Data Structures and Algorithms for the Finite Element Method on a Data Parallel Supercomputer

S. Lennart Johnsson and Kapil Mathur

YALEU/DCS/TR-743
September 1989

To appear in the International Journal of Numerical Methods in Engineering.

Data Structures and Algorithms for the Finite Element Method on a Data Parallel Supercomputer

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

Abstract

This article describes the formulation and implementation of the finite element method on a data parallel computing system, such as the Connection Machine® system. Data structures, storage requirements, communication and parallel arithmetic complexity are analyzed in detail for the cases when a processor is assigned to a finite element, and when a processor is assigned to a nodal point per element. Data parallel algorithms for grid generation, evaluation of the elemental stiffness matrices, and for the iterative solution of the linear system are presented. An algorithm for computing the elemental stiffness matrices concurrently, as well as computing the matrix elements of a single elemental stiffness matrix concurrently without communication is presented. A conjugate gradient solver with diagonal pre–conditioner is used for the solution of the linear system. Results from an implementation of the finite element method in three dimensions based on iso–parametric brick elements are also presented. For single–precision floating–point operations the measured peak performance is in the range $1.1 - 1.8$ Gflops s^{-1} for evaluating the elemental stiffness matrices and $0.5 - 0.7$ Gflops s^{-1} for the conjugate gradient solver. The time per conjugate gradient iteration for an application with $\sim 400,000$ degrees of freedom is approximately 1.25 s for double–precision (software) floating–point operations. With hardware support for double–precision floating–point operations, the time per conjugate gradient iteration for a finite element with with $\sim 400,000$ degrees of freedom is projected to be ~ 0.15 s.

1 Introduction

The finite element method is frequently used for solving boundary and initial value problems that arise in stress analysis in computational solid mechanics. The finite element method is also used in the analysis of flow of viscous fluids and steady–state field problems such as heat conduction, lubrication of bearings, seepage through porous media, and electro–magnetic field problems. Data sets associated with scientific and engineering simulations are often very large. Consequently, there is a need for supercomputing.

[1] Also, Departments of Computer Science and Electrical Engineering, Yale University, New Haven, CT 06520

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 increased performance can no longer be achieved by improvements of currently known technologies. The Connection Machine system offers supercomputer performance, and has many characteristics that are expected in supercomputers with a performance orders of magnitude higher than what is currently available. Thousands to tens of thousands of operations can be performed concurrently. Architectures of this kind are often referred to as *data parallel* to emphasize the massive parallelism, and distinguish the architectures from *control parallel* 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. It is necessary to re–evaluate the choice of data structures and algorithms for data parallel computers. Important issues are load balance and data motion across processors. In the data parallel formulation of the finite element method presented in this article, the total storage requirement as well as storage requirement per processor, processor utilization, computational and data motion requirements, uniformity of operations across the data structure, higher level programming primitives, and programming complexity are considered.

Two model applications with a domain that can be mapped into a parallelepiped are presented. The domain is subdivided into finite elements in the form of bricks, all of the same order. The model problems are suitable for investigating:

- the consequences of different choices of elementary objects and their representation in the data parallel model of computation,
- algorithms for the computation of the elemental stiffness matrices given the different choices of elementary objects and their representation, and
- different techniques for solving the equilibrium equations,

without the added complexity of handling elements of different types and order, and domains of arbitrary shapes. The first application is a square cantilever plate fixed at one end and with a distributed load at the other end. The plate geometry is modeled by a finite element mesh that has one element in the thickness direction, 10 elements in the length direction and 400 elements in the width direction. This geometry and boundary conditions are similar to one of the applications reported by Wagner and Swanson [27]. The second application fully utilizes the Connection Machine storage on a system with 32K physical processors.

The computations in applying the finite element method consists of three distinct phases:

1. Decompose the physical domain into subdomains in the form of finite elements, each represented by a number of nodal points.

2. Evaluate a set of local interaction equations for each finite element. There is a set of equations associated with every node in each finite element. This phase is local to a finite element.

3. Compute the desired field as a function of the governing equations and the applied boundary conditions.

The equations corresponding to a nodal point are coupled to the equations for all other nodes on the same element, and the set of nodes belonging to neighboring elements. In the second step, the coupling to nodes in the same element is accounted for by computing *elemental stiffness matrices*. The coupling to the nodes on adjacent elements is resolved in the third phase. This coupling is often achieved through an *assembly* of the elemental stiffness matrices into a *global stiffness matrix*. This matrix is sparse and often banded, whereas the elemental stiffness matrices are dense.

Since data parallel computers with supercomputer performance have become available fairly recently, the next section of this article is devoted to a discussion of the essential characteristics of the data parallel programming model, and its implementation on the Connection Machine system. The mathematical formulation to model the response of a body in the presence of external loads and position constraints is outlined in section 3, and the finite element implementation is briefly discussed in section 4. A detailed discussion on the data structure relevant to the data parallel programming environment, an analysis of the storage requirements, parallel arithmetic complexity, and communication complexity is given in section 5. This includes a discussion on a data parallel, three dimensional grid generation algorithm based on the work by Steger and Sorenson [25]. Two different approaches for the concurrent generation of the elemental stiffness matrices are described, and issues related to the solution of the equilibrium equations are discussed in detail. Section 6 analyzes the performance of the data parallel finite element implementation for the two sample applications.

2 The data parallel programming model

2.1 Overview

In a data parallel programming model, algorithms are designed based on the structure and representation of the problem domain. An essential characteristic of data parallel algorithms is the choice of elementary objects. These elementary objects are subject to the same transformations (at least most of the time) 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. For the finite element method, the physical domain is discretized by a set of finite elements. In the simplest case, all the elements are identical in shape and order of approximation. In two dimensions, triangular and rectangular elements are most frequently used to construct a finite element discretization of the geometry. For three dimensional geometries, brick, prism, and pyramid elements are commonly used. In this article, only brick elements are considered.

