PARALLEL FINITE ELEMENT ANALYSIS OF BIOMECHANICAL STRUCTURES ON THE NCUBE 6400

S. Chinchalkar and T. F. Coleman

July 1991

1 Introduction

This paper presents parallel 3-D finite element analysis for distributed memory multiprocessors. Traditionally, finite element analysis has been performed on sequential computers. Current research in high performance finite element analysis shows considerable promise for fast, efficient implementation on MIMD and SIMD computers [Farhat 87a], [Johnsson 90].

The finite element system of equations can be solved by a direct method, or by an iterative method [Carter 89]. Although iterative methods are attractive when a good initial estimate of the solution is available, direct methods are generally preferred because once the coefficient matrix (global stiffness matrix) is factored, solution to multiple load cases and sensitivity analysis becomes considerably simpler.

Several researchers have used the substructuring method for parallel finite element analysis on hypercubes [Adams 84], [Farhat 87b], [Carey 86]. It is described only briefly here; refer to [Adams 84] for details. In this method, the structural domain is decomposed into several subdomains (substructures), one for each processor. Nodes that are common to more than one subdomain form the 'interface'. The resulting global stiffness matrix has an 'arrowhead' pattern, with diagonal blocks corresponding to each substructure and the interface, and off-diagonal blocks coupling each substructure with the interface. Each processor assembles the element stiffness matrices for its substructure. Each processor then factorizes independently, one diagonal block and updates the interface block. All processors, in parallel, solve a system of equations corresponding to the interface block. Subsequently, the processors recover the displacement solution for their substructure, independently.

While the substructuring method appears to be attractive in terms of division of tasks among processors, its applicability for massively parallel systems (more than 1000 processors) is not obvious. Firstly, balancing the computational load among different processors is not simple; the computational load is a function of the number of degrees of freedom internal to each substructure and the bandwidth of each substructure. Therefore a simple algorithm that assigns an equal number of elements to each processor is not guaranteed to partition the computational load equally among the processors. Another concern is that the size of the interface becomes very large when the number of processors is large. For example, in problem 7 (see section 5), 512 processors are required to compute the solution of a 2562 element model. For this problem, each processor has only 5 elements associated with it and a very large fraction of the nodes would be a part of the interface. Thus, most of the computational effort would be spent in factoring the interface block. In this case, the method is not as attractive as when the number of substructures is small.

This paper demonstrates the use of a standard, banded Cholesky method for solving the finite element system of equations. The uniformity of the underlying data distribution ensures high performance due to load balance. Moreover, since a distributed banded Cholesky algorithm is likely to be a part of a standard parallel numerical library, it reduces the burden on the applications programmer, making this method simpler to implement than the substructuring method. Since a parallel solver requires the rows of the coefficient matrix to be distributed in a wrap fashion, it might appear that the assembly of the element stiffness matrices would not be efficient. However, as shown in this paper, the calculation of element stiffness matrices, assembly and the calculation of Gauss-point stresses can be done efficiently in parallel without any inter-processor communication. In fact, once the nodal coordinates and element connectivity is made available to all processors, message passing is required only during the factorization and solution stages.

The next few sections describe how parallelism was exploited during the assembly, solution and stress recovery stages of the finite element analysis. The parallel program developed was tested on large 3-D finite element problems arising from biomechanical structural systems, on an Ncube 6400. High performance Basic Linear Algebra Subprograms (BLAS) were used to improve the execution speed.

2 The Ncube 6400

The Ncube 6400 (also known as the Ncube 2) is a distributed memory MIMD machine with up to 8192 node processors and 1 host processor. Each node processor is a 64-bit chip supporting up to 64 Mbytes of memory [Dunigan 91]. The programs written as a part of this work were tested on a 1024 processor machine with 4 Mbytes of memory per processor. High performance BLAS run at an asymptotic speed of about 1.8-2.0 Mflops (depending on the subprogram) per processor. Data is sent from one processor to another by explicit message-passing. Nearest neighbor communication takes approximately $200 + 0.6N$ microseconds for a message of N bytes [Dunigan 91].

The Portable Instrumented Communication Library (PICL) [Geist 91] was used to write portable host and node programs. The use of this software library relieves the programmer of the additional burden of learning a new parallel programming environment. The overhead in using this library as compared to the native Ncube communication utilities is typically small [Geist 90].

3 Finite Element Analysis

Linear, elastic, static 3-D structural finite element analysis problems can be modeled by the following differential equations:
Equilibrium:

$$\sigma_{ij,i} + f_j = 0 \qquad \text{on} \quad \Omega \tag{1}$$

Compatibility:

$$\epsilon_{ij,kl} + \epsilon_{kl,ij} = \epsilon_{ik,jl} + \epsilon_{jl,ik} \qquad \text{on} \quad \Omega \tag{2}$$

Constitutive Law:

$$\sigma_{ij} = E_{ijkl}\epsilon_{kl} \qquad \text{on} \quad \Omega \tag{3}$$

Boundary Conditions:

$$S = \tilde{S} \qquad \text{on} \quad \Gamma_s$$
$$u = \tilde{u} \qquad \text{on} \quad \Gamma_u \tag{4}$$

where σ is the stress tensor, ϵ is the strain tensor, f is the body force, E is the material property tensor, S is the surface traction and u is the displacement. Γ_s is the boundary on which surface tractions are specified, Γ_u is the boundary on which the displacements are specified and Ω is the domain on which the solution is desired.

