The Finite Element Method on a Data Parallel Computing System

Kapil K. Mathur and S. Lennart Johnsson*

Thinking Machines Corporation

245 First Street,

Cambridge, MA 02142, USA.

mathur@think.com, johnsson@think.com

Abstract

A data parallel implementation of the finite element method on the Connection Machine system CM–2® is presented. This implementation assumes that the elementary unit of data is an unassembled nodal point. In the context of the CM–2, each virtual processor represents an unassembled nodal point and nodal points shared between elements are replicated on different virtual processors. An algorithm for computing each elemental stiffness matrix concurrently, as well as different elemental stiffness matrices concurrently, without inter–processor communication is presented. The performance of the elemental stiffness matrix computation is in the range 1.6 – 1.9 GFlops s^{-1}. The sparse system of linear equations that results from the finite element discretization has been solved by a conjugate gradient method with a diagonal preconditioner. The rate of convergence of the conjugate gradient iterations for boundary conditions which correspond to uniaxial deformations depends nonlinearly on the order of interpolation of the elements and linearly on the mesh discretization. Sample code segments are provided to illustrate the programming environment on a data parallel architecture.

*Also, Departments of Computer Science and Electrical Engineering, Yale University, New Haven, CT 06520, USA.

1 Introduction

This article outlines a methodology for implementing the finite element method in a data parallel programming environment. It also discusses the choice of algorithms for the concurrent computation of elemental stiffness matrices, and the solution of the equilibrium equations. The Connection Machine System CM–2® is used as the model architecture for the data parallel implementation.

Many applications in science and engineering for which the finite element method is used require the creation of, and subsequent computation on, very large data sets. Stress analysis of structures and solids, thermal and electric field problems, and fluid dynamics problems are examples of such applications. The finite element method consists of discretizing a domain into a set of finite elements. First, local computations are performed on these elements to generate local data structures. The solution over an element is approximated by polynomials computed to approximate the true solution at certain nodal points on an element. The order of the approximation depends on the number of nodal points per element. The size of the local data structures depends on the order of approximation, and the number of degrees of freedom per node ($\sim un \times un$ for stress analysis, where n is the number of nodes per element, and u the number of degrees of freedom per node).

These local data structures are then assembled (either explicitly or implicitly) to form interaction equations between the global state variables being modeled. A linear system of sparse equations results from this assembly procedure, which can then be solved either by direct methods or by iterative methods. If a direct method is used to solve the system of linear equations, then an explicit assembly of the local data structure is required. This assembly is often part of the solver, as for instance is the case in the Harwell sparse matrix packages [3, 4, 5]. For an iterative solver the matrix assembly may be avoided, as described later.

The Connection Machine system model CM–2 [2] has a primary storage of 512 Mbytes distributed uniformly among 65536 1–bit processors operating synchronously on a single instruction stream (SIMD). There are 16 processors to a processor chip, and 4096 such chips are connected as a 12 dimensional hypercube. Two processor chips share a floating–point unit. Lattices of up to 12 dimensions are subgraphs of the hypercube, and can be emulated without any delay. The programming systems on the CM–2 include primitives for configuring the processors as a lattice of the desired dimension, and for relative and absolute addressing in the chosen lattice.

The Connection Machine system CM–2 needs a host computer. Currently, three families of host architectures are supported – the SUN–4 series, the VAX family with the BI–bus, and the Symbolics 3600–series. The Connection Machine system is mapped into the address space of the host computer, which stores the program and scalar data. Instructions that work on data stored on the Connection Machine system are sent to a control unit that broadcasts the instructions to all the processors.

Parallel extensions of the programming languages Fortran–77, C, and Common Lisp are available on the CM–2. The extensions are CM–Fortran [12], C* [11], and *Lisp [13] respectively. An application program written in any of these languages is translated by the appropriate compilers into code that makes direct calls to PARIS, the parallel instruction set of the Connection Machine system [14], and into code that is executed sequentially on the host computer. Direct calls to PARIS can also be made by the application program from any of the three high level programming languages. The data parallel implementation of the finite element method described here has been developed in the high level programming language *Lisp, the language of choice at the time this implementation was made. Sample code segments in CM–Fortran are provided to illustrate programming in a data parallel environment.