If finite elements are chosen as the elementary objects then the evaluation of the local interactions of any finite element involves the same (or similar) sequence of operations being applied to all the elements in the mesh. Each operation in the sequence can then be performed concurrently for all the finite elements in the mesh. However, the series of operations required to compute the local interaction of one finite element may themselves be quite computationally intensive, especially for higher order three–dimensional finite elements. Therefore, it may be more convenient to extend the degree of concurrency further by choosing the elementary object as a node within a finite element. With a nodal point of a finite element as the elementary object, the sequence of operations required to compute the local interaction of a finite element can themselves be performed concurrently. For three–dimensional elements, the degree of concurrency increases by a factor proportional to the third power of the order of the element when the elementary object is a node per finite element. Thus, the degree of concurrency may be one to two orders of magnitude greater than the degree of concurrency obtained when a finite element is chosen as the elementary object.

To achieve good performance it is necessary to understand how the operations available in a data parallel environment can be implemented effectively on real architectures. This investigation uses the Connection Machine system as the model architecture.

2.2 The Connection Machine

2.2.1 Architecture

The Connection Machine system [14] model CM–2 [4], has a primary storage of 512 Mbytes (with 256 Kbit memory chips) expandable up to 2 Gbytes (with 1 Mbit chips) distributed evenly among 64K 1–bit processors. There are 16 such processors to a processor chip, and two such 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 [15,2,3]. On the Connection Machine system, the emulation of lattices with sides that are powers of two is supported by an address mode and communication primitives that use a binary–reflected Gray code

[24,21,16] for each dimension of the lattice.

A key characteristic of a high performance architecture is the ability to move data at a high rate. The primary storage on the Connection Machine has 64K ports, and a bandwidth of approximately 50 Gbytes s^{-1} at 8 MHz. The programming languages provide a lattice addressing mode, and primitives for communication in such lattices. Although the hardware allows for concurrent communication on all ports, the current release of the lattice emulation software works on one dimension of the lattice at a time. The peak data motion rate is 16 Gbytes s^{-1} for two–dimensional lattices, and 10 Gbytes s^{-1} for three–dimensional lattices. The primary reason for a decrease in the peak rate as the dimensionality of the lattice increases is because of an increase in the surface area for a given volume of data. Moreover, the fact that the current release of the emulation software works on one lattice dimension at a time also contributes to the degradation in the peak data motion rate with the dimensionality of the lattice. For arbitrary communication patterns, the CM–2 is equipped with a "router", which selects the shortest paths between the source and the destination of a message.

The Connection Machine system needs a host (front–end) computer. The Connection Machine system is mapped into the address space of the host. The host stores the program and scalar data, and executes instructions on scalar data. Instructions for data stored in the Connection Machine system are sent to a control unit that broadcasts the instructions to all processors. Currently, three families of host architectures are supported: the VAX family with the BI–bus, the Symbolics 3600 series, and the SUN–4 series by SUN microsystems. The Connection Machine system can optionally be equipped with a secondary storage system, known as the Data–Vault. There are eight I/O channels, each with a peak data transfer rate of $\sim$ 30 Mbytes s^{-1}. The storage system is expandable up to 640 Gbytes in 5 Gbyte increments.

Of particular interest for scientific visualization, is the framebuffer available with the Connection Machine system. The framebuffer is an I/O device that is used to display images computed in the Connection Machine system. User interface is provided to display any two–dimensional grid set of processors.

2.2.2 Programming languages

High–level programming languages currently available on the Connection Machine system are *Lisp, C*, and CM–Fortran. They are parallel extensions 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. Other extensions that are very useful in many instances are the "reduction" and "copy" operations. For example, a global summation which is extremely useful for evaluating inner products, is a single instruction. "Scans" [1] and "spreads" are some

examples of parallel prefix operations included in the extended set of operations. 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 (see examples in the next section). The programming languages for the Connection Machine system are extensions of conventional languages. This makes the debugging process and the debugging tools very similar to those available on conventional architectures.

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 addresses of the processors. The Connection Machine supports two different forms of addressing modes: *cube–addressing* and *lattice–addressing*. Cube–addressing makes use of the normal binary address mapping, whereas the lattice mode factors the address space into disjoint subspaces, one for each dimension of the lattice. The size of a subspace depends on the number of lattice nodes in that dimension. The configuration of the lattice of processors in the lattice–addressing mode is under program control. The assignment of lattice points to processors makes use of the binary–reflected Gray code to preserve adjacency in the lattice when embedded in the Connection Machine memory.

Algorithm design for data parallel architectures is often made based on a conceptual separation between the operations on the elementary objects, and the interaction between elementary objects. In scientific and engineering applications, the interaction between elementary objects has traditionally been represented as sparse matrices, which may be emulated as vectors, or lists. In a data parallel environment, pointers are often used to define the immediate relationship between objects. However, if the objects can be placed on a regular lattice then the pointers become unnecessary in a lattice addressing mode.

A matrix representation for a collection of local data values is often very useful as the elements of an elementary object. The Connection Machine programming languages supports arrays local to an elementary object. For example, in the implementation of the finite element method an elemental stiffness matrix is conceptually convenient and computationally a very useful representation.

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 Connection Machine programming model 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.

The next section introduces some essential features of the high level programming languages available on the Connection Machine system CM–2. The programming language, *Lisp, was used for the data parallel implementation of the the finite element method on the Connection Machine system. Parallel versions of C and Fortran are also available, and future scientific applications are likely to be implemented predominantly in these languages.

