metric defined by the function $a(\phi_j, \phi_i)$, then the matrix A is diagonal. Unfortunately, for irregular problems, such functions are difficult to construct. In the FE method the functions are chosen to be functions with local support so that most elements of the matrix A are zero. The non-zero elements of A correspond to basis functions whose regions of support overlap. The basis functions are defined in terms of nodes of the elements.

The FE method has six basic stages:

1. Construct a decomposition of the domain Ω into elements.
2. Choose suitable basis functions. The basis functions are parameterized in terms of values at nodes in the elements. The nodes may be internal, on the edges, or at the vertices of an element. When using linear basis functions, as I do, the nodes are usually at the vertices of the elements.
3. Calculation of the Galerkin system.
4. Imposition of boundary conditions.
5. Solution of the Galerkin system.

A model problem

The model problem I coded on the CM was a heat flow problem.

$$-\left(\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} \right) = 4800(x - x^2) \quad \text{in} \quad \Omega$$

$$\phi = 0 \quad \text{on} \quad \partial\Omega_g$$
$$\frac{\partial \phi}{\partial n} = 0 \quad \text{on} \quad \partial\Omega_s$$

The first boundary condition is known as the essential boundary condition and has to be explicitly applied in the algorithm. The second boundary conditions is know as the natural boundary condition and it is the boundary condition implicit in the FE method. This problem was assigned as part of a computer science course "Finite Element Methods for Elliptic PDEs Using Parallel Computers." The problem was solved for a rectangular domain $(x, y) \in \Omega; 0 \leq x \leq 0.5, 0 \leq y \leq 1$ discretized into $N \times 2N$ square shaped elements. The domain and the boundary segments are shown in figure **??**.

A suitable form for the elements of the Galerkin operator is

$$a(\phi_j, \phi_i) = \int_\Omega \frac{\partial \phi_j}{\partial x}\frac{\partial \phi_i}{\partial x} + \frac{\partial \phi_j}{\partial y}\frac{\partial \phi_i}{\partial y} dV$$

The Galerkin operator

The basis functions I use are bilinear functions. The nodal basis function for the i^{th} node N_i has the value 1 at node i, zero at all other nodes, and is a bilinear function in between. Figure **??** shows the basis function for one node. The element $A_{i,j}$ of the Galerkin operator is the integral of the dot product of the gradient of the basis functions for the i^{th} and j^{th} nodes. The basis function for the i^{th} node is only non-zero in the four elements that have node i as a vertex. Thus, the only non-zero elements of the operator are $A_{i,j}$ where nodes i and j are vertices of a common element.

Almost all FE codes try to avoid constructing the full Galerkin operator because it is a very sparse matrix. Instead they construct local operators for each element. In the model problem the element operator is a 4×4 matrix, where the elements are the evaluation of the integral $a(N_i, N_j)$ over the element. Here, i and j run from 1 to 4, the local node numbering for the element. The global operator is formed as a sum of the element matrices. The application of the operator can be performed as a three stage operation,

1. spread the nodal values to form a 4-vector for each element,
2. parallel application of the element matrices,
3. gather the RHS 4-vector values and sum into the output vector.

This is the standard way that unassembled Galerkin operators are applied on vector computers (Ferencz, 1989). These calculations are performed for each *element* so the vector length is the total number of elements.