2 The finite element method

A brief outline of the finite element method [10, 15] is presented. Classically, in stress analysis, the finite element discretization of the application is formulated from a variational principle representing the statement of virtual work of the form

$$0 = -\int_V \delta \mathrm{Tr}(\epsilon\sigma) dV + \int_{S_\sigma} \delta \mathbf{u}^T \mathbf{T} dS, \tag{1}$$

where $\delta\mathbf{u}$ is the virtual displacement field compatible with the virtual strain $\delta\epsilon$ and σ is the Cauchy stress in equilibrium with the applied traction field $\mathbf{T}$. After discretizing the domain into finite elements, and introducing piecewise interpolation functions, which are non–zero only in the domain of one finite element, approximations for the displacement field and the corresponding strain field are obtained as

$$\{u\} = [N]\{U\}, \tag{2}$$

and

$$\{\epsilon\} = [N']\{U\}. \tag{3}$$

In the above equations, the matrix $[N]$ comprises of a set of interpolation functions and $[N']$ is a matrix containing the spatial derivatives of these interpolation functions. The final system of equations that results from the above approximations is of the form

$$[K]\{U\} = \{F\}, \tag{4}$$

where the global stiffness matrix $[K]$ is really a collection of elemental stiffness matrices

$$[K] = \sum_{\mathrm{i}} \left[K^{(\mathrm{i})}\right]. \tag{5}$$

The evaluation of the elemental stiffness matrices involves computation of interpolation functions which are local over the domain of a single finite element. The inherent parallelism in the concurrent generation of the elemental matrices is clear. However, the analysis in the next section shows that there is significant parallelism in the generation of individual elemental stiffness matrices too [9]. This allows for the exploitation of the various levels of parallelism in the finite element method that can be easily exploited on a data parallel architecture, such as the Connection Machine system. For higher order elements, this concurrency within a finite element significantly reduces the time taken to generate the elemental stiffness matrices.

3 Data parallel implementation of the finite element method

In a data parallel environment, the implementation of a computation intensive algorithm, such as the finite element method, involves a judicious choice for the logical unit for the data. All processors of the computing system can operate on such logical units concurrently. For example, if the computing system is configured as a three dimensional lattice of processors, and the logical unit is selected to be a variable $\mathbf{x}$, then on encountering a statement of the form

$$\mathbf{x} = 1.0, \tag{6}$$

the computing system sets the value of $\mathbf{x}$ to 1.0 in all processors forming the lattice.

For the finite element method, the logical unit on which the lattice of processors operate may be either

- a finite element from the finite element mesh, or
- an unassembled nodal point of the finite element mesh.

For a mesh composed of identical finite elements, the latter choice has several advantages [9]. Briefly, the two main advantages are:

1. for three dimensional brick elements, the degree of concurrency is a factor of n greater than the degree of concurrency obtained when finite elements are chosen as logic units of data, and
2. the local storage requirements for the elemental stiffness matrices is a factor of $\frac{1}{6}(3n + 1)$ less than the storage requirements for the first choice.

The influence of the amount of local storage and total storage available on the number of three dimensional Lagrangian elements that can be accommodated on a Connection Machine System with 512 Mbytes of total storage is shown in Table (1). The choice of the logical unit is clearly dependent on the available local and total storage. Table (2) summarizes the communication and arithmetic requirements for the two choices of the logical unit of data described above. The ratio of the number of data element transfers to arithmetic operations per logical unit of data is approximately the same for the two choices.

Based on the storage requirements, the finite element implementation described in this article assumes that the logical unit for the data parallel environment is an unassembled nodal point. All data associated with a logical

Table 1: The maximum number of degrees of freedom for three dimensional Lagrange elements that can be accommodated in 512 Mbytes of storage partitioned into 8 Kbytes per physical processor.

	order	nodes per elem.	virtual processor ratio	maximum deg. of freedom (approx.)	maximum number of elements (approx.)
Processor per unassembled node	$1 \times 1 \times 1$	8	8	780,000	260,000
	$2 \times 2 \times 2$	27	4	295,000	32,000
	$3 \times 3 \times 3$	64	1	148,000	9,000
	$4 \times 4 \times 4$	125	1	62,000	4,000
Processor per element, unsym.	$1 \times 1 \times 1$	8	2	390,000	130,000
Processor per element, sym.	$1 \times 1 \times 1$	8	4	780,000	260,000

Table 2: Data element transfers and arithmetic operations per logical unit for Lagrange elements in three dimensions.