These equations are solved numerically by discretizing the domain into finite elements and modeling the displacement field within each finite element by simple low order polynomials (shape functions) [Zienkiewicz 77]. An element stiffness matrix is formed as follows:

$$K_e = \int_{\Omega_e} B^T D B d\Omega_e \tag{5}$$

where B is a matrix of shape function derivatives and D is the material property matrix. The integral is evaluated by Gaussian quadrature as follows

$$K_e = \sum_{i=1}^{ngpt} B^T D B w_i |J| \tag{6}$$

where $ngpt$ is the number of Gauss-points, w_i is the Gauss-point weight and $|J|$ is the determinant of the Jacobian. The element stiffness matrix is small; 60×60 for a 20-noded brick element (B20) and 45×45 for a 15-noded wedge element (W15). The global stiffness matrix is obtained by assembling the element stiffness matrices. The nodal displacements are obtained by solving the following large sparse system of equations

$$Ku = F \tag{7}$$

where K is a positive definite, symmetric, banded global stiffness matrix, u is the vector of nodal displacements and F is the vector of externally applied nodal forces. The stresses at 'Gauss points' within each element are computed using the following relationship

$$\sigma = D B u_e \tag{8}$$

where u_e is the vector of nodal displacements for element e.

Most of the computational effort is spent in solving the finite element system of equations (7); calculation of element stiffness matrices and recovery of Gauss point stresses is low order (linear) work.

The following notation is used in the remainder of this paper:

n_e = number of finite elements

n_n = number of nodes

n_b = number of nodal supports

n_d = $3n_n - n_b$ = number of degrees of freedom

b = semi bandwidth of the global stiffness matrix

p = number of processors

The number of floating point operations needed for different portions of the analysis is given by:

$$\begin{aligned}
\text{assembly} &= 2.2 \times 10^5 n_e \\
\text{factorization} &= n_d b^2 \\
\text{solution} &= 4 n_d b \\
\text{stress calc.} &= 1.2 \times 10^4 n_e
\end{aligned}$$

The estimates for assembly and stress calculation is based on 20-noded brick elements only. If 15-noded wedge elements are present, the flop count at every Gauss-point is slightly smaller. The total flops required for assembly and stress calculation depends on the number of Gauss-points in the element. Note that the amount of CPU time spent on these operations cannot be estimated by the number of floating point operations alone; the length of the vectors being operated upon and integer arithmetic also affects the required CPU time. In case of parallel analysis, communication overhead, redundant computation and processor idle time are quite significant.

4 Parallel Finite Element Analysis

As mentioned earlier, in this work, a parallel banded Cholesky factorization is used to solve the finite element system of equations. A problem dependent data file containing nodal coordinates, element connectivity, nodal loads and boundary conditions is read in by processor 0 of the hypercube. Processor 0 then broadcasts this data to all other processors. Alternatively, the host can read the data file and broadcast the data to all processors of the cube. However, on the Ncube, the host and node CPU's and compilers are different and use different byte ordering and sizes of C structures. This creates compatibility problems when messages are sent from the host to the node and vice-a-versa. Hence, for convenience, the host did not participate in the analysis; node 0 was responsible for the I/O and distribution of data. Node I/O on disks attached to the host is particularly slow and was excluded while estimating the performance of the program. The next few sections describe in greater detail different stages of the analysis.

4.1 Initializations

After all the nodes of the hypercube receive the nodal coordinates, element connectivity and other input data, each processor computes the bandwidth of the global stiffness matrix and allocates space for the portion of the global stiffness matrix that belongs to it. Each processor is also responsible for setting up the load vector (right hand side). The shape function derivatives with respect to the natural coordinates are functions of the natural coordinates only (and not the global (x,y,z) coordinates). Therefore they can be calculated beforehand, for each Gauss-point, independent of the element number. Each processor computes all shape function derivatives for each element type (in this case B20 and W15 elements) and stores them for later use in generation of the B matrix needed in element stiffness matrix and stress calculations.

4.2 Assembly

This section describes how the element stiffness matrices are computed and assembled efficiently in parallel. This has to be done keeping in mind the overall efficiency issues and not just those during the assembly stage. Since the cost of factoring the global stiffness matrix exceeds all other costs, it is important that the data be distributed in a manner that suits best the factorization algorithm. For efficient parallel factorization, rows of the global stiffness matrix are distributed by a wrap mapping (row i belongs to processor $\mathrm{P}(i) = i \bmod p$). This ensures that all processors have some rows to update, until the final stages of factorization. One could also use a block mapping, where a block of n/p consecutive rows is assigned to each processor. In this case, however, a processor is idle as soon as all updates to its block are completed. This leaves several processors idle during the later stages of factorization.

