The Finite Element Method on a Data Parallel Architecture

Kapil Mathur and S. Lennart Johnsson

YALEU/DCS/TR-740
September 1989

To appear in the "Fifth International Symposium on
Numerical Method in Engineering",
Luzanne, Sept. 11 -15, 1989.

The Finite Element Method on a Data Parallel Architecture

Kapil K. Mathur and S. Lennart Johnsson[1],
Thinking Machines Corporation,
245 First Street,
Cambridge, MA 02142
mathur@think.com, johnsson@think.com

Abstract

Two data parallel implementations of the finite element method are discussed using the Connection Machine® system, CM–2, as the model architecture. The first implementation focuses attention on discretizations composed of brick elements which allows for the exploitation of the lattice emulation capability of the model architecture. The second implementation describes a data parallel algorithm for more general discretizations comprising of different types of elements and arbitrary geometries. The data parallel implementation for the generation of the local data structures (elemental stiffness matrices) and for the solution of the resulting linear system using an iterative solver is described in detail for the two representations. The two algorithms are compared in terms of performance, storage requirements, and communication complexity. Peak performances well in excess of 1.5 GFlops s^{-1} have been measured for the evaluation of the elemental stiffness matrices. The peak performance of the iterative solver is 850 Mflops s^{-1}.

1 Introduction

The finite element method is frequently used for solving boundary and initial value problems that arise in many scientific simulations. Data sets associated with such scientific and engineering simulations are often very large. Consequently, there is a need for supercomputing. All current supercomputers are parallel architectures, and future supercomputers are expected to have a large number of memory modules and processing units. Orders of magnitude increase in performance can no longer be achieved by improvements to conventional architectures. Massively parallel architectures, often referred to as *data parallel* offer supercomputing performance. The peak performance is orders of magnitude greater than what is expected from conventional *control parallel* supercomputer architectures which usually offer a considerably lower degree of concurrency.

Almost all the existing general–purpose finite element programs have been developed for sequential machines. Recently, several researchers have investigated the finite element method in the context of parallel computations on multiprocessors. Carey et al.

[1]Also affiliated with the Department of Computer Science, Yale University

[1986,1988], Farhat [1987], and Law [1986] discuss most of the general methods for parallelizing the finite element method primarily with coarse–grained configurations with shared or distributed memory. For data parallel architectures, it is necessary to re–evaluate the choice of data structures and algorithms. Important issues are load balance and data motion across the processors.

This article summarizes two data parallel implementations of the three dimensional finite element method on the Connection Machine system CM–2. The first implementation describes a mapping strategy for meshes composed of brick elements with arbitrary interpolation order. This mapping strategy takes advantage of the lattice emulation capability of the model data parallel architecture which allows for fast nearest neighbor communication. The second implementation describes a more general data representation which is valid for meshes comprising of different types of elements and for domains with arbitrary geometries. A sample application with more than 25,000 degrees of freedom has been simulated with both the implementations. Detailed performance measurements are reported along with the convergence behavior of the iterative method.

2 The Data Parallel Programming Model

Algorithms for a data parallel programming environment should be designed based on the structure and representation of the physical and computational domains. An essential characteristic of data parallel algorithms is the choice of the elementary objects. These units of data are subject to the same transformations concurrently. Different classes of elementary objects are subject to different transformations, but may be operated upon concurrently. An algorithm is expressed as a sequence of transformations of the state of an elementary object, and interactions between elementary objects.

2.1 The Connection Machine Model CM–2

2.1.1 Architecture

This study uses the Connection Machine system as the model architecture. The Connection Machine system model CM–2 [Hillis, 1985, Technical summary, 1989], has a primary storage expandable up to 2 Gbytes distributed evenly among 64K bit–serial processors. There are 16 such processors to a processor chip. Two processor chips share an industry standard floating–point unit. The processor chips are interconnected as a 12–dimensional boolean cube. The topology of this network can efficiently emulate arbitrary lattices. For arbitrary communication patterns, the Connection Machine system is equipped with a general purpose communication hardware called the "router", which selects the shortest paths between the source and the destination of a message.