Virtual processor	Type of communication	Element transfers	Arithmetic
Processor per unassem. node	Intra–element (all to all)	$2(n-1)u$	$(2nu-1)u$
	Inter–element (assembly)	$6u$	$3u$
Processor per element	Intra–element	-	$(2nu-1)nu$
	Inter–element (assembly)	$6n^{\frac{2}{3}}u$	$3n^{\frac{2}{3}}u$

unit is represented by a *virtual processor*. Each virtual processor is assigned a part of the memory, and the number of virtual processors is limited by the size of the total memory. The operations required on the data of a virtual processor is executed by a *physical processor*. The number of virtual processors per physical processor is called the *virtual processor ratio*. Each physical processor is shared by this many virtual processors. With a virtual processor representing an unassembled nodal point, any nodal point that is shared between finite elements is replicated on different virtual processors of the CM–2 as many times as the number of elements between which it is shared. This replication of the nodal points ensures that the virtual processor utilization and the load balance is 100% for all finite elements of the same shape and order of interpolation. It should be noted that even though the nodal points are replicated the data is not. The matrices are represented in unassembled form. The total storage requirements of the unassembled matrices exceeds the storage required by an assembled matrix. However, the excess storage is not to the extent of node replication [9].

For regular two or three dimensional finite element meshes the lattice emulation feature of the programming systems on the CM–2 is used advantageously. It simplifies the programming, and allows for efficient communication. Choosing virtual processors to represent a nodal point per element is compatible with configuring the processors of the Connection Machine system, CM–2, as a lattice of the appropriate dimension.

3.1 Generation of the elemental stiffness matrices

When one virtual processor of the Connection Machine system, CM–2, represents an unassembled nodal point, the generation of the elemental stiffness matrix for each element is shared by n processors. The implementation described in this article generates the rows of the elemental stiffness matrix corresponding to the nodal point represented by each processor concurrently. However the concurrent generation of the rows of the elemental stiffness matrices requires the use of Gauss quadrature to perform the numerical integration. The implementation described here performs this quadrature sequentially on each virtual processor. Briefly, the data parallel algorithm for the generation of the elemental stiffness matrices in parallel takes the form

```
        for all quadrature points, k
```

```
evaluate jacobian and shape function derivatives
   at the quadrature point, k.
add contribution of the quadrature point, k,
   to the rows of the elemental matrix stored
   on the virtual processor.
```

The implementation of the above pseudo–code in a high–level programming language available on the Connection Machine system is quite similar to the corresponding code for a sequential computing system. The major difference is in the fact that atleast two loop levels are not visible in the pseudo–code, namely, the loop corresponding to the number of finite elements in the mesh and the loop corresponding to the the number of rows in the elemental stiffness matrix. To emphasize this point, the corresponding pseudo–code for a sequential computing system is also presented:

```
for all finite elements in the mesh
   for all quadrature points, k
      evaluate jacobian and shape function derivatives
         at the quadrature point, k.
      for all rows, i, in elemental stiffness matrix
         add contribution of the quadrature point, k,
            to the rows, i, of the elemental stiffness matrix.
```

By comparing the two pseudo–codes, it is clear that the cumbersome book–keeping seen in most sequential codes to keep track of the various loop levels, is not present in a data parallel programming environment.