Since each processor needs a few rows of the global stiffness matrix for the next stage, ideally, each processor should compute only those entries in the element stiffness matrices that contribute to these rows. It is shown below that this is indeed possible with very little redundant computation.

Each degree of freedom (dof) is given a unique number (from 1 to n_d) in the final system of equations, stored in the array `dofnum()`. Nodal supports are excluded from this set. The dof numbers for element i are denoted by N_{ij}, where the subscript j goes from 1 to 60 for B20 elements and from 1 to 45 for W15 elements. The entry (j, k) in the element stiffness matrix for element i is assembled in location (`dofnum`(N_{ij}),`dofnum`(N_{ik})) of the global stiffness matrix. Unfortunately, the node numbering *within* an element is rarely in the same order as the global node numbering. For example, a W15 element, could have its nodes numbered as:

`local number`	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`global number`	42	22	15	99	82	77	37	32	33	43	23	16	94	89	90

This implies that the entry (`dofnum`(N_{ij}),`dofnum`(N_{ik})) could go into either the upper or the lower triangular portion of the global stiffness matrix depending on whether `dofnum`(N_{ij}) is greater or less than `dofnum`(N_{ik}). Since only the upper triangular portion of the global stiffness matrix is assembled, a row of the global stiffness matrix on a processor could receive a few contributions from every row of an element stiffness matrix. This would require a processor to compute the entire element stiffness matrix, or selectively compute a few entries in it. This can be avoided by applying a permutation P to the dof numbers so as to arrange them in ascending order. The element stiffness matrix can now be computed as:

$$K_e = \sum_{i=1}^{ngpt} P^T B^T D B P w_i |J| \tag{9}$$

$$= \sum_{i=1}^{ngpt} (BP)^T D (BP) w_i |J| \tag{10}$$

This ensures that the upper triangular portion of K_e is assembled in the upper triangular portion of the global stiffness matrix, K, and that during assembly, all entries in a row of K_e are added to only one row of K. Each processor thus needs to selectively compute a few rows of each K_e.

While computing the product $(BP)^T D(BP)$, the sparsity of B and P can be exploited easily. For an isotropic material, D is given by

$$D = \frac{E}{(1+\nu)(1-2\nu)} \begin{bmatrix} 1-\nu & \nu & \nu & 0 & 0 & 0 \\ \nu & 1-\nu & \nu & 0 & 0 & 0 \\ \nu & 1-\nu & \nu & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1-2\nu}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1-2\nu}{2} & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1-2\nu}{2} \end{bmatrix}_{6\times 6} \tag{11}$$

where E is the Young's modulus and ν is the Poisson's ratio. For a B20 element, the matrix of shape function derivatives, B is given by

$$B = \begin{bmatrix} \frac{\partial N_1}{\partial x} & 0 & 0 & \dots & \dots & \dots & \frac{\partial N_{20}}{\partial x} & 0 & 0 \\ 0 & \frac{\partial N_1}{\partial y} & 0 & \dots & \dots & \dots & 0 & \frac{\partial N_{20}}{\partial y} & 0 \\ 0 & 0 & \frac{\partial N_1}{\partial z} & \dots & \dots & \dots & 0 & 0 & \frac{\partial N_{20}}{\partial z} \\ \frac{\partial N_1}{\partial y} & 0 & \frac{\partial N_1}{\partial z} & \dots & \dots & \dots & \frac{\partial N_{20}}{\partial y} & 0 & \frac{\partial N_{20}}{\partial z} \\ \frac{\partial N_1}{\partial x} & \frac{\partial N_1}{\partial z} & 0 & \dots & \dots & \dots & \frac{\partial N_{20}}{\partial x} & \frac{\partial N_{20}}{\partial z} & 0 \\ 0 & \frac{\partial N_1}{\partial y} & \frac{\partial N_1}{\partial x} & \dots & \dots & \dots & 0 & \frac{\partial N_{20}}{\partial y} & \frac{\partial N_{20}}{\partial x} \end{bmatrix}_{6\times 60} \tag{12}$$

24 out of 36 entries in D and half the entries in B are zero. The matrices $\bar{B}$, C, and T are defined as follows:

$$\begin{aligned} \bar{B} &= BP \\ C &= D\bar{B} \\ T &= \bar{B}^T C \end{aligned} \tag{13}$$

Each processor 'computes' $\bar{B}$ by storing the shape function derivatives in their permuted locations, instead of explicitly multiplying B by P. The matrix C can be computed as a series of outer products of the form

$$c_{i,1:n} = c_{i,1:n} + d_{ij}\bar{b}_{j,1:n} \tag{14}$$

Outer products corresponding to $d_{ij} = 0$ can be skipped. The matrix C is computed by all processors. Each processor needs to compute only a few rows of T. The upper triangular portion of a row of T can be updated by outer products of the form

$$t_{i,i:n} = t_{i,i:n} + \bar{b}_{ji} c_{j,1:n} \tag{15}$$

and outer products corresponding to $\bar{b}_{ji} = 0$ can again be skipped. The calculation of $T = \bar{B}^T C$ is done in a distributed manner, whereas all processors compute $\bar{B}$, P,

```
for i = 1 to n
        a_ii = sqrt(a_ii)
        for j = i + 1 to n
                a_ij = a_ij / a_ii
        end
        bcast(a_{i,i+1:n})
        for j = i + 1 to n
                if (P(j) = me) then
                        for k = j to n
                                a_jk = a_jk - a_ij a_ik
                        end
                endif
        end
end
```

Figure 1: Parallel Cholesky factorization

and C. This redundant computation (overhead) is low as compared to the cost of computing T.