2.2.3 *Lisp

The programming language *Lisp is a parallel extension of Common Lisp. There is one additional data type: a parallel variable, called a *pvar*. In *Lisp, parallel variables (or *pvars*) are defined by a statement of the form

(***defvar** *pvar pvar-expression*).

The current implementation of *Lisp allocates a *pvar* across the entire configuration of the Connection Machine. The same section of the storage on every processor is assigned to a given *pvar*. One particular element of the parallel variable can be referenced by the statement

(**pref** *pvar address*),

which returns the element specified by *address* of the parallel variable *pvar*. The function

(***set** *pvar-1 pvar-2*)

assigns elements of the parallel variable *pvar-2* to the corresponding values of the elements of the parallel variable *pvar-1*. This instruction is equivalent to the BLAS–1 subroutine SCOPY [22]. Similarly, individual elements of a parallel variable can be set by using the Common Lisp function **setf** as

(**setf** (**pref** *pvar address*) *var*).

The programming language *Lisp defines parallel array variables as the parallel equivalent of Common Lisp arrays. In the programming language *Lisp, **pvar arrays** are parallel variables containing one array per processor. A variety of methods are available for allocating parallel array variables.

The two global addressing schemes available on the Connection Machine system are readily accessible from the programming language *Lisp. Addresses corresponding to

the conventional binary addressing scheme are called *cube–addresses* and addresses in multi–dimensional lattices are called *grid–addresses*. Further, references on the grid addresses can be made with either absolute lattice coordinates or relative addresses. The specific functions available in *Lisp are as follows

cube–address: (**pref** *pvar address*),

grid–address: (**pref** *pvar* (**grid** *grid–address*)),

and for concurrent access across all active processors

cube–address: (**pref!!** *pvar–expression cube–address–pvar*),

grid–address: (**pref!!** *pvar–expression* (**grid!!** *grid–address–pvar*)),

relative–grid–address: (**news!!** *pvar–expression* relative–grid–address),

where *grid–address* is the absolute lattice–address of a processor, *relative–grid–address* is the relative lattice–address for a processor. The standard operators of Common Lisp have been extended to the corresponding concurrent versions by the suffix !!. As an example of some of the concurrent operation primitives available in *Lisp, the following *Lisp statement implements a Jacobi iteration for the five–point stencil

```
(*set new–est
      (*!!
          (!! 0.25)
          (+!!
              (news!! current–est -1 0)
              (news!! current–est 0 -1)
              (news!! current–est 0 1)
              (news!! current–est 1 0)
              right–hand–side
          )
      )
).
```

The operation ***set** is a local memory movement in all active processors. The corresponding operation used for inter–processor communication is ***pset**, which like **pref** has several forms, depending on the addressing scheme being used by the application program.

In general, concurrent operations available in *Lisp are performed on active processors only. Conditional statements are available in *Lisp to change the state of one or

more processors. Some examples of the conditional statements are ***all**, ***when**, ***if**, and ***cond**. *Lisp also provides very useful global operators. Of particular interest for numeric applications are ***min**, ***max**, ***sum**, and **scan!!**. The ***min**, ***max**, and ***sum** operators compute the minimum, maximum, and the sum of the values for a parallel variable in the currently selected set of processors. An example of the "scan" function is

(**scan!!** *pvar function* :**direction** *segment-pvar* :**include-self**),

where the *segment-pvar* divides the address space into non-overlapping segments. A segment consists of a sequence of processors in ascending cube-address order. A new segment of processors begins at each processor in which *segment-pvar* is true. Even if all *segment-pvar* components are **nil**, however, there is always at least one segment beginning with the selected processor having the lowest cube address. The **scan!!** operation is then concurrently applied to all segments. The **scan!!** argument *function* is one of the following associative binary *Lisp functions : +, **and**, **or**, **logand**, **logior**, **max**, and **min**. In addition, the **copy** function is also supported as a scanning function, even though there is no such *Lisp function. As an example, if "+" were the *function* every *pvar* location will contain the sum of the *pvar* elements in processors with lower addresses in its segment (if the **direction** is *increasing*), including it's own original *pvar* value, if **include-self** is true.

2.2.4 C*

The programming language C* is an extension of C with a strong design influence from C++. In the programming language C*, objects that are of the same nature are members of the same **domain**. Conceptually a **domain** is similar to a **class** in C++. In the data parallel model of computation, a processor is associated with every instance of a **domain**. Every member of a **domain** has the same storage layout. Referencing a **domain** implies a selection of processors. Only processors associated with an instance of the referenced domain remain active. There are two new data types: **mono** and **poly**. Data of type **poly** are allocated on the Connection Machine system. Data belonging to any **domain** is by default of type **poly**. Data that is not of type **poly** is of type **mono**, and resides in the storage of the host machine.

There are only very few new operators in C*. Most C operators are extended to C* by the distinction between **mono** and **poly**. Communication between the host and the Connection Machine is implicit. For example, broadcasting from the host occurs if a **mono** value is assigned to a **poly** variable. Similarly, a reduction operation is performed if the combined result of the elements of a **poly** variable is desired. The result of the reduction is a **mono** value in the host. Interaction between the elements

of a single or several **poly** variables results in several communication patterns on the Connection Machine system.

2.2.5 CM–Fortran

The programming language CM–Fortran that is currently available on the Connection Machine is similar to the proposed Fortran–8x standard [23]. This standard has array constructs and operations on such constructs. The array extensions of the Fortran–8x standard, available on CM–Fortran, permit application programs to exploit the data parallel environment. The CM–Fortran compiler generates code that makes direct calls to the parallel instruction set of the Connection Machine system. The array variables are allocated either on the Connection Machine system or on the host depending on their use within an application program. A given array is allocated on the Connection Machine system whenever the array is used in one or more Fortran 8x array operations within the application program. Compiler directives are available to allow one or more axes of a CM–based array to be stored on a single processor.