3.2 Solution of the linear system of equations

The data structure used in the generation of the elemental stiffness matrices may also be used in the solution phase, if an iterative method is used. The implementation of the finite element method described here uses the conjugate gradient method to solve the system of equations. For the results reported in this article a diagonal preconditioner was used for simplicity of implementation. Other preconditioners are currently being investigated in the context of solving large sparse systems that arise from discretization techniques such as the finite element method. For the conjugate gradient solver, the major computational effort is spent in the evaluation of the residual vector and the acceleration parameters [6]. The implementation described in this article does not assemble the elemental stiffness matrices into a global stiffness matrix. Instead, the residual vector corresponding to the linear system

$$Ax = b, \text{ where, } A = \sum_i A_i \tag{7}$$

is evaluated by using the following data parallel algorithm. Some critical segments of the algorithm have been expressed in CM–Fortran. For this algorithm, each virtual processor stores the following data values locally:

1. The three rows of the unassembled elemental stiffness matrix, K, corresponding to the nodal point represented by the virtual processor. The corresponding CM–Fortran DIMENSION statement for three dimensional linear brick elements is of the form

   ```
   CMF$LAYOUT KU(:SERIAL, , , ), KV(:SERIAL, , , ), KW(:SERIAL, , , )
     REAL KU(24, 32, 32, 32), KV(24, 32, 32, 32), KW(24, 32, 32, 32)
   ```

   The above statement directs the CM–Fortran compiler to create three local (in–processor) arrays of size 24 each. In addition, at run time the Connection Machine virtual processors will be configured as a three dimensional lattice of size $32 \times 32 \times 32$.

2. The three components of the displacement vector, $x = \{u \ v \ w\}^T$.

3. The three components of the load vector, $f = \{f_u \ f_v \ f_w\}^T$.

With the above storage scheme the following steps are necessary to compute the global residual vector during the conjugate gradient iteration process.

1. Broadcast the value of the displacement vector x to all other virtual processors forming the finite element. This operation is a segmented "all–to–all" broadcast [8] and is easily implemented by the use of nearest neighbour communications only. The implementation described here divides each finite element amongst n virtual processors, therefore, the total number of data elements that need to be communicated are atmost $O(n)$. After this broadcast, every virtual processor stores an elemental displacement vector, X.

2. Perform a local (in–processor) matrix–vector multiplication of the form

$$r = KX \qquad (8)$$

where K is of size $3 \times 3n$, X is $3n$ long and r is 3 long. The local vector r contains the three unassembled components of the residual corresponding to the nodal point represented by the processor. The corresponding code segment in CM–Fortran is

```
RU = 0.0
RV = 0.0
RW = 0.0
DO I=1,24
   RU = RU + KU(I, :, :, :) * X(I, :, :, :)
   RV = RV + KV(I, :, :, :) * X(I, :, :, :)
   RW = RW + KW(I, :, :, :) * X(I, :, :, :)
END DO
```

3. Assemble the local residual vector, r. As before, this operation requires nearest neighbour communications. Only eighteen nearest neighbour communications are required in three dimensions to assemble the entire residual vector. The following code segment in CM–Fortran illustrates the assembly process for the u component of the local residual.

```
(WHERE EAST) RU = RU + EOSHIFT(RU, 1, 1)
(WHERE WEST) RU = EOSHIFT(RU, 1, -1)
(WHERE SOUTH) RU = RU + EOSHIFT(RU, 2, 1)
(WHERE NORTH) RU = EOSHIFT(RU, 2, -1)
(WHERE FRONT) RU = RU + EOSHIFT(RU, 3, 1)
(WHERE BACK) RU = EOSHIFT(RU, 3, -1)
```

In the above code segment EAST, WEST, etc. are boolean arrays which define the right–hand and left–hand boundaries of each finite element in the three dimensions, respectively. Each call to EOSHIFT generates PARIS code that results in only nearest neighbour communications on the lattice of virtual processors.

4. Now compute the real residual

$$r = r - f, \tag{9}$$

or in terms of CM–Fortran

```
RU = RU - FU
RV = RV - FV
RW = RW - FW
```

4 Applications

This section first investigates the influence of the interpolation order and quadrature rule on the time taken for the generation of the elemental stiffness matrices for all the elements in the mesh. Next, the performance of the conjugate gradient solver is investigated by analyzing the time required for the different functions in an iteration. Finally, some results from some simulations on a rod clamped at one end and pulled by a distributed force on the other are presented.