When the number of processors is large, each processor has only a few rows of the global stiffness matrix. For example, in problem 7 shown in the next section, there are 2562 B20 elements and this problem is solved using 512 processors. When $n_e = 2562$ and $p = 512$, there are $60 \times 2562 = 153720$ rows of all the element stiffness matrices put together (some of which overlap) shared by 512 processors. Ignoring boundary conditions, this leaves, on the average, only about 300 rows per processor, much less than the number of elements. In fact, for problem 7, the global stiffness matrix with each processor receives contributions from no more than 334 elements; that is the global stiffness matrix with an individual processor does not receive any contributions from the element stiffness matrices of at least 2228 elements. It would be wasteful then, to perform all the preliminary calculations (such as computing $D\bar{B}$) for these elements, and skip the product $\bar{B}^T D\bar{B}$. Therefore, each processor determines beforehand, for each element (based on `dofnum`(i), for each dof i), whether it needs to compute any rows of that element stiffness matrix at all. If no rows need to be computed, further operations on that element are skipped. This results in substantial savings in computational effort, especially for large problems.

4.3 Parallel Cholesky factorization

The global stiffness matrix, which is distributed by rows now needs to be factored. The Cholesky factor is computed in parallel by the node processors using the algorithm given in [Geist 86]. The algorithm for the dense case is presented in Figure 1. It was modified for the banded case taking into account the banded storage scheme. In this algorithm, P(i) is the processor that possesses row i of the global stiffness

```
for i = 1 to n
      s = 0
      for j = 1 to i − 1
            if (P(j) = me) then
                  s = s + l_ij y_j
            endif
      end
      t = fan-in(s)
      if (P(i) = me) then
            y_i = (b_i − t)/l_ii
      endif
end
```

Figure 2: Parallel forward elimination

matrix. The operation `bcast` broadcasts a vector to all nodes of the hypercube. A minimum depth spanning tree rooted at P(i), the initiator of the broadcast, is used to efficiently broadcast the vector. When all updates to row j are completed, the row is sent off to all other processors before proceeding with the rest of the update. This reduces the the amount of time other processors have to wait for row j of the matrix. This is called 'pipelining' [Geist 86]. Details such as 'pipelining' and subscripting for the parallel banded storage scheme are not shown in Figure 1 for simplicity.

4.4 Solution of triangular system of equations

Even though on a serial computer, the solution of a triangular system of equations (forward elimination and back substitution) is low order work, on a parallel computer, it is more difficult to obtain high performance due to the distributed nature of the Cholesky factor and the reduction in the amount of computation to be performed. The primary concern is to ensure that the amount of CPU time spent on the solution of triangular system of equations is a small fraction of the total computational cost. There are several methods for solving a triangular system of equations. They can be classified into 'fan-in/fan-out' methods [Romine 88], cyclic methods [Li 89] and wave-front methods [Heath 88]. For the purpose of this paper, the fan-in/fan-out methods were used because of their simplicity and reasonably efficient performance which kept the cost of solving the system of equations, a small fraction of the cost of factoring the global stiffness matrix. Cyclic methods can out-perform fan-in/fan-out methods though; refer to [Eisenstat 88] for a comparison of these methods.

Figure 2 shows the parallel algorithm for forward elimination, i.e. solving

$$Ly = b \tag{16}$$

for y [Romine 88]. Note that the matrix L is distributed by columns. The `fan-in`

```
for i = n to 1 step -1
      if (P(i) = me) then
            x_i = x_i/l_ii
      endif
      bcast(x_i)
      for k = 1 to i-1
            if (P(k) = me) then
                  x_k = x_k - l_ki * x_i
            endif
      end
end
```

Figure 3: Parallel back substitution

operation gathers contributions from all processors and stores the result in processor P(i). This algorithm was modified for the banded case. On the Ncube, a `fan-in` operation can be performed quite fast, which is crucial for the success of this algorithm. Figure 3 shows the parallel algorithm for back substitution, i.e. solving

$$L^T x = y \tag{17}$$

for x [Geist 86], when the matrix L^T is distributed by rows. It is assumed that the vector y is stored in x and is over-written by the solution. This algorithm was also modified for the banded case. It is shown in the next section that the cost of solving the system of equations was only 3-6% of the cost of factoring the coefficient matrix.