A typical applications program written in CM–Fortran contains several different kinds of source constructs that are translated by the compiler into code that executes on the host and operates on data residing on both the Connection Machine system and the host. Statements that affect the flow of execution control within an application program (for example the **IF** statement) always translate to code that is executed entirely on the host. Operations that affect only scalar data are also executed on the host, and scalar data is always allocated on the host. An array operation may execute on the Connection Machine hardware depending on where the array data on which it operates is allocated. As an example of a data parallel operation in CM–Fortran, the five–point stencil for the Jacobi–iteration expressed above in *Lisp is now expressed in CM–Fortran.

$$new - est = \tfrac{1}{4}[\mathbf{cshift}\,(A,1,-1) + \mathbf{cshift}\,(A,1,+1) + \mathbf{cshift}\,(A,2,-1) + \mathbf{cshift}\,(A,2,+1) + rhs]$$

The compiler translates the above CM–Fortran statement to generate calls to several nearest neighbor communication primitives. The primary intent of the above example is to illustrate the fact that the two loop levels typically seen in traditional Fortran programs have vanished. Not only is the programming more elegant, it also follows the logical operations that are typically used in the conceptual design stage of an algorithm, and is easier to debug.

3 Mathematical background

The deformation of a solid in response to external loads and constraints is governed by the balance laws of continuum mechanics. The mathematical formulation described below consists of four sections: kinematics, the balance laws of continuum mechanics, the constitutive equations, and the boundary conditions. Details can be found in [9,12].

3.1 Kinematic relationships

Kinematics provides a set of equations which relate deformation quantities such as strain to the displacements of the material at any arbitrary point within the body. Assuming infinitesimal strains, the *strain tensor*, ϵ, is defined as

$$\epsilon = \text{Symm}(\nabla \mathbf{u}), \tag{1}$$

where $\mathbf{u}$ is the displacement vector.

3.2 Balance laws

In the absence of body forces and neglecting inertia, the balance of linear momentum states that

$$\nabla \cdot \sigma = 0, \tag{2}$$

where σ is the Cauchy *stress tensor*. The balance of angular momentum requires that the stress tensor σ be symmetric.

3.3 Constitutive equations

Kinematics and balance laws do not provide enough mathematical relations to solve the boundary value problem. Additional relationships are necessary to close the system of equations. These equations come from constitutive laws which describe the material response. The constitutive laws predict the behavior of the material in response to external loads. Assuming isotropic material behavior, the stress required to achieve a certain state of strain is given by the generalized Hookes law

$$\sigma = 2\mu\epsilon + \lambda \text{Tr}(\epsilon)\mathbf{I}, \tag{3}$$

where μ and λ are material parameters frequently known as the Lamé constants. For linearly elastic materials, μ and λ are constants.

3.4 Boundary conditions

To solve the boundary value problem described by the system of equations above, boundary conditions must be specified over the surface of the physical domain. In general

$$\mathbf{u} = \bar{\mathbf{u}} \text{ on } S_u \tag{4}$$

and

$$\sigma \cdot \mathbf{n} = \bar{\mathbf{T}} \text{ on } S_\sigma, \tag{5}$$

where S_u is the portion of the surface of the body with displacement boundary conditions ($\bar{\mathbf{u}}$), S_σ is the portion of the surface with traction boundary conditions ($\bar{\mathbf{T}}$), and $\mathbf{n}$ is the surface normal at the point where the traction vector, $\bar{\mathbf{T}}$ is applied. For the resulting boundary value problem to be well–posed, the surface, S, of the body must be decomposed such that

$$S_u \cup S_\sigma = S \tag{6}$$

and

$$S_u \cap S_\sigma = \emptyset. \tag{7}$$

4 Numerical formulation

To solve the mathematical model described above numerically, it is necessary to reduce the continuous (infinite) physical domain to a discrete (finite) computational domain. This discretization can be accomplished by a variety of different methods, such as

- a finite difference method,
- a finite volume method,
- a finite element method,
- a boundary element method, or
- a spectral method.

The finite element method is the most popular approach used in structural and solid mechanics. For areas of application where a variational principle is known (as in stress analysis described in this article) the finite element method may be regarded as an approximation technique that develops from the known variational principle. However, in general, for most applications, only governing differential equations and the applied boundary conditions are known. Therefore, it is more illustrative to regard the finite element method as an approximation which results from a weighted residual method.

The finite element approximation of the system of equations described in the previous section is obtained from the variational principle which represents the statement of virtual work as [31]

$$0 = -\int_V \delta \mathrm{Tr}(\epsilon\sigma) dV + \int_{S_\sigma} \delta \mathbf{u}^T \mathbf{T} dS, \tag{8}$$

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 $\mathbf{T}$. By expressing the symmetric tensors, σ and ϵ as vectors of length six so that

$$\{\sigma\} = \{\sigma_{11}\ \sigma_{22}\ \sigma_{33}\ \sigma_{21}\ \sigma_{31}\ \sigma_{32}\}^T \tag{9}$$

and

$$\{\epsilon\} = \{\epsilon_{11}\ \epsilon_{22}\ \epsilon_{33}\ 2\epsilon_{21}\ 2\epsilon_{31}\ 2\epsilon_{32}\}^T \tag{10}$$

respectively, the above functional may be restated in vector notation as

$$0 = -\int_V \delta\{\epsilon\}^T \{\sigma\} dV + \int_{S_\sigma} \delta\{U\}^T \{T\} dS, \tag{11}$$

where $\{U\}$ is a vector containing the components of the *displacement field* $\mathbf{u}$.