Table (3) summarizes the performance of the single–precision data parallel implementation of the generation of the elemental stiffness matrices. The timings reported are based on a Connection Machine system CM–2 with 16,384 physical processors equipped with floating-point hardware. The clock rate was 7 MHz. The finite element meshes used were such that all the physical processors of the computing system were fully utilized. The floating point operations per second extrapolated to a full Connection Machine system CM–2 with 65,536 physical processors is in the range of 1.5 – 1.9 GFlops s^{-1} for a Sun–4 host computer and is approximately 1.5 – 1.7 GFlops s^{-1} for the Symbolics host computer. The floating point rate improves significantly as the virtual processor ratio increases. Note that in the algorithm for elemental stiffness matrix computation outlined in Section 3.1 no inter–processor communication is required.

To evaluate the performance of the conjugate gradient solver, the individual steps outlined in the previous section for the computation of the matrix vector product were timed. This product comprises almost all the work for the evaluation of the global residual. Table (4) shows the time taken by

Table 3: Time taken to generate single–precision elemental stiffness matrices of finite element meshes which fully utilize a Connection Machine system CM–2 with 16,384 physical processors at a virtual processor ratio of one.

Interpolation Order	Number of nodes per element	Quadrature Order	CM time Sun–4	CM time Symbolics
$1 \times 1 \times 1$	8	$2 \times 2 \times 2$	0.233	0.231
$2 \times 2 \times 2$	27	$2 \times 2 \times 2$	0.634	0.726
$2 \times 2 \times 2$	27	$3 \times 3 \times 3$	2.641	2.441
$3 \times 3 \times 3$	64	$3 \times 3 \times 3$	5.297	5.627
$3 \times 3 \times 3$	64	$4 \times 4 \times 4$	12.144	13.445

each step of one iteration of the conjugate gradient process carried out in single–precision. The finite elements were all three dimensional Lagrange elements of first order, and the timings were measured at a virtual processor ratio of one. At this virtual processor ratio, the default configuration of the physical processors on a Connection Machine processor chip for the emulation of a three-dimensional lattice is a $2 \times 2 \times 4$ sublattice. Therefore, at a virtual processor ratio of one, two unassembled finite elements are placed on a processor chip. All intra–element communications (corresponding to the all–to–all operation) are on–chip. In contrast, inter–element communications corresponding to the assembly operation are mostly off–chip. The on–chip communications rate is about one order of magnitude greater than the off–chip communications rate. As the virtual processor ratio increases a larger fraction of the total communication becomes on–chip communication. The off–chip communication is minimized if the aspect ratio of the domain that is placed on a chip is as close to one as possible, since the communication is equally frequent in all three lattice dimensions.

In the numerical study of the influence of the order of interpolation, the discretization parameter (or the number of elements), the Poisson ratio, and the aspect ratio of the elements on the convergence behavior of the conjugate gradient method with a diagonal preconditioner, all computations were performed using double–precision floating point arithmetic. The finite element mesh was constructed with elements with interpolation order $p_1 \times p_2 \times p_3$. The number of finite elements in this mesh were $N_1 \times N_2 \times N_3$. The boundary conditions on the mesh corresponded to fixing the face of the mesh defined by the y–z plane at x=0 and applying a distributed force on the face defined

Table 4: Time per iteration of the conjugate gradient method for first order Lagrange elements in three dimensions. The Connection Machine system CM–2 had 16384 physical processors, and the virtual processor ratio was one.

	Time (milli–second)	%
"all–to–all" broadcasting	9.3	40.8
Local matrix vector product	3.8	16.7
Assembly	5.2	22.8
Acceleration parameters	1.9	8.3
Update displacement vector	2.6	11.4
Time per iteration	22.8	100.0

Table 5: Influence of the mesh discretization parameter, h on the convergence behavior of the conjugate gradient method for a mesh with straight edges at various orders of interpolation, p. Mesh discretization $= N \times 1 \times 1$; order of interpolation for a finite element $= p \times 1 \times 1$; total number of degrees of freedom $= 12 \times (p \times N + 1)$. The convergence tolerance for the iterative solver was a normalized global residual of 5.0×10^{-8}.

N, number	Conjugate gradient iterations				
of elements	$p = 1$	$p = 2$	$p = 3$	$p = 4$	$p = 5$
100	100	215	436	652	1083
200	200	442	869	1331	2166
300	300	670	1301	2003	3251
400	400	899	1773	2673	4319
500	500	1129	2167	3345	5400

by the y–z plane at x=L, the other end of the bar.