2.1.2 Programming languages

High–level programming languages currently available on the Connection Machine system are *Lisp, C*, and CM–Fortran. They are parallel additions of Common Lisp, C++, and Fortran–77, respectively. The most important extensions are the existence of a parallel data type and operations that can be performed concurrently on the parallel variable. An instance of the parallel variable is allocated across the entire configuration of the Connection Machine. The elements of a parallel variable are operated upon concurrently by a single instruction. It is also possible to operate concurrently on distinct subsets of a parallel variable. Since no enumeration of the elements is required, one or several loop levels disappear from the corresponding sequential code.

In a data parallel model each instance of an elementary object is assigned to a unique processor. However, in simulations involving very large data sets, the number of elementary objects may far exceed the number of *physical processors*, but the application may still fit in the primary storage of the computer. The data parallel programming model on the Connection Machine system supports the notion of *virtual processors*. Virtual processors are distributed evenly among the physical processors. Virtual processors assigned to the same physical processor time share it for execution, and are assigned distinct portions of its memory. The number of virtual processors per physical processor is called the *virtual processor ratio* for the configuration and is under the control of the application program.

In general, concurrent operations available on data parallel architectures are performed by all active processors. The state of the processors may be changed by the use of conditional or sub–selection mechanisms. Several sub–selection mechanisms are available in the parallel extensions of the high level programming languages. Conditional statements can be used to make the sub–selection based on variable values or on the addresses of the processors.

3 Data parallel implementation of the Finite Element Method

Important design issues for the data parallel implementation of the finite element method are: parallel grid generation for transforming the physical domain to a discretized computational domain, data structures for representing this computational domain, generation of the elemental stiffness matrices concurrently, and concurrent solution of the resulting sparse linear system. For each of these issues, several factors need to be considered. Some of the more important issues are storage requirements, communication complexity, parallel arithmetic complexity, uniformity of computations (load balance), and programming complexity.

3.1 Data structure

On a data parallel architecture computations for all elementary objects are performed concurrently. In the context of the finite element method, there are two possible choices for the elementary objects:

- one processor representing a finite element, or
- one processor representing one nodal point *per* finite element, that is, nodal points which are shared between elements are replicated on separate processors.

When a processor is assigned to a nodal point, the processor performs all computations associated with that nodal point, both in the evaluation of the elemental stiffness matrices and during the solution phase. In assembled form, some nodal points are shared by more elements than others. Consequently, the computational load among the nodal points is not uniform. By having a processor represent one nodal point per finite element (or an unassembled nodal point), the computational effort required for all nodal points is identical. The location of the nodal point in the computational domain (interior or on the boundary of the mesh) and the finite element does not influence the computational effort. In this data representation, the elemental stiffness matrix is distributed over the processors representing the nodes of a finite element. The computation of the elemental stiffness matrix can be organized such that no communication is required. For this representation, a suitable data structure is a *split* lattice, that is, each surface representing a boundary between two brick elements is duplicated. When the elementary object is chosen to be one finite element, and the region consists of a mesh of elements. The mapping between the elements and a lattice of processors is straightforward.

The Unassembled Nodal Point Mapping: An unassembled nodal point of the mesh is mapped on to a processor of the data parallel architecture. Nodal points that are shared between elements are replicated on separate processors. Only the information about the geometry (the global coordinates) needs to be replicated on the processors representing the same nodal point. Figure (1) shows this mapping for a two dimensional finite element mesh with four bilinear elements labeled A, B, C, and D. The mesh has nine nodes labeled one through nine. The 3×3 layout of nodes is mapped on to a 4×4 lattice of processors. Nodal point labeled five is shared by all four elements and is consequently placed on four separate processors. In this mapping scheme, each element has a subset of processors working towards the generation of the elemental stiffness matrices and in the evaluation of the sparse matrix–vector product during the solution phase. Moreover, if the elemental stiffness matrices are not explicitly assembled into a global stiffness matrix, the storage required to store these matrices is shared evenly by the subset of processors. Each processor stores the rows of the elemental stiffness matrix corresponding to the unassembled nodal point represented by the processor. This corresponds to a $u \times nu$ matrix per processor, where u is the number of degrees of freedom per node and n is the number of nodes per