The displacement field, $\{U\}$, and the corresponding *strain field*, $\{\epsilon\}$, at any arbitrary point within the body are approximated from

$$\{u\} = [N]\{U\}, \tag{12}$$

and

$$\{\epsilon\} = [N']\{U\}, \tag{13}$$

respectively. The matrix $[N]$ comprises of a set of *interpolation* or *shape functions* and $[N']$ is a matrix containing the derivatives of these interpolation functions. The stress tensor in the variational statement (Equation (11)) may be replaced by a function of the strain tensor, by using the constitutive equations expressed in vector–notation as

$$\{\sigma\} = [C]\{\epsilon\}, \tag{14}$$

where $[C]$ is the *constitutive matrix* containing the Lamé constants μ and λ. For the forms of $\{\sigma\}$ and $\{\epsilon\}$ defined by Equations (9) and (10) the resulting constitutive matrix $[C]$ is symmetric. The final system of equations that results from the above interpolations is of the form

$$[K]\{U\} = \{F\}, \tag{15}$$

where the *stiffness matrix* $[K]$ is defined as

$$[K] = \int_V [N']^T [C][N'] dV, \tag{16}$$

and the force vector $\{F\}$ may be expressed as

$$\{F\} = \int_{S_\sigma} [N]^T \{T\} dS. \tag{17}$$

The stiffness matrix is obtained through the *assembly* of the *elemental stiffness matrices*, $[K^{(\mathrm{el})}]$, where

$$\left[K^{(\mathrm{el})}\right] = \int_{V_{\mathrm{el}}} [N']^T [C][N'] dV, \tag{18}$$

and

$$[K] = \sum_{\mathrm{el}} \left[K^{(\mathrm{el})}\right]. \tag{19}$$

For the elemental stiffness matrix computation the interpolation functions are local over the domain of a single finite element, and the order of the matrix of interpolation functions, $[N]$, is $n \times u$, where n is the number of nodes on the finite element, and u is the number of degrees of freedom per node.

5 A data parallel implementation

Some of the important design issues that arise for the implementation of the finite element method in a data parallel environment are

- parallel grid generation for transforming the physical domain to a discretized computational domain,
- a data structure for representing this computational domain,
- generation of the elemental stiffness matrices concurrently, and
- concurrent solution of the system of linear equations.

For each of these issues, several factors need to be considered. Some of the more important issues are as follows

- storage requirements,
- communication complexity,
- parallel arithmetic complexity,
- uniformity of computations, and
- programming complexity.

5.1 Parallel grid generation

For the model problem elliptic partial differential equations can be used to generate a smooth grid that permits a one–to–one mapping so that the mesh lines do not cross. In this article, all finite element meshes have been generated by the solution of Laplace equations [26]:

$$\begin{aligned} \nabla^2 \xi &= 0 \\ \nabla^2 \eta &= 0, \\ \nabla^2 \zeta &= 0 \end{aligned} \tag{20}$$

where $\{\xi\ \eta\ \zeta\}^T$ are the coordinates of the nodal points in the regular *computational domain.* The global coordinates $\left(\{x\ y\ z\ \}^T\right)$ of the nodal points in the physical domain are evaluated by first transforming the above governing equations (Equations (20)) to the global coordinate space. The resulting transformed equations, which are coupled and non–linear, are of the form:

$$\{a_1\ a_2\ a_3\ a_4\ a_5\ a_6\}^T \{x_{\xi\xi}\ x_{\eta\eta}\ x_{\zeta\zeta}\ x_{\xi\eta}\ x_{\xi\zeta}\ x_{\eta\zeta}\} = 0, \tag{21}$$

where the coefficients $a_1 \cdots a_6$ are functions of the local derivatives contained in the Jacobian matrix

$$[J] = \begin{bmatrix} x_\xi & y_\xi & z_\xi \\ x_\eta & y_\eta & z_\eta \\ x_\zeta & y_\zeta & z_\zeta \end{bmatrix} \tag{22}$$

The transformed equations are solved together with the boundary information to evaluate the global coordinates of the nodal points. The transformation in two–dimensions are given in [25].

The implementation of the above algorithm on the Connection Machine system CM–2 uses the grid addressing mode. The computational domain is first mapped on to the three–dimensional lattice of processors, assuming that each processor represents one nodal point of the computational domain. Equations of the form given by Equation (21) are solved to evaluate the global coordinates (x, y, and z) of the nodal points by Jacobi's method. A second–order interpolation for the partial derivatives yields a nineteen point stencil. This scheme is inherently parallelizable, and requires communication between lattice points that differ by one in at most two of the indices, as seen by the discretized form of Equation (21)

$$\begin{aligned} a_1 [X_{i+1jk} - 2X_{ijk} + X_{i-1jk}] + \\ a_2 [X_{ij+1k} - 2X_{ijk} + X_{ij-1k}] + \\ a_3 [X_{ijk+1} - 2X_{ijk} + X_{ijk-1}] + \\ \tfrac{a_4}{4} [X_{i+1j+1k} + X_{i-1j-1k} - X_{i-1j+1k} - X_{i+1j-1k}] + \\ \tfrac{a_5}{4} [X_{i+1jk+1} + X_{i-1jk-1} - X_{i-1jk+1} - X_{i+1jk-1}] + \\ \tfrac{a_6}{4} [X_{ij+1k+1} + X_{ij-1k-1} - X_{ij-1k+1} - X_{ij+1k-1}] = 0. \end{aligned} \tag{23}$$