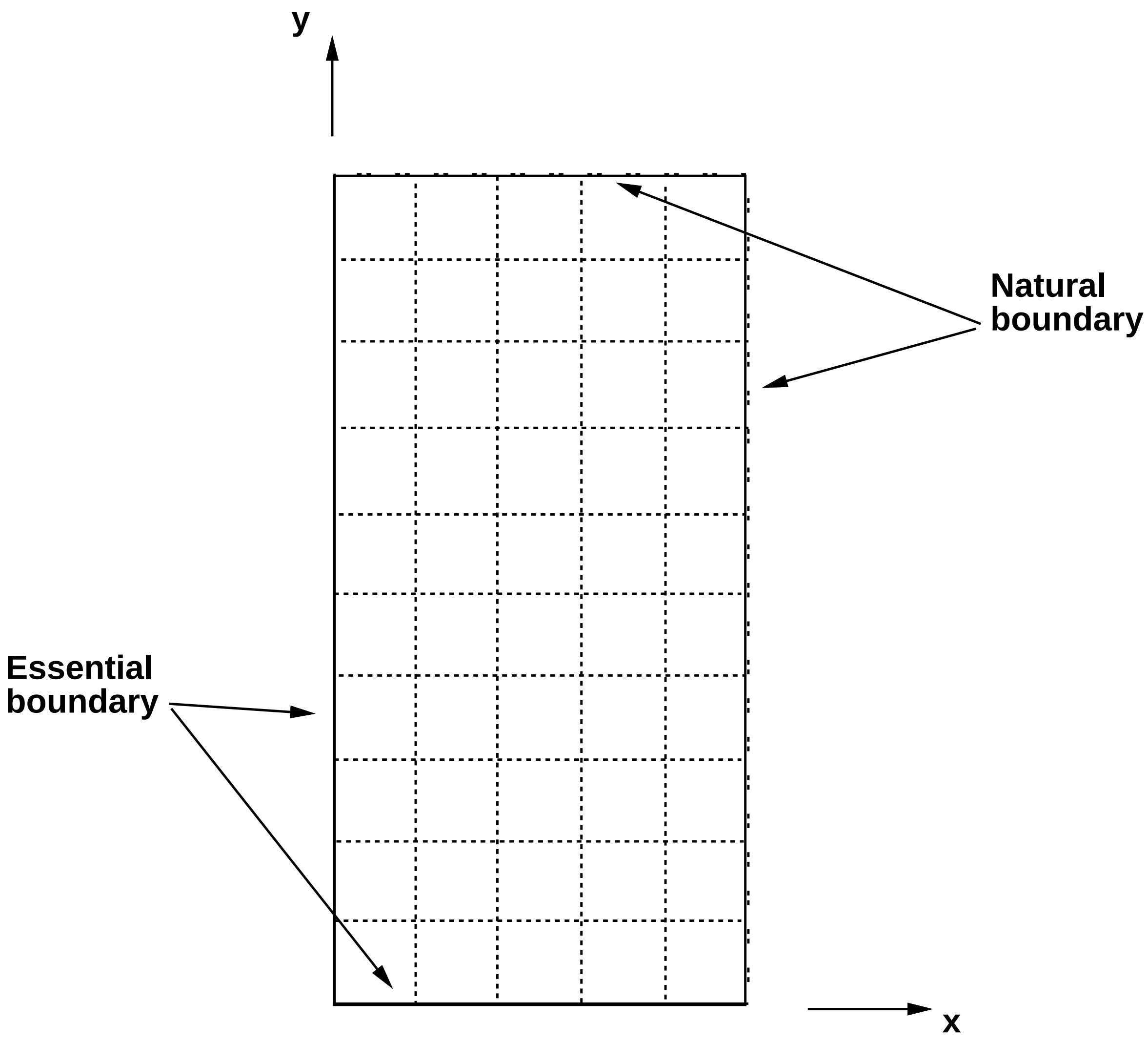

Figure 1: The model domain. Essential boundary conditions are applied on the left and bottom edges; natural boundary conditions apply on the top and right edges. $\boxed{\text{dave3-domain}}$ [NR]

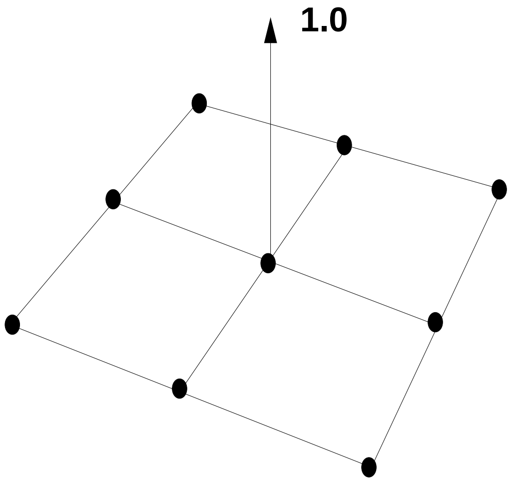

Figure 2: The basis function for a node is one at the node and only non-zero in the four elements connected to the node. dave3-nodaldiag [NR]