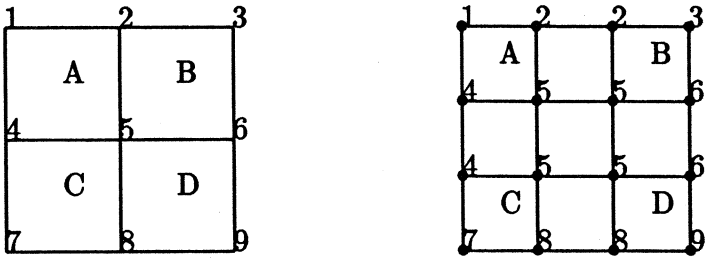

Figure 1: Mapping the physical domain composed of brick/rectangular elements on to the data parallel architecture. In the example shown above, the finite element mesh comprises of four bilinear elements labeled A–D. The nodes are labeled one to nine. The processors of the data parallel architecture are represented by dots.

element. This is especially advantageous for three dimensional discretizations because of limited local storage per processor of the model data parallel architecture. When the elemental stiffness matrices are distributed evenly over a subset of processors, higher order elements can be used in the construction of the mesh.

The data parallel implementation of the finite element method for this mapping can be divided into two distinct sections – the evaluation of the elemental stiffness matrices and the solution of the resulting sparse linear system. For the mapping described here, the evaluation of the elemental stiffness matrices requires no communication. The numerical quadrature for each matrix element is performed sequentially. However, several matrix elements can be computed in parallel. This is in addition to the concurrency already present between different finite elements. Further details are available in Johnsson and Mathur [1989]. The resulting sparse linear system has been solved by a conjugate gradient method with diagonal scaling. In the data parallel implementation of the conjugate gradient method, the main computational and data communication effort is in the sparse matrix–vector product of the form

$$\{r\} = \{b\} - [A]\{x\}, \tag{1}$$

where the coefficient matrix $[A]$ is not explicitly assembled but is stored as

$$[A] = \sum_{i}^{n} \left[A^{(i)}\right]. \tag{2}$$

For the mapping scheme used in the implementation, this sparse matrix–vector product involves:

1. Accumulation of the unknowns from the processors representing the unassembled nodes. All processors in the subset of processors forming the element require the unknowns from every other processor in this subset. This type of communication is often termed as a segmented "all to all" broadcast [Johnsson and Ho, 1989] and can be implemented very efficiently by the use of nearest neighbor communication when the processors of the data parallel architecture are configured as a lattice. In addition to the matrix containing the unassembled rows of the nodal point represented by the processor, after a segmented all to all broadcast every processor also stores a vector of length nu containing the accumulated unknown vector.

2. A local matrix–vector product $[(u \times nu) \times (nu \times 1)]$ is then performed by every processor. After this multiplication, every processor contains the unassembled contribution of the nodal point to the product vector $(u \times 1)$.

3. Finally, the product vector is assembled by performing nearest neighbor communication among processors representing the same nodal point.

For the example two dimensional mesh shown in Figure (1) and in simulations involving stress analysis, the three sections described above are:

1. For all elements (A–D) accumulation of the eight displacement components associated with each element.

2. Multiplication of the two rows of the unassembled stiffness with the accumulated displacement vector.

3. Assembly over all processors representing replicated nodal points (nodes labeled 2, 4, 5, 6, and 8).

One Element Per Processor Representation: The second implementation describes a data parallel algorithm for more general discretizations comprising of different types of elements and arbitrary geometries. The processors of the data parallel architecture are configured with two views (or processor sets). The first set views the processors of the data parallel architecture to represent a *collection of finite elements.* The local data structures (elemental stiffness matrices) are evaluated in this processor set. For discretizations composed of different types of elements, the SIMD nature of the model architecture requires that the local storage requirements, parallel floating point arithmetic, and communication complexity be based on the element in the set with the largest data structure. For example, for a mesh composed of trilinear bricks and triquadratic bricks, the allocated storage per processor and the time required to evaluate the elemental stiffness matrices are all based on the requirements of the triquadratic elements. In this example, processors which represent trilinear bricks will perform some redundant operations. The second view of the data parallel architecture is that of a *set of nodal points.* The sparse