4.5 Calculation of Gauss-point stresses

Calculation of Gauss-point stresses is the most straight-forward operation and can be carried out in parallel without any inter-processor communication. Each processor is assigned a block of n_e/p elements. Since each processor knows the entire displacement solution, it can compute the stress in any element independently, using equation 8. Even though the amount of computation required differs for different element types, load balancing is hardly an issue here, because this portion of the analysis requires an insignificant fraction of the total computational cost.

5 Numerical Experiments

For the purpose of this study, several problems arising from orthopedic biomechanics were chosen. The number of finite elements in these problems ranged from 40 to 2562. Figures 4-10 show the discretized finite element models. Table 1 provides a short description along with some specifications for these models. The problems are listed in

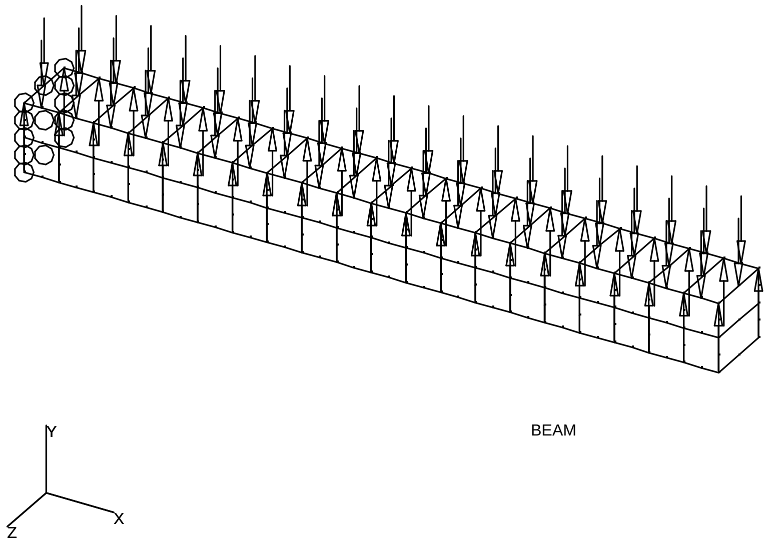

Figure 4: Problem 1: Cantilever beam subject to uniformly distributed load

increasing 'size'. 'Size' here is a measure of the number of floating point operations needed for Cholesky factorization ($n_d b^2$) and is thus a measure of the amount of CPU time required. For each of these models, a finite element analysis was performed on the smallest cube capable of solving the problem. The size of the cube was chosen based on the memory requirements for the particular problem and was restricted to be a power of 2. It should be noted that the problems being solved were not the largest problems capable of being solved on a cube of the chosen size.

Table 2 shows the amount of time taken by different stages of the finite element analysis. The Mflop rate was computed for Cholesky factorization only. It can be seen that parallel computation of element stiffness matrices and the assembly takes very little time. Stress recovery and other overhead such as calculation of bandwidth and other initializations also consumes negligible computational effort. On the other hand, solution to system of equations via forward elimination and back substitution takes approximately 3-6 % of the time taken for factorization. On a serial machine, since Cholesky factorization takes $n_d b^2$ flops and forward elimination and back substitution together take $4 n_d b$ flops, the ratio of factorization time to solution time is approximately $b/4$. However, on a parallel machine, this ratio is much smaller. There are two reasons for this. Firstly, since the amount of computation to be performed is significantly smaller than that required during factorization, the communication overhead is significantly larger. As mentioned earlier, on the Ncube, nearest neighbor communication requires approximately $200 + 0.6N$ microseconds for a message of N bytes. Therefore, there is a large penalty for sending short messages (200 microsec-