Use of the conjugate gradient method

The system of linear equations produced by the FE method can be solved either by direct or iterative methods. The aim in both cases is to avoid explicitly constructing the full Galerkin matrix. For machines with vector and/or low level parallel architectures, direct methods can be useful. The most popular are frontal or multi-frontal methods (Duff et al., 1986). These direct methods do not map well onto a massively parallel architecture. In contrast, iterative methods can be implemented efficiently on a massively parallel architecture. The most popular iterative method is the conjugate gradient (CG) algorithm. The Galerkin matrix is a real, symmetric, positive definite operator. For this type of operator the CG method has guaranteed convergence. The CG algorithm does not require construction of the global matrix. The main operations in the CG algorithm are calculation of the result of applying the operator to a vector and calculation of the dot product of two vectors.

The standard CG method to calculate a sequence of iterates $x^0, x^1, \cdots$ with $Ax^i \rightarrow b$ as $i \rightarrow \infty$ is given by setting $p^0 = b - Ax^0 \equiv r^0$, and then for $i = 0, 1, 2, \cdots$

$$
\begin{aligned}
\alpha_k &= (r^k, r^k)/(p^k, Ap^k) \\
x^{k+1} &= x^k + \alpha_k p^k \\
r^{k+1} &= r^k - \alpha_k A p^k \\
\beta_k &= (r^{k+1}, r^{k+1})/(r^k, r^k) \\
p^{k+1} &= r^{k+1} + \beta_k p^k
\end{aligned}
\tag{1}
$$

$(.,.)$ is the L^2 inner product.

This method is clearly suitable for solution of the unassembled Galerkin system. The calculation of Ap_k, which is performed once per iteration, can be done using the method outlined in the previous section. The only global operations in the algorithm are the calculation of the two inner products,(p^k, Ap^k) and (r^{k+1}, r^{k+1}).

IMPLEMENTATION ON THE CONNECTION MACHINE

The two main operations to be considered are parallel construction of the unassembled Galerkin operator and parallel application of the operator. The aim of the algorithm is to spread the calculation evenly over the available processors while minimizing the amount of processor-processor communication.

Because I wish to minimize the communication overhead, I chose a slightly different method of constructing the unassembled operator. Instead of using the element as the basic unit, I used the node. This means that my unassembled system will consist of rows of the global operator rather than submatrices. The reason for this choice should become clear as I outline my method.

Parallel computation of the Galerkin operator

The elements of the i^{th} row of the Galerkin operator are given by evaluation of the function $a(N_j, N_i)$ for all $N - j$. These elements are only non-zero for those basis functions whose support overlaps that of the function N_i. The only nodal basis functions that need to be considered are those that share a common element with the i^{th} node. If the local numbering scheme shown in Figure **??** is used, then we can clearly see that there are only nine non-zero elements in each row. Each node in the mesh has a nine-vector of operator coefficients associated with it. These vectors can be calculated in parallel for each node in the mesh,

$$\begin{pmatrix} A_1 \\ A_2 \\ A_3 \\ A_4 \\ A_5 \\ A_6 \\ A_7 \\ A_8 \\ A_9 \end{pmatrix} = \begin{pmatrix} a(N_1, N_5)_{\mathit{1}} \\ a(N_2, N_5)_{\mathit{1,2}} \\ a(N_3, N_5)_{\mathit{2}} \\ a(N_4, N_5)_{\mathit{1,3}} \\ a(N_5, N_5)_{\mathit{1,2,3,4}} \\ a(N_6, N_5)_{\mathit{2,4}} \\ a(N_7, N_5)_{\mathit{3}} \\ a(N_8, N_5)_{\mathit{3,4}} \\ a(N_9, N_5)_{\mathit{4}} \end{pmatrix}$$

where $a(N_i, N_j)_{k,l}$ represents evaluation of the integral for nodal functions i and j with the integral evaluated over the elements k and l.