In the first three sets of simulations the Poisson ratio of the material was set to be zero, thus forcing the deformation field to be one dimensional. The first set of simulations investigated the influence of the mesh discretization parameter, h, in one dimension at constant order of interpolation, p. Table (5) and Figure (1) show the results of this set of simulation for different values of p. Clearly, the number of iterations required for the conjugate gradient iterations to converge is $O(N)$ or $O(\frac{1}{h})$, as expected for a second–order elliptic differential operator for which the condition number is $O(\frac{1}{h^2})$, and therefore the convergence rate for the conjugate gradient method $O(\frac{1}{h})$

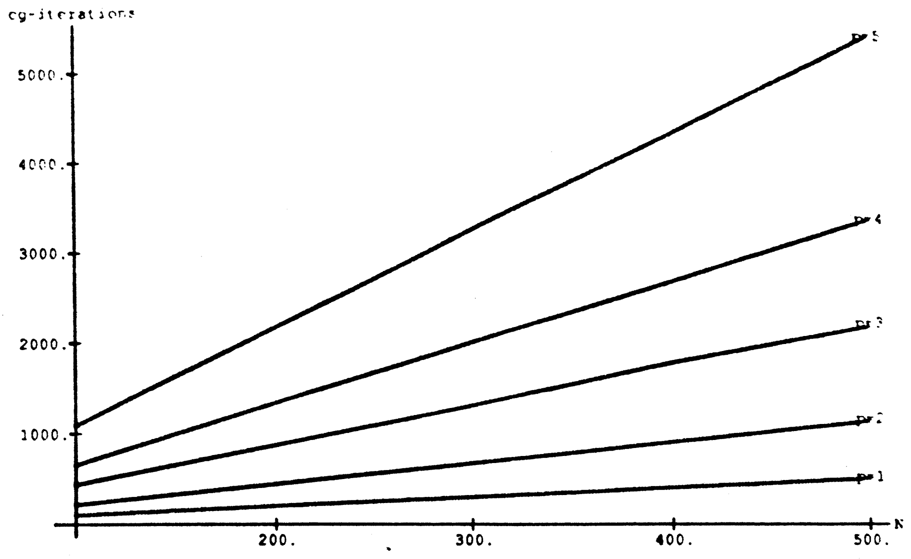

Figure 1: Influence of the mesh discretization parameter, h, on the convergence behavior of the conjugate gradient method with diagonal scaling for several interpolation orders. The iterative solver was assumed to have converged when the normalized global residual reached a value of 5×10^{-8}.

[1, 7].

The second set of simulations investigates the influence of the order of interpolation p on the convergence behavior of the conjugate gradient method. The mesh discretization was kept fixed at $120 \times 1 \times 1$ for this set of simulations. The results of these simulations have been summarized in Table (6) and Figure (2). The dependence of the conjugate gradient iterations on the order of interpolation is significantly non–linear. Based on the results summarized in Table (6) and on other numerical experiments, the non–linear dependence of the conjugate gradient iterations on p has a functional form

$$N_{\text{cg}} \sim N \times p^{1.4} \tag{10}$$

where N_{cg} is the number of conjugate gradient iterations required for convergence.

The third set of simulations varied the interpolation order and the number of elements in the x–dimension such that the number of nodal points, and consequently the number of degrees of freedom, were kept fixed. The simulation results are summarized in Table (7) and Figure (3) for the case

Table 6: The convergence behavior of the conjugate gradient iterations with diagonal scaling as a function of the order of interpolation, p. Mesh discretization $= 120 \times 1 \times 1$; order of interpolation for a finite element $= p \times 1 \times 1$; total number of degrees of freedom $= 12 \times (120 \times p + 1)$. The convergence tolerance for the iterative solver was a normalized global residual of $\leq 5.0 \times 10^{-8}$.

p, order of interpolation	Total degrees of freedom	Conjugate gradient iterations
1	1452	120
2	2892	260
3	4332	522
4	5772	787
5	7212	1309