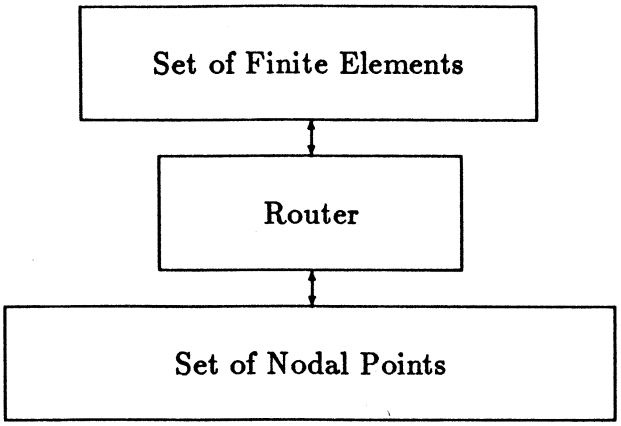

Figure 2: The two views of the processors of the data parallel architecture used to map discretizations composed of different types of elements and having arbitrary geometries. The box labeled "router" is the general purpose communication hardware on the model architecture. The arrows represent the direction of communication between the two processor sets.

matrix–vector multiplication performed by the iterative solver requires communication between the two processor sets. This communication is in general long range, in contrast to the nearest neighbor communication in the first implementation. The interaction between the two processor sets is shown in Figure (2).

As before, the implementation of the finite element method can be segmented into two parts. The generation of the elemental stiffness matrices uses a data parallel algorithm which is similar to the one used by the first implementation. The sparse matrix–vector product required by the iterative method for the evaluation of the global residual is more complex. First, the nodal values are accumulated from the processor set representing the nodal points as local vectors in the processor set representing elements. A local matrix–vector multiplication is then performed by each processor in this processor set. Finally, the local vectors are then assembled into global nodal values which are stored in the processor set representing nodal points. Clearly, communication between the two processor sets is required. The communication pattern that is generated depends on the connectivity of the elements in the mesh. In general, this communication pattern is long range and requires the use of the "router".

4 Application

To evaluate the performance of the data parallel implementations, several simulations involving three dimensional stress fields were performed. Single precision floating point arithmetic was used in this timing analysis. The finite element meshes were constructed

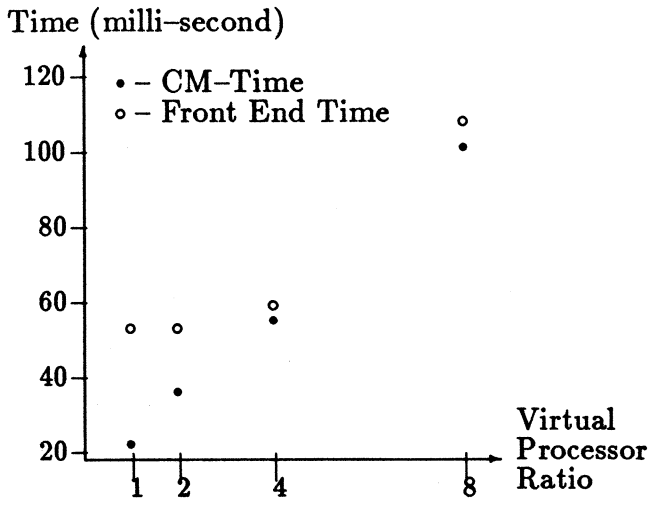

Figure 3: The front–end and the CM time for the evaluation of the elemental stiffness matrices as a function of the virtual processor ratio. All reported times are in milli–seconds for single precision floating point operations.

using three dimensional Lagrange elements. For these simulations, $u = 3$ and $n = (p+1)^3$ where p is the interpolation order of the Lagrange elements. Figure (3) shows the CM time and the front–end time for the generation of the elemental stiffness matrices for trilinear brick elements as a function of the virtual processor ratio when the unassembled nodal point data representation is used. A peak performance corresponding to 2.4 Gflops s^{-1} is obtained at a virtual processor ratio of eight with a CM utilization of over 96%. After a virtual processor ratio of four, the difference between the front–end time and the CM time is approximately a constant. Most of this difference is the time used to evaluate the shape functions at the quadrature points in the local coordinate system. These functions are the same for *all* processors and are therefore evaluated on the front end.