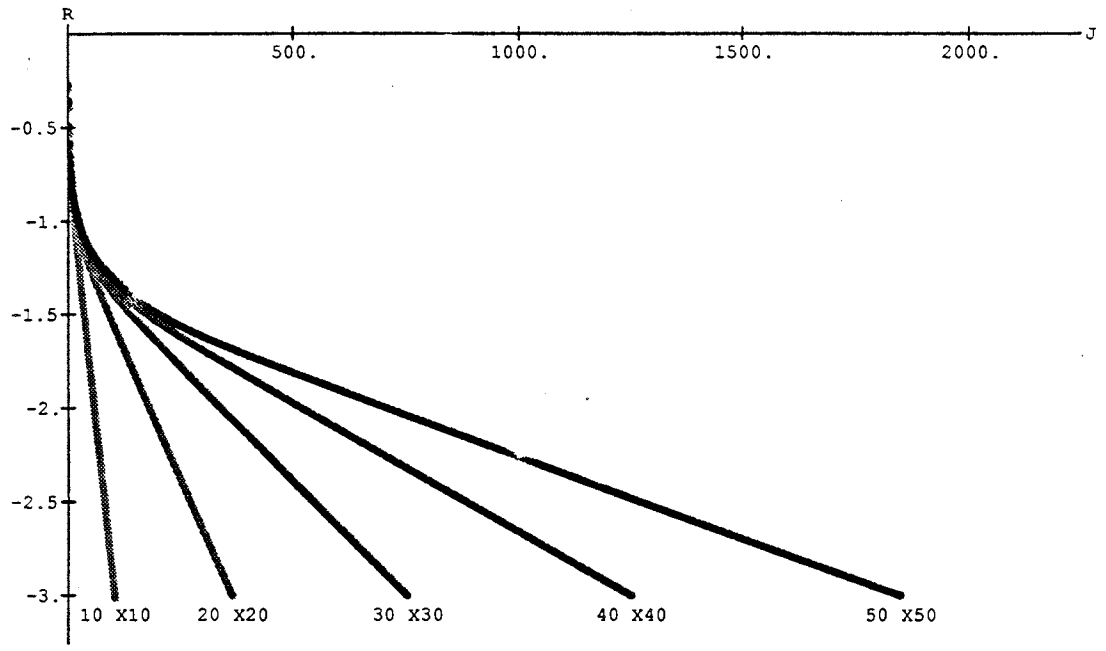

Figure 1: Normalized ℓ_2-norm of the residual for the Jacobi method for square grids and random initial state; J is the Jacobi iteration number and R is the logarithm (base 10) of the normalized global residual.

Though the Jacobi iterative scheme has a poor rate of convergence the number of iterations required for the physical domains of the two applications reported in this article are small because of very good initial guesses. The domains for the two applications reported in this article are regular and the normalized residual obtained after the first iteration was sufficiently small ($\leq 1.0 \times 10^{-4}$). The speed–up in the data level implementation of the Jacobi iteration process is proportional to the number of processors, up to the number of processors being equal to the number of lattice points. For a general case, where a good initial guess is not readily known, a numerical algorithm with a convergence rate better than the Jacobi method may be required. The convergence rate of the Jacobi iteration process is proportional to the number of grid points [30] as shown in Figures (1) and (2) where a *random initial guess* was used. The CPU time per Jacobi iteration, at a virtual processor ratio of one, is $\sim$ 0.0135 s for two–dimensional meshes and $\sim$ 0.075 s for three–dimensional meshes.

Figure 2: The number of Jacobi iterations for convergence (normalized ℓ_2-norm less than $\leq 1.0 \times 10^{-3}$); N is the total number of grid points in the finite element mesh and J is the number of Jacobi iterations required for convergence.

5.2 Data structure

Computations for all elementary objects can be performed concurrently on the Connection Machine system. For the two applications described later, it is natural to chose the lattice addressing mode. There are two possible choices of elementary objects:

- one processor of the Connection Machine 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.

With a processor assigned to a nodal point, the processor performs all computations associated with that nodal point, both in the computation of the elemental stiffness matrix and during the solution phase. In assembled form some nodal points are shared by more elements than others for higher order elements. Consequently, some nodal points have to perform more computation than others. By having a processor represent one nodal point per finite element, the computational effort required for all nodal points is identical, regardless of where on the element the node is located, and regardless of whether the element is an interior element or on the boundary. The only distinguishing feature is the element order. With a processor per node per finite element 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 (section 5.5).

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 obvious. In the case of elementary objects in the form of one nodal point per finite element a suitable data structure may be obtained by a *split* lattice, i.e., each surface (line in two–dimensions) representing a boundary between two brick (rectangular) elements is duplicated.

The two possible schemes of choosing an elementary object discussed above are compared in detail on the basis of storage requirements, parallel arithmetic complexity, and communication complexity for both Lagrange and Serendipity elements. In addition, for direct solvers, an explicit assembly of the global stiffness matrix is required. However, when an iterative solver is used, the global stiffness matrix need not be explicitly formed. Therefore, the assembled and unassembled cases have been analyzed separately.

5.3 Storage requirements

5.3.1 Lagrange elements

One processor per finite element: The unassembled elemental stiffness matrix for an element is stored on the processor representing the element. The elemental stiffness matrix stored on the processor has $(p+1)^2$ rows and columns in two dimensions and $(p+1)^3$ rows and columns in three dimensions, where p is the order of the finite element ($p = 1$ for linear elements, $p = 2$ for quadratic elements, and so on). Symmetry of the stiffness matrix is easily exploited because the entire elemental stiffness matrix resides locally on the processor. The storage and computational requirements are uniform among all processors when the mesh is composed of similar finite elements.