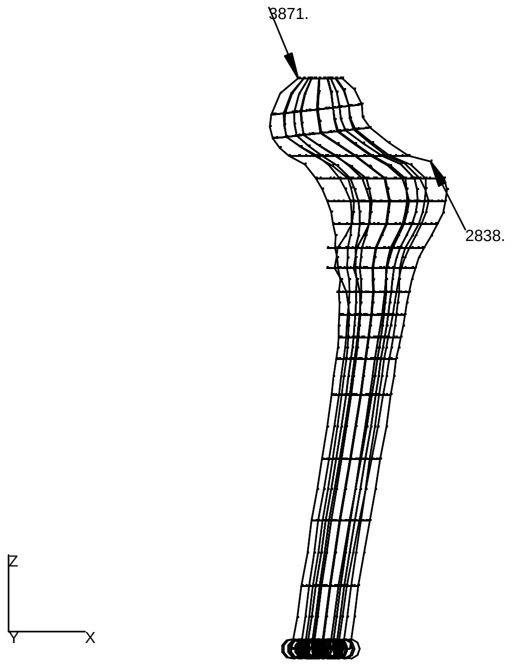

Figure 5: Problem 2: Proximal femur with compressive bending load

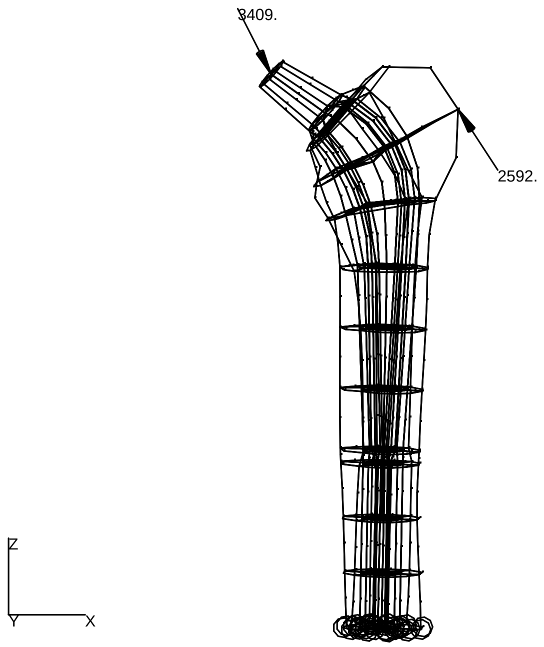

Figure 6: Problem 3: Proximal femur with c9001 hip implant

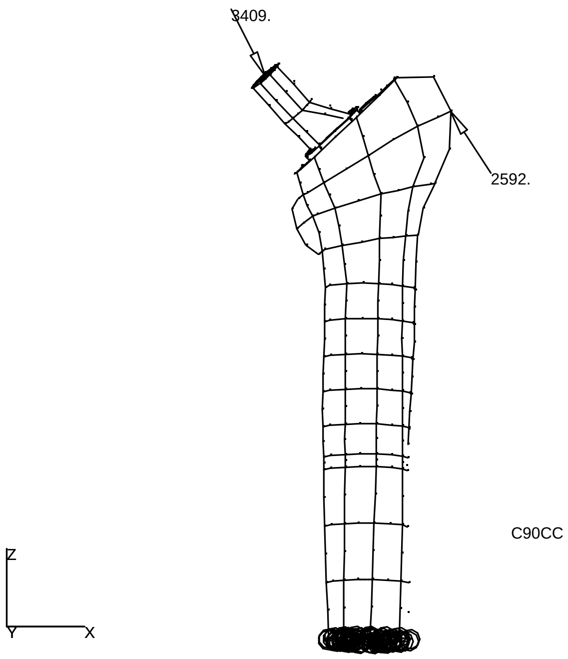

Figure 7: Problem 4: Proximal femur with cemented HSS 8911 cobalt-chrome implant

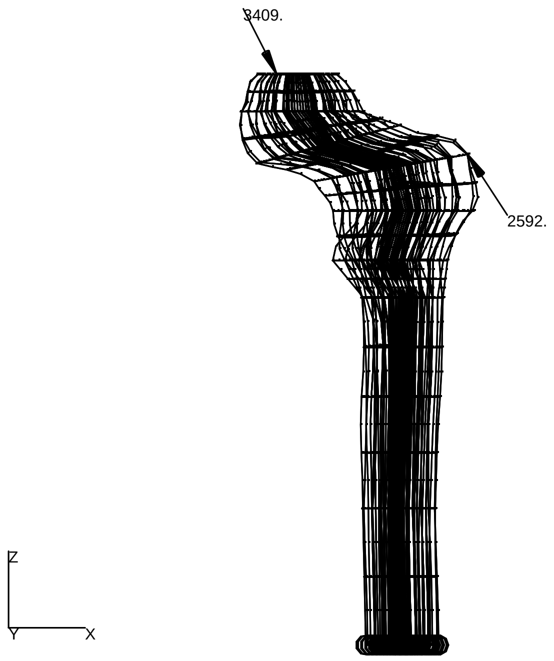

Figure 8: Problem 5: The proximal femur meshed with 1152 3-D finite elements

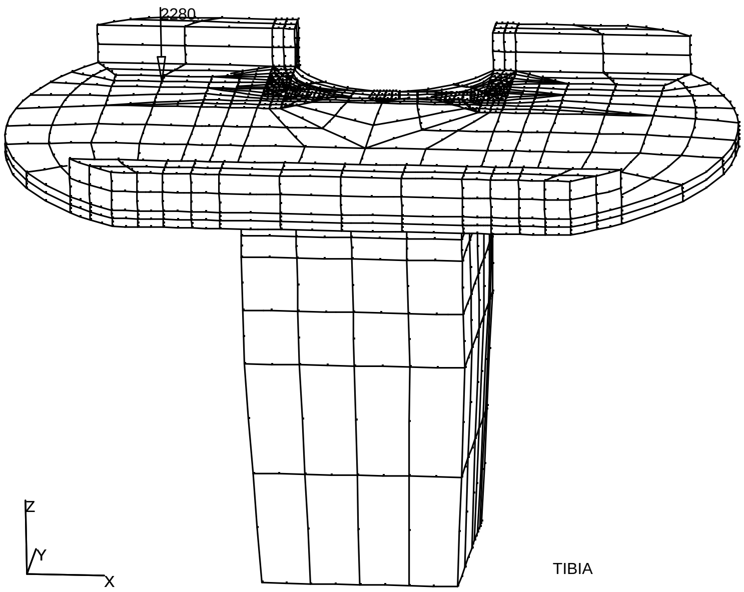

Figure 9: Problem 6: Posterior cruciate ligament sparing tibial component for use in total knee arthoplasty