The performance of the conjugate gradient solver with a diagonal preconditioner is shown in Figure (4) as a function of the virtual processor ratio when one processor represents an unassembled nodal point. The peak performance measured is approximately 850 Mflops s^{-1} at a virtual processor ratio of eight where the CM utilization is approximately 95%. As before, as the virtual processor ratio increases, the front–end overhead reduces and consequently the CM utilization improves significantly.

Finally, the convergence behavior of the conjugate gradient method with diagonal scaling is presented. The geometry of the physical domain corresponds to a square plate ten units long, one unit thick and ten units wide. One face in the length–width plane was fixed and the other face has applied traction boundary conditions which corresponded to a uniform unit load. This domain was discretized by a mesh comprising of trilinear bricks. The mesh had ten elements in the length dimension, one element in the thickness dimension

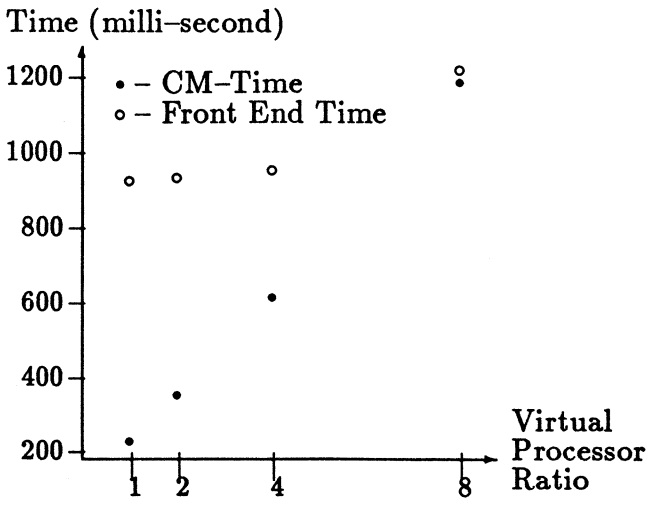

Figure 4: The time per conjugate gradient iteration (with diagonal scaling) as a function of the virtual processor ratio. All reported times are in milli–second and correspond to single precision floating point operations. The finite element meshes used in this analysis comprised of trilinear brick elements.

and 400 elements in the width dimension. Figure (6) shows the evolution of the magnitude of the normalized global residual during the iteration process.

5 Conclusions

A data parallel implementation of the finite element method with two different mapping schemes is described using the Connection Machine system, CM–2 as the model architecture. The first mapping between the processors of the data parallel architecture and the logical units of data on which all the processors operate concurrently takes advantage of nearest neighbor communication when the processors are configured as a lattice. This mapping ensures uniform processor utilization and an efficient utilization of the storage. The mapping described here works very well for meshes comprising of brick elements. A more complex mapping is necessary for more meshes composed of different types of elements and for domains with arbitrary geometries. This mapping views the processors of the data parallel architecture as two sets. Again, the processor utilization and the storage requirements per processor are uniform. Communication between these two sets is required for for the solution phase of the algorithm. Preliminary timing analysis show that a data parallel architecture is extremely efficient exploiting the inherent parallelism in scientific applications, such as the finite element method.