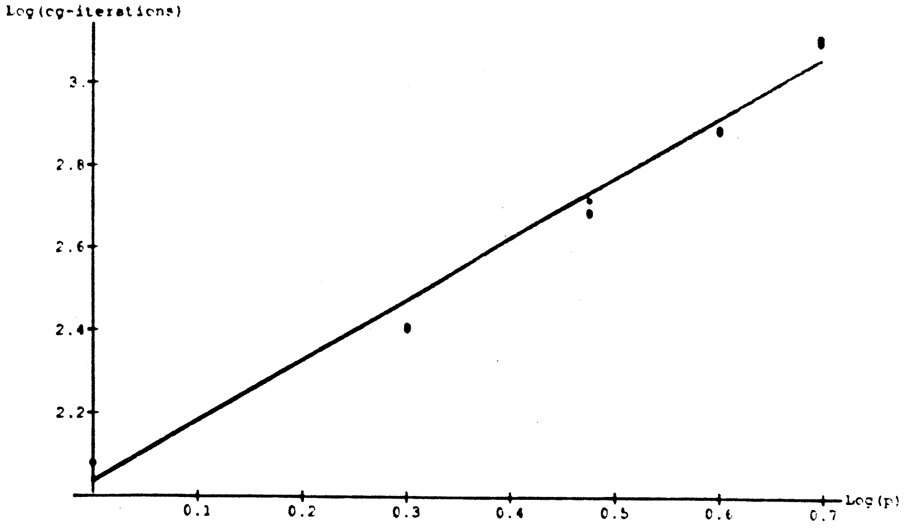

Figure 2: Number of conjugate gradient iterations required for the normalized global residual to reach a cut–off value as a function of the order of interpolation, p. The mesh discretization parameter, h was kept fixed. Different points for a fixed order of interpolation represent different values of the normalized global residual (corresponding to 1×10^{-3}, 1×10^{-4}, 1×10^{-5}, and 5×10^{-8} respectively).

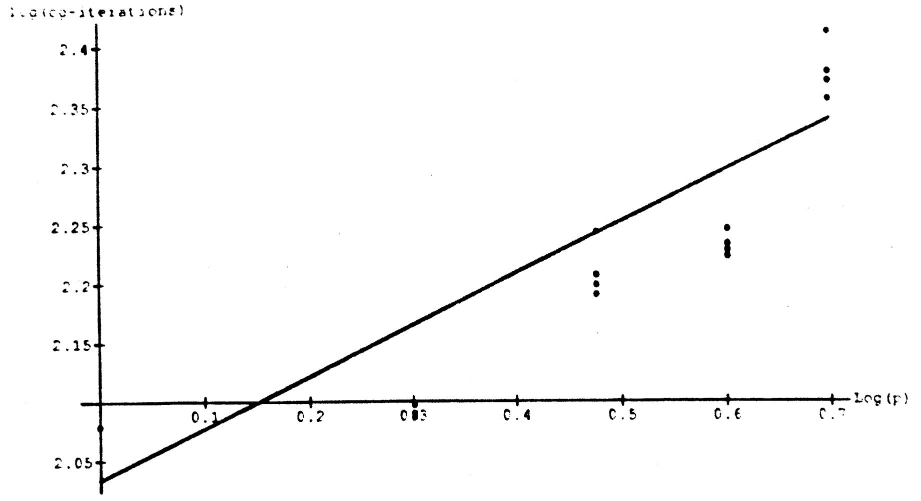

Figure 3: The convergence behavior of the conjugate gradient method for different order of interpolation. The total number of degrees of freedom were kept fixed so that the nodal discretization for the meshes was 121 × 2 × 2. Different points for a fixed value of the interpolation order represent different values of the convergence tolerance.

Table 7: Influence of the interpolation order on the convergence behavior of the conjugate gradient method at a fixed number of degrees of freedom. N is the number of elements and p is the order of interpolation.

N	p	Number of conjugate gradient iterations			
		1.0×10^{-3}	1.0×10^{-4}	1.0×10^{-5}	5.0×10^{-8}
120	1	120	120	120	120
60	2	122	122	123	125
40	3	155	158	161	175
30	4	167	169	171	176
24	5	227	235	239	258

when the interpolation order for the finite elements composing the mesh was $p \times 1 \times 1$ and the number of elements in the three dimensions were $N \times 1 \times 1$.