	# elements (n_e)	# nodes (n_n)	# free dofs (n_d)	semi bandwidth (b)	Description
Problem 1	40	393	1158	87	cantilever beam with uniformly dist. load
Problem 2	272	1171	3366	300	bone only model of proximal femur
Problem 3	252	1101	3213	411	c9001 hip implant and proximal femur
Problem 4	752	3253	9567	864	cemented HSS 8911 cobalt chrome implant
Problem 5	1152	4869	14028	1044	bone only model of proximal femur
Problem 6	1504	8021	23628	2589	posterior cruciate ligament sparing tibial component
Problem 7	2562	12328	35985	3006	distal femur with Insall-Burstein implant

Table 1: Details of 7 finite element models

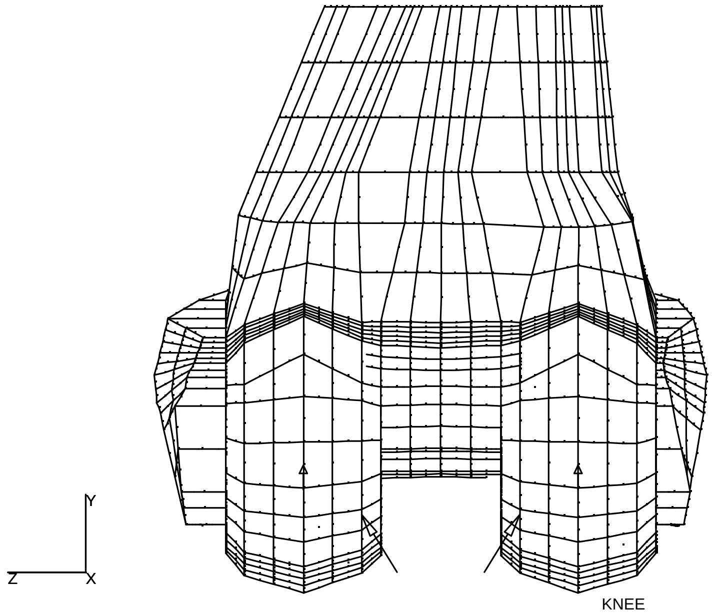

Figure 10: Problem 7: Distal femur with a cemented Insall-Burstein total condylar femoral component

onds). During the solution to system of equations, messages sent from one processor to another are only a few bytes long, as opposed to factorization, where a typical message is several thousand bytes long. Therefore, the processor idle time is considerably more than during factorization. Secondly, the vectors being operated upon by each processor are considerably shorter in length; b/p during solution as compared to b during factorization. Therefore, the asymptotic speed of BLAS is not attained during the solution stage. For example, the execution time of the subroutine `daxpy()`, which performs the operation $y \leftarrow y + a * x$ where x and y are vectors of length n and a is a scalar, is $11 + 0.964n$ microseconds. Since a `daxpy()` on vectors of length n takes $2n$ flops, the asymptotic speed is approximately 2.07 Mflops. However, in Problem 7, during the solution stage, the length of a vector being operated upon is 3006/512, which is 5 on some processors and 6 on others. This corresponds to an execution speed of 0.63-0.71 Mflops, which is significantly less than the asymptotic speed.

It is also important to note that I/O time is not negligible, even though it was excluded when computing the Mflop rate. I/O time includes the time spent in reading the input data file from the disk and writing out the nodal displacements and Gauss-point stresses to disk. This program did not use the concurrent file system directly attached to the hypercube; instead, it used the host file system. I/O on the host file system is a function of the load on the host due to other user processes and is particularly slow from the nodes. However, no attempt was made to speed up I/O.

	p	I/O	assembly	Cholesky	solution	other	total	Mflops
Problem 1	1	5	6	6	0.4	0.6	18	1.5
Problem 2	4	39	15	44	3	2	103	6.9
Problem 3	4	35	14	72	3	2	126	7.5
Problem 4	32	112	21	180	11	2	326	40
Problem 5	64	192	21	263	18	2	496	58
Problem 6	256	259	11	1005	35	1	1311	158
Problem 7	512	335	10	1905	59	2	2311	171

Table 2: Time in seconds for different operations during finite element analysis on the Ncube

Since I/O in finite element applications is only for reading in the input data and writing results after all computation is completed, the fractional cost is expected to be low, when the finite element analysis is a part of a more complex setup such as structural optimization.