To calculate the operator elements for the nodes, I must know whether the neighboring nodes exist, to account for boundary nodes, and the (x, y) coordinates of the neighboring nodes. At this stage the connectivity of the nodes enters the problem. The coordinate information is stored as an array on the CM. Each processor (or virtual processor) is responsible for calculating one row of the Galerkin operator and one

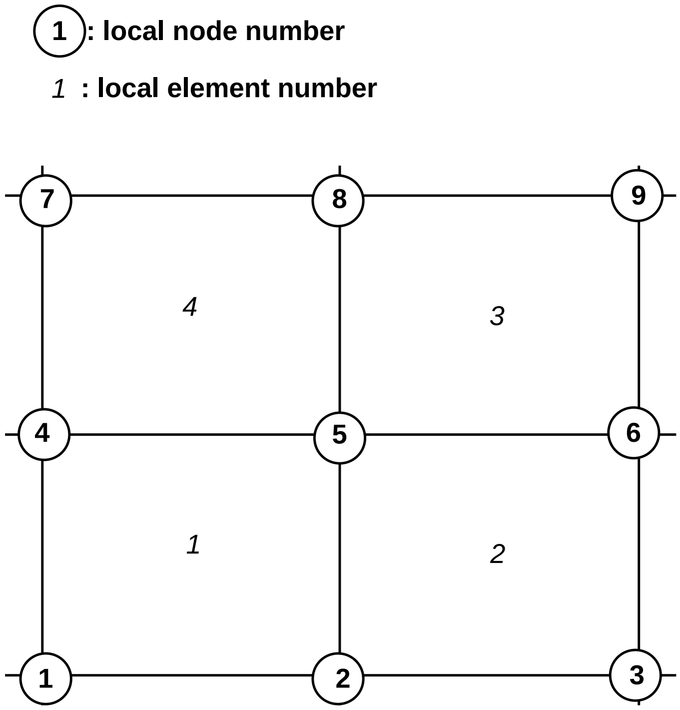

Figure 3: The local numbering scheme for nodes involved in calculation of one row of the operator. dave3-localel [NR]

element of the RHS of the equation. To do this, it must gather information about the coordinates of its eight neighboring nodes.

If the nodes are connected in a regular 2-D mesh, the coordinates can be stored in a 2-D array. The array will be mapped onto the processors of the CM with neighboring elements in the array stored in neighboring processors (or the same processor if there are more array elements than processors). The transfer of coordinates from each node to its neighbors is a simple "shift" operation. Eight shifts, each by one element, are sufficient to gather all the information needed to construct the row of the operator for each node.

If the nodes are not connected in a regular 2-D mesh, the transfer of coordinate information is more complicated. To gather the coordinate information, each processor must perform an indexed lookup of the coordinates of its neighbors. This type of data transfer is much less efficient than the simple shift operation. The CM is much better at transferring data when a regular transfer pattern is known at compile time than when it has to perform a global "send" operation at run time.

Once the coordinate information has been gathered, the calculation of the rows of the global operator is a fully parallel operation. It is not too important, since the data transfer occurs only once in this process. In my implementation of the method on the CM, the cost of calculating the operator was small compared with the cost of the solution phase. If a higher order FE method were used the cost of assembling the operator might be greater, however, a higher order method produces a more complex operator so the solution phase would also be more expensive.

After the operator has been calculated the boundary conditions are applied. For essential boundary conditions, this merely involves zeroing elements of the operator. The boundary information is distributed across all the nodes so the operation can be performed in parallel. The natural boundary conditions require no modification of the operator because it is implicit in the FE method. More complex boundary conditions usually involve modification of the operator and/or the RHS. These will also be parallel computations.