If the stiffness matrix is assembled most processors store the rows (or columns) corresponding to p^2 assembled nodes in two dimensions and p^3 assembled rows in three dimensions. However, processors representing elements on the boundary of the physical domain need to store $(p+1)^2$ and $(p+1)^3$ rows (or columns), respectively. Processors representing finite elements that are not on the boundary are assigned one corner node, $2(p-1)$ nodes along the two edges and $(p-1)^2$ interior nodes in two dimensions. For a three–dimensional finite element mesh, this assignment is one corner node, $3(p-1)$ nodes along three edges, $3(p-1)^2$ nodes on three faces, and $(p-1)^3$ interior nodes. The assembled stiffness matrix may be stored in either a *dense* representation or a *sparse* representation for each element. The non-uniform data allocation reduces the storage utilization by a factor of $\sim (\frac{p}{p+1})^4$ in two–dimensions and by a factor of $\sim (\frac{p}{p+1})^6$ in three–dimensions, assuming that most elements in the lattice are interior elements. An

implementation with some redundant operations can make the computations uniform over all processors.

One processor per nodal point: The storage of the elemental stiffness matrix is shared equally by n processors, where n is the number of nodal points per element ($n = (p+1)^2$ in two dimensions and $n = (p+1)^3$ in three dimensions). Without assembly of the global stiffness matrix each processor stores the u ($u = 2$ in two dimensions and $u = 3$ in three dimensions) unassembled rows (columns) corresponding to the nodal point represented by the processor. The summation required in computing the product of the stiffness matrix with the displacement vector requires no communication. However, the displacement vector has to be accumulated by the communication of the nodal displacements from all processors that make up the finite element. Every processor broadcasts its displacement values to every other processor on the element (*all–to–all broadcasting* [20]). In contrast, when the columns of the stiffness matrix corresponding to a nodal point are stored on the processor representing the nodal point, only partial inner products can be computed concurrently. An all–to–all reduction among the processors representing the nodal points on an element is required to accumulate the partial inner products.

When the elemental stiffness matrices have to be assembled into a global stiffness matrix, two storage assignment schemes are possible. Either a row (column) of the assembled stiffness matrix may be stored per processor, or each row (column) of the assembled matrix may be shared among the processors from which it is assembled. For the latter case, in two dimensions, the sharing is between four processors for a corner node, two processors for an edge node. Nodes lying in the interior of a finite element require no sharing. Similarly, in three dimensions, the sharing is made between eight processors for a corner node, four processors for edge nodes, two processors for face nodes, and again no sharing is needed for interior nodes.

If the rows (columns) of the assembled stiffness matrix corresponding to a nodal point are *not* shared among the processors over which the assembly is performed then the effective processor utilization decreases, with an increase in the the maximum processor load. The fraction of the total number of processors being utilized after assembly is $\sim (\frac{p}{p+1})^2$ in two dimensions and $\sim (\frac{p}{p+1})^3$ in three dimensions. In three dimensions the number of active processors is about 50% for $p = 5$, but only 12.5% for first order elements. Moreover, the number of matrix elements for a processor representing a corner node is $(2p + 1)^3$ as compared with $(p + 1)^3$ matrix elements for a processor representing an interior node. This ratio is approximately 6 for $p = 5$. Hence, when the elementary object represents one processor per node per element, assembling the stiffness matrix and assigning its rows (columns) corresponding to the nodal point to *one* of the processors from which the assembly was performed results in a very poor processor utilization. Sharing of an assembled row (column) among the processors from

which it is assembled results in a significant decrease in the computational efficiency.

Comparison of storage and arithmetic requirements. The unassembled stiffness matrix requires the same amount of overall storage for the two possible choices of the elementary object, but it is easy to exploit symmetry of the elemental stiffness matrix with a processor representing a finite element.

The savings in the *used storage* due to assembly is the same for the two choices of elementary objects. However, with the current storage allocation scheme of the Connection Machine system, no savings of *allocated storage* is realized when finite elements are chosen as the elementary objects because of the storage requirements of the processors representing elements on the boundary. When a nodal point of a finite element is the elementary object, the allocated storage actually increases after the assembly process, if the assembled rows (columns) are not shared. Sharing therefore becomes necessary for an efficient storage utilization.

Representing the element–wise assembled stiffness matrices as dense matrices simplifies the local data structure significantly, but as the order of the finite elements increases, the density of this matrix decreases appreciably. In two dimensions, the density is 100% for $p = 1$ and $\sim 25\%$ for $p = 5$, and in three dimensions the corresponding densities of the stiffness matrices are 100% and $\sim 15\%$ respectively.

The assembly process results in nonuniformity of the data structures. In the case of one nodal point per processor, processors representing internal nodes have storage requirements that are different from all other processors. There are also different storage requirements for processors representing nodes on the faces, edges, and corners of an element. Moreover, the storage requirements also depend on the location of the nodal point in the physical domain. With one processor per finite element, only a distinction between internal elements and elements on the boundary is required. To realize any storage and computational advantages, when the elementary object is a nodal point per finite element, the programming complexity of the assembly process is considerably increased. When the elementary object is a finite element, the added programming complexity is significantly less. The data parallel implementation of the finite element method discussed later does not assemble the elemental stiffness matrices.

The total number of arithmetic operations is the same regardless of whether a processor represents a finite element, or a nodal point per element. However, the degree of concurrency in the latter case is $(p+1)^u$ times greater than the degree of concurrency in the former case, and the solution time correspondingly smaller with a sufficiently large number of physical processors (so that the virtual processor ratio is one).