The number of iterations required for the normalized global residual error to reach a value of 1.0×10^{-3}, 1.0×10^{-4}, 1.0×10^{-5}, and 5.0×10^{-8} for the five meshes are shown in Table (7) and Figure (3). Comparing Figures (2) and (3), the convergence behavior depends on the interpolation order of the elements less strongly in this set of simulations, than in the second set. These results and results from other numerical experiments not reported here, fit the equation

$$N_{\text{cg}} \sim (N \times p)\, p^{0.4} = N \times p^{1.4} \tag{11}$$

The above sets of simulations were repeated for a non–zero Poisson ratio ($\nu = 0.3$). These simulations correspond to fully three dimensional deformations. The convergence behavior of the conjugate gradient iterations with diagonal scaling for this set of simulations yield a similar functional form to within a constant factor:

$$N_{\text{cg}} \sim 1.5N \times p^{1.4}$$

Finally, a set of simulations were performed in which the aspect ratio of the finite elements was varied. For the geometry and boundary conditions that were used in the simulations, the number of iterations did not depend on the aspect ratio of the elements when the global stiffness matrix was scaled such that the diagonal entries were all one.

Some interesting conclusions can be drawn from the last set of simulations. The meshes used in this set of simulations had 121 nodes in the x–direction. As expected, the number of iterations required by the conjugate gradient solver is 120 for linear elements, which is the same as the number of elements in that direction. Without a better preconditioner, the convergence rate for this discretization cannot be improved because the propagation length of the iteration process is equal to the interpolation order of the element. Although the propagation length of the iteration process increases with the order of the element, the number of iterations required for convergence to achieve the same tolerance for the normalized global residual increases. This investigation concludes that at least for meshes with straight edges the influence of the interpolation order of the elements on improving the convergence of the conjugate gradient method is minimal.

References

[1] Owe Axelsson and V.A. Barker. *Finite Element Solutions of Boundary Value Problems.* Academic Press, 1984.

[2] Thinking Machines Corp. Connection machine model cm–2 technical summary. Technical Report HA87-4, Thinking Machines Corp., 1987.

[3] Iain S. Duff. Ma28 – a set of fortran subroutines for sparse unsymmetric linear equations. Technical Report AERE R8730, HMSO, London, AERE Harwell, 1977.

[4] Iain S. Duff. Ma32 – a package for solving sparse unsymmetric systems using the frontal method. Technical Report AERE R10079, HMSO, London, AERE Harwell, 1981.

[5] Iain S. Duff and John K. Reid. A set of fortran subroutines for solving sparse symmetric sets of linear equations. Technical Report AERE R10533, HMSO, London, AERE Harwell, 1982.

[6] Gene Golub and Charles vanLoan. *Matrix Computations.* The Johns Hopkins University Press, 1985.

[7] Anne Greenbaum, Congming Li, and Han Zheng Chao. Parallelizing preconditioned conjugate gradient algorithms. Technical report, Courant Institute of Mathematical Sciences, New York University, November 1988.

[8] S. Lennart Johnsson and Ching-Tien Ho. Spanning graphs for optimum broadcasting and personalized communication in hypercubes. *IEEE Trans. Computers*, 38(9):1249–1268, September 1989.

[9] S. Lennart Johnsson and Kapil K. Mathur. Data structures and algorithms for the finite element method on a data parallel supercomputer. *International Journal of Numerical Methods in Engineering*, 1989. Department of Computer Science, Yale University, Technical Report YALEU/DCS/RR-743, Technical Report CS89-1, Thinking Machines Corp., December, 1988.

[10] J. Tinsley Oden and Graham F. Carey. *Finite Elements: Mathematical Aspects*, volume IV. Prentice-Hall, 1983.

[11] John Rose and Guy L. Steele. C*: an extended C language for data parallel programming. Technical report, Thinking Machines Corp., 1987.

[12] Rosenbloom. Using Fortran-8X style of programming. Technical report, Thiniking Machines Corp., 1987.

[13] Thinking Machines Corp. **Lisp Release Notes*, 1987.

[14] Thinking Machines Corp. *Paris Release Notes*, 1987.

[15] O.C. Zienkiewicz. *The Finite Element Method.* McGraw-Hill, 1967.