Parallel solution by the conjugate gradient method

The coordinate and boundary information is no longer needed, once the operator has been constructed and boundary conditions have been applied. All that remains is to solve the system of linear equations. This can be efficiently done by the CG method. The CM code for the iterations of the CG algorithm is almost a one-to-one match with the algorithm outline given in Equation 1. This is one of the more attractive features of CM-Fortran, there is none of the extra clutter of do loops to obscure the meaning of the code. Figure **??** shows the conjugate gradient loop in its entirety. The most important point about this code is that there is only one distribution step per iteration. This is the minimum possible amount of communication for the CG algorithm. The only other interprocessor communication is in the "SUM" operation which is used to calculate the inner products.

The layout of the arrays and the coding of the distribution routine depend on the connectivity of the nodes. In the model problem the nodes can be mapped to a 2-D array and the distribution can be performed as a series of shift operations. Figure **??** shows the CM code to define the array layout and Figure **??** the routine "DISTRIBUTE".

Table 7 shows the timings for the model problem. The first set of figures is for the code given in Figure **??**. The second set of figures is for a version of the code in which the distribution and application steps were replaced by a call to a routine compiled with the stencil compiler. This is a special purpose compiler for compiling algorithms that can be coded as a stencil operation. Biondo Biondi used a pre-release version of this compiler to create the routine that I used.

There are several points to note:

- The cost of the construction step is negligible compared to the solution step.
- It is very important to make the array dimensions a power of two when using the "CSHIFT" operator. The first case (129×257) is a worst case example with each dimension being a power of two plus one.

```
      DO WHILE( NORMRATIO > ERRLIM )

C        Distribute the step information to all the nodes.
         CALL DISTRIBUTE( STEP, LOCSTEP, NNODEX, NNODEY )

C        apply the operator
         CONJSTEP = 0.
         DO I =1,9
            CONJSTEP = CONJSTEP + OPERATOR(I,:,:) * LOCSTEP(I,:,:)
         END DO

C        calculate alpha
         ALPHA = PREVNORM / SUM( STEP * CONJSTEP )

C        update soln
         SOLN = SOLN + ALPHA * STEP

C        update residuals
         RESID = RESID - ALPHA * CONJSTEP

C        calculate norm of residuals and beta
         RESIDNORM = SUM( RESID*RESID )
         BETA = RESIDNORM/PREVNORM

C        update the step
         STEP = RESID + BETA * STEP

C        calculate relative error in residuals.
         NORMRATIO = RESIDNORM/RHSNORM

         PREVNORM = RESIDNORM

      END DO
```

Figure 4: CM-Fortran code for the CG loop

```
      REAL OPERATOR(9,NNODEX,NNODEY)
      REAL LOCSTEP(9,NNODEX,NNODEY)
CMF$LAYOUT OPERATOR(:SERIAL,:NEWS,:NEWS)
CMF$LAYOUT LOCSTEP(:SERIAL,:NEWS,:NEWS)

      REAL SOLN(NNODEX,NNODEY)
      REAL STEP(NNODEX,NNODEY)
      REAL RESID(NNODEX,NNODEY)
      REAL CONJSTEP(NNODEX,NNODEY)
CMF$LAYOUT SOLN(:NEWS,:NEWS)
CMF$LAYOUT RESID(:NEWS,:NEWS)
CMF$LAYOUT STEP(:NEWS,:NEWS)
CMF$LAYOUT CONJSTEP(:NEWS,:NEWS)
```