The computational speed might appear to be very low considering the fact that each processor can compute at approximately 2.0 Mflops. There are two reasons for this. Firstly, since the finite element system of equations is banded, the amount of computation is much less than that required when the matrix is full (nb^2 as opposed to $n^3/3$). This reduces the amount of computation to be performed by a factor of more than 30, since the bandwidth is typically n/10. In particular, after row i of the Cholesky factor has been computed, in the dense case, all rows $j > i$ need to be updated by a rank-one update, whereas in case of a banded system, no more than b rows have to be updated. Although the size of the vectors being broadcast is less than in the dense case, as already mentioned, the percentage communication overhead is increased considerably, resulting in much lesser efficiency. The penalty is particularly high on the Ncube 6400, for which the ratio of computational speed to communication speed is very large. Secondly, except for problem 1, all other problems came from existing structural models of bio-mechanical systems. Their sizes were not designed to fit exactly on the hypercube. In particular, about 29 percent of the memory on each processor was unused in problem 7. The speed is likely to be higher when solving a large problem that occupies the entire memory on all node processors.

6 Conclusion

This paper demonstrates parallel 3-D finite element analysis on the Ncube 6400 using a parallel banded Cholesky factorization algorithm. It is shown that except for the factorization of the global stiffness matrix and the solution of the finite element system of equations, no interprocessor communication is necessary for obtaining high performance. This method can hence be used on any MIMD machine that can solve efficiently, a banded system of equations. Several problems arising from structural models of biomechanical systems were analyzed. The largest problem solved had 2562 elements and ran at a speed of 171 Mflops on 512 processors.

7 Acknowledgment

The first author would like to thank Avram Edidin, Ken Mann, John Dawson and Stefan Gabriel for providing finite element models used in this paper. Most of the initial work involving 2-D finite element analysis was performed on an Intel IPSC/2 at the Cornell Theory Center. Subsequent numerical experiments on large 3-D models were performed on an Ncube 6400 operated by the Sandia National Laboratory.

References

[Adams 84] Adams L. M. and Voigt, R. G., "A Methodology for Exploiting Parallelism in the Finite Element Process", *High Speed Computation*, Kowalik, J. K. (ed.), NATO ASI Series, Vol. F7, pp. 373-392, 1984.

[Carey 86] Carey, G. F., "Parallelism in Finite Element Modelling", *Communications in Applied Numerical Methods*, V2, pp. 281-287, 1986.

[Carter 89] Carter, W. T., Sham, T.-L. and Law, K. H., "A Parallel Finite Element Method and its Prototype Implementation on a Hypercube", *Computers and Structures*, V31, N6, pp. 921-934, 1989.

[Dunigan 91] Dunigan, T. H., "Performance of the Intel iPSC/860 and Ncube 6400 Hypercubes", ORNL/TM-11790, Oak Ridge National Laboratory, April 1991.

[Eisenstat 88] Eisenstat, S. C., Heath, M. T., Henkel, C. S. and Romine, C. H., "Modified Cyclic Algorithms for Solving Triangular Systems on Distributed-Memory Multiprocessors", *SIAM Journal on Scientific and Statistical Computing*, V9, N3, pp. 589-600, 1988.

[Farhat 87a] Farhat, C., Wilson, E. and Powell, G., "Solution of Finite Element Systems on Concurrent Processing Computers", *Engineering with Computers*, V2, pp. 157-165, 1987.

[Farhat 87b] Farhat, C. and Wilson, E., "A New Finite Element Concurrent Computer Program Architecture", *International Journal for Numerical Methods in Engineering*, V24, pp. 1771-1792, 1987.

[Geist 86] Geist, G. A. and Heath, M. T., "Matrix Factorization on a Hypercube Multiprocessor", *Hypercube Multiprocessors 86*, SIAM, pp. 161-180, 1986.

[Geist 90] Geist, G. A., Heath, M. T., Peyton, B. W., and Worley, P. H., "PICL: A Portable Instrumented Communication Library", ORNL/TM-11130, Oak Ridge National Laboratory, July 1990.

[Geist 91] Geist, G. A., Heath, M. T., Peyton, B. W., and Worley, P. H., "A Users' Guide to PICL: A Portable Instrumented Communication Library", ORNL/TM-11616, Oak Ridge National Laboratory, February 1991.

[Heath 88] Heath, M. T. and Romine, C. H., "Parallel Solution of Triangular Systems on Distributed-Memory Multiprocessors", *SIAM Journal on Scientific and Statistical Computing*, V9, N3, pp. 558-588, 1988.

[Johnsson 90] Johnsson, S. Lennart and Mathur, K. K., "Data Structures and Algorithms for the Finite Element Method on a Data Parallel Supercomputer", *International Journal for Numerical Methods in Engineering*, V29, pp. 881-908, 1990.

[Li 89] Li, G. and Coleman, T. F., "A New Method for Solving Triangular Systems on Distributed-Memory Message-Passing Multiprocessors", *SIAM Journal on Scientific and Statistical Computing*, V10, N2, pp. 382-396, 1989.

[McCarthy 89] "Bone Geometry from CAT scans for Finite Element Analysis: Image Processing Considerations and Modelling Effects", Ph.D. Thesis, Cornell University.

[Romine 88] Romine, C. H. and Ortega, J. M., "Parallel Solution of Triangular Systems of Equations", *Parallel Computing*, V6, pp. 109-114, 1988.

[Zienkiewicz 77] Zienkiewicz, O. C., "The Finite Element Method", Third Edition, McGraw Hill, London, 1977.