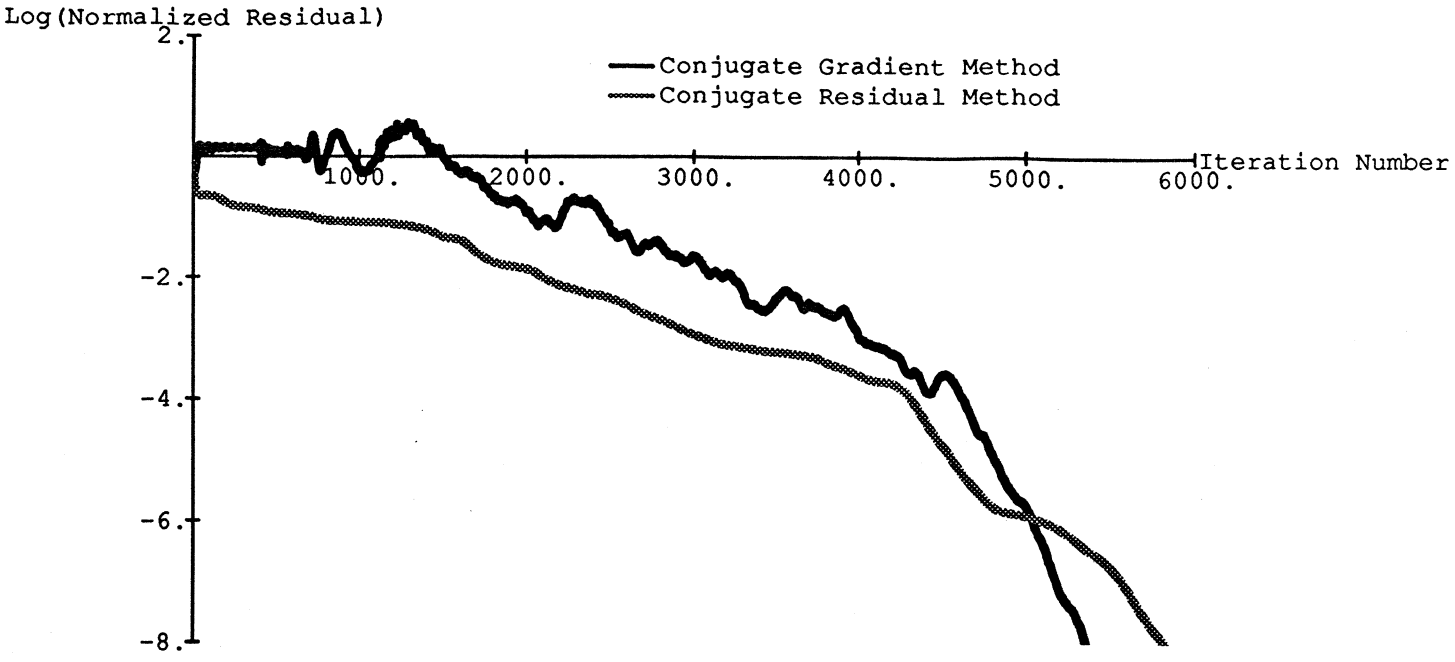

Figure 5: The evolution of the magnitude of the normalized global residual (log, base 10) during the conjugate gradient iteration process. The finite element mesh for this simulation was composed of trilinear brick elements. The mesh had ten elements in the length direction, one element in the thickness direction and 400 elements in the width direction.

6 References

1. Carey, G. F. [1986], "Parallelism in Finite Element Modeling", *Communications in Applied Numerical Methods*, Vol. 2, 281–287.

2. Carey, G. F., Barragy, E., McLay R. and Sharma M. [1988], "Element–by–element vector and Parallel Computations", *Communications in Applied Numerical Methods*, Vol. 4, 299–307.

3. Farhat C. and Wilson, E. [1987], "A New Finite Element Concurrent Computer Program Architecture", *International Journal for Numerical Methods in Engineering* Vol. 24, 1771–1792.

4. Hillis, D. W. [1985], *The Connection Machine*, MIT Press, Cambridge, MA.

5. Johnsson, S. L. and Ho, C. T. [1989], "Spanning Graphs for Optimum Broadcasting and Personalized Communication in Hypercubes", *IEEE Trans. Computers.* Also Technical Report YALEU/DCS/RR–610.

6. Johnsson, S. L. and Mathur, K. K. [1989], "Data Structures and Algorithms for the Finite Element Method on a Data Parallel Supercomputer", To be published

in *International Journal for Numerical Methods in Engineering.* Also, Thinking Machines Corporation Technical Report CS 89–1.

7. Law, K. H. [1986], "A Parallel Finite Element Solution Method", *Computers and Structures*, Vol. 23, No. 6, 845–858.

8. Mathur, K. K. and Johnsson, S. L. [1989], "The Finite Element Method on a Data Parallel Computing System", To be published in *International Journal of High–Speed Computing.* Also, Thinking Machines Corporation Technical Report CS 89–2.

9. Connection Machine Model CM–2 Technical Summary, Thinking Machines Corporation, Version 5.1, May 1989.