Figure 5: Definition of the arrays for a regular 2-D problem. The operator and the work vector for step information are serial vectors of nine points on each processor. All the other arrays have one element per processor.

```
      SUBROUTINE DISTRIBUTE( INPUT, OUTPUT, NNODEX, NNODEY )

C     Gather information from all connected nodes.
C     This can be done with shift operations.
C
      INTEGER NNODEX, NNODEY

      REAL OUTPUT(9,NNODEX,NNODEY)
CMF$LAYOUT OUTPUT(:SERIAL,:NEWS,:NEWS)

      REAL INPUT(NNODEX,NNODEY)
CMF$LAYOUT INPUT(:NEWS,:NEWS)

      OUTPUT(5,:,:) = INPUT

      OUTPUT(4,:,:) = CSHIFT( OUTPUT(5,:,:), DIM=1, SHIFT=-1 )
      OUTPUT(1,:,:) = CSHIFT( OUTPUT(4,:,:), DIM=2, SHIFT=-1 )
      OUTPUT(7,:,:) = CSHIFT( OUTPUT(4,:,:), DIM=2, SHIFT=1 )

      OUTPUT(6,:,:) = CSHIFT( OUTPUT(5,:,:), DIM=1, SHIFT=1 )
      OUTPUT(3,:,:) = CSHIFT( OUTPUT(6,:,:), DIM=2, SHIFT=-1 )
      OUTPUT(9,:,:) = CSHIFT( OUTPUT(6,:,:), DIM=2, SHIFT=1 )

      OUTPUT(2,:,:) = CSHIFT( OUTPUT(5,:,:), DIM=2, SHIFT=-1 )
      OUTPUT(8,:,:) = CSHIFT( OUTPUT(5,:,:), DIM=2, SHIFT=1 )

      RETURN
      END
```

Figure 6: Routine to distribute step information to all processors for a 2-D mesh.

	Construction	Solver		
Problem size	Time/s	Time/s	Iterations	MFlops
129×257	.07	35.6	393	9.5
128×256	.07	6.9	392	95
256×512	.22	27.7	792	127
Using Stencil compiler				
64×128	.03	.63	200	88
128×256	.07	3.6	390	183
256×512	.22	12.7	792	276

Figure 7: Timing figures for the model problem.

- As the problem size increases, the number of iterations required to converge also increases. Theory predicts that the number of iterations is $O(\text{Number of nodes})$. The observations are a good match to the theory.
- The stencil compiler is much more efficient than the normal Fortran compiler for stencil operations. The routine produced by the stencil compiler was separately tested and found to run at 500MFlops. I believe that the part of the algorithm that controls the speed of the solution is now the "SUM" operation.

Discussion

The speeds obtained here only apply to regularly connected meshes, I would expect irregular meshes to run at a slower speed because of the higher cost of the distribution step. There are two alternatives that could be considered for irregularly connected meshes. Use a 1-D array of nodes and distribute data using a global indexing scheme. It was suggested by Mathur (1990) that if this approach is used the indexing arrays should be randomly constructed so that global communication conflicts are reduced. Another approach would be to map the irregular mesh onto a regular 2-D mesh with some elements missing. There would be a computational overhead proportional to the number of missing nodes, but the overall cost might be reduced because of the speed of the regular communications. This idea is illustrated in Figure **??**.

If, as in the model problem, the grid and the domain are both rectangular, the operator produced by the FE method is similar to a finite difference operator. For this type of very regular problem it is probably wiser to use a finite difference method. However, the operator calculation code that I implemented does not assume this rectangular layout. The elements must be connected in a regular mesh, but they may have arbitrary shapes. This case is related to the conformally mapped finite difference scheme described by Fornberg (1989). Each method has some advantages. More time is spent in constructing the operator in the FE code, but boundary conditions are generally much easier to implement.

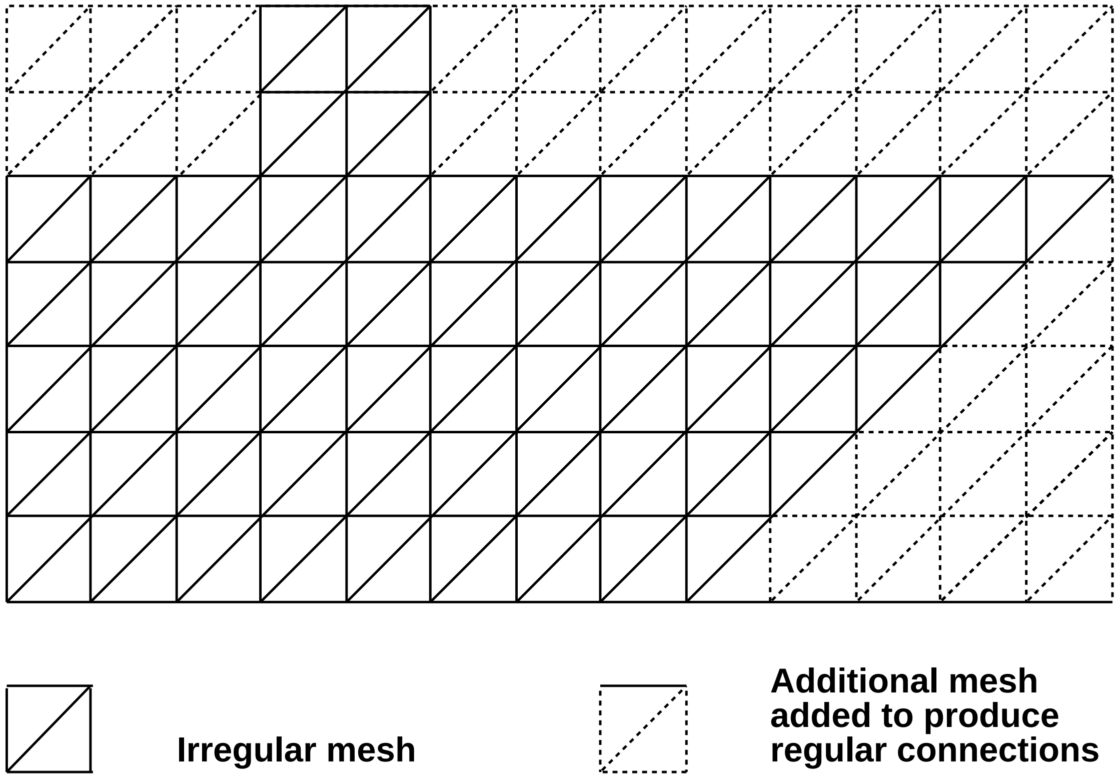

Figure 8: Construction of an enlarged computational mesh to ensure regular connections between nodes. dave3-irreg [NR]

One common procedure for speeding up the CG method is to use a preconditioning operator. A diagonal preconditioner is obviously very easy to apply. If better preconditioners are desired the method depends on the layout of the data. If the global Galerkin operator is constructed, an incomplete Choleski preconditioner can be used. When the unassembled system consists of element matrices the preconditioning is usually done with element-by-element operators (Lee and Wathen, 1989). This is not simple to do when using the row-wise decomposition that I propose, but I believe that it should be possible to construct an efficient preconditioning operator with the row-wise data layout.

CONCLUSIONS

I have written a finite element code that runs efficiently on the CM. A key factor in obtaining this efficiency is the use of a regularly connected mesh. This allows the use of fast "CSHIFT" operations for processor-processor communication rather than the slower global routing. I solved the Galerkin system of equations by the conjugate gradient method. This allowed me to avoid constructing the global operator. I chose to use a nodal decomposition of the operator rather than the usual element matrix decomposition. I need only distribute data once per iteration of the CG algorithm with this decomposition. If the stencil compiler is used, the speed of the algorithm is limited by the speed of the SUM operation. On a 4096 processor machine the code ran at 275MFlops. If a 64K processor machine were to be used I would expect to achieve a speed of 4.4GFlops.

ACKNOWLEDGMENTS

I would like to thank David Silvester for teaching an interesting course and introducing me to the finite element method.

REFERENCES

Duff, I., Erisman, A., and Reid, J., 1986, Direct Methods for Sparse Matrices: Oxford University Press.

Fornberg, B., 1989, Accurate representations of interfaces in elastic wave calculations: Geophysics, **53**, 625–638.

Golub, G., and Van Loan, C., 1989, Matrix Computations (*second edition*): John Hopkins University Press.

Johnson, C., 1989, Numerical solution of partial differential equations by the finite element method: Cambridge University Press.

Lee, H-C., and Wathen, A.J., 1989, On element-by-element preconditioning for general elliptic problems: Numerical Analysis Report **NA-89-10**, Computer Science Dept., Stanford University.

Mathur, K., 1990, On the use of randomized address maps in unstructured three dimensional finite element simulations: Thinking Machines Corporation, technical report series, **CS90-4**.