Table (1) summarizes the storage requirements and effective processor utilization for Lagrange elements in two and three dimensions. No symmetry assumptions have been

		Two dimensions	Three dimensions
Order		p	p
Degrees of freedom per nodal point, u		2	3
Number of nodes per element, n		$(p+1)^2$	$(p+1)^3$
Processor per node	Nodal points per processor	1	1
	Unassembled stiffness matrix	$nu \times u$	$nu \times u$
	Assembled stiffness matrix		
	Corner node	$(2p+1)^2u^2$	$(2p+1)^3u^2$
	Side node ($p>1$)	$(2p+1)(p+1)u^2$	$(2p+1)^2(p+1)u^2$
	Interior node ($p>1$)	nu^2	nu^2
	Face node ($p>1$)		$(2p+1)(p+1)^2u^2$
	Average[a] ($p>1$)	$(p^2+4p+4)u^2$	$[8m^3+q^3+12qm^2+6q^2m]u^2$
	Processor spatial utilization[b]	$\sim(\frac{p}{p+1})^2$	$\sim(\frac{p}{p+1})^3$
	Processor temporal utilization	$\sim(\frac{p+1}{2p+1})^2$	$\sim(\frac{p+1}{2p+1})^3$
Processor per element	Nodal points per processor	n	n
	Unassembled stiffness matrix	$nu \times nu$	$nu \times nu$
	Assembled stiffness matrix		
	Sparse representation	$(n-1)^2u^2$	$p^3u^2[8m^3+q^3+12qm^2+6q^2m]$
	Dense representation (max)	$(2p+1)^2u \times (p+1)^2u$	$(2p+1)^3u \times (p+1)^3u$
	Average per processor[c]	$(2p+1)^2u^2$	$(2p+1)^3u$
	Processor spatial utilization	1	1
	Processor temporal utilization	$\sim(\frac{p}{p+1})^2$	$\sim(\frac{p}{p+1})^3$

Table 1: Storage requirements and processor utilization for Lagrange finite elements in two and three dimensions.

[a] $q = p-1$ and $m = \frac{2p+1}{p+1}$.

[b] When assembled rows are not shared.

[c] In dense representation.

made for the elemental stiffness matrices.

5.3.2 Serendipity elements

Two dimensional Serendipity elements of order p have $4p$ nodal points on the edges. In addition, a interior nodes are required in order that all the terms of a complete p–th order expansion are available. The number of interior nodes necessary are summarized in Table (2). For three–dimensional Serendipity elements there are $4(3p-1)$ nodal points on the edges, 8 of which are corner nodes. This yields a total of $4(3p-1)+a$ nodal points per finite element.

One processor per element: The number of matrix elements in the elemental stiffness matrices is $nu \times nu$, where n is the number of nodal points per finite element ($n = 4p + a$ in two–dimensions and $4(3p-1)+a$ in three–dimensions) and u is the number of degrees of freedom per nodal point. The number of matrix elements can be reduced to $\frac{1}{2}nu \times (nu+1)$ after accounting for symmetry. When these elemental stiffness matrices are assembled into a global stiffness matrix, rows (columns) associated with nodes shared between elements can be assigned to processors using the scheme described for Lagrange elements. The assembled stiffness matrix is assigned to processors by row partitioning. In the row partitioned representation, the multiplication of the stiffness matrix by a vector[2] requires communication to assemble the vector, but no communication is needed for a row summation. Processors representing elements lying in the interior of the physical domain store $3p-2+a$ rows (columns). A subset of processors representing elements lying on the boundary of the physical domain store $4(3p-1)+a$ rows (columns). For higher order elements, the density of the sub–matrix of the global stiffness matrix, stored on every processor is approximately 60%.

One processor per nodal point per finite element: Each processor stores the rows corresponding to the nodal point represented by the processor. Therefore, a matrix of size $u \times nu$ is stored on each processor.

When the stiffness matrix is assembled, the number of elements in a row (or column) depends upon the nodal point corresponding to the row. For example, a corner node couples to all nodes in four elements (a total of $12p-3+4a$ nodes) in two–dimensions and eight elements in three–dimensions (a total of $54p-27+8a$ nodes). Similarly, a node on the edge is shared by two elements in two–dimensions and four elements in three–dimensions, thus coupling with $7p-1+2a$ and $33p-15+4a$ nodes respectively. Assembling the global stiffness matrix without sharing the assembled rows (columns) yields a very nonuniform storage requirement. Sharing is essential for an effective use

[2]For example, in an iterative solver.

of the Connection Machine system, both for load balance and an efficient memory utilization.

Comparison of storage and arithmetic requirements. The storage requirements per processor is $O(n^2)$ for one finite element per processor, and $O(n)$ for one nodal point per finite element per processor. Table (2) summarizes the storage requirements for two and three dimensional Serendipity elements. As before, it is straightforward to exploit symmetry when the elementary object is a finite element and quite complex when the elementary object is a nodal point per finite element. For a nodal point per finite element per processor, there is the additional complexity of making allocated storage correspond to required storage. If a three–dimensional lattice of processors is used directly, then only $\frac{n_s}{n_l}$ processors (and consequently storage) do useful work, where n_s is the number of nodal points in a Serendipity element of order p and n_l is the number of nodal points in a Lagrange element of order p. For $p = 5$, this processor (storage) utilization is $\sim \frac{1}{3}$. A more efficient mapping of the nodes of Serendipity elements to processors is required both with respect to storage management and computational efficiency for the current storage allocation scheme, and the scheduling of virtual processors. Embeddings of the type described in [15,2,3] may be used.

Assembly leads to uneven storage requirements for both types of elementary objects. Further, for the case when the elementary object is a node per finite element, it is necessary to share the assembled rows (columns) between the processors from which they are assembled. As before, this assembly process adds significantly to programming complexity.

The degree of concurrency when a processor represents a nodal point per finite element is a factor of n greater than the degree of concurrency when a processor represents a finite element. The total number of arithmetic operations and storage requirements are approximately the same for the two choices of the elementary objects.

Table (2) summarizes the storage requirements for Serendipity elements. No symmetry assumptions are made for the elemental stiffness matrices.

5.4 Communication complexity

This section discusses the communication needs for Lagrange and Serendipity elements. The two elementary objects are analyzed in the context of an iterative solver for which the dominating communication requirements occur in the sparse matrix–vector multiplication. These communications are all short range. A direct solver requires long range (global) communication in the lattice. Some types of pre–conditioners for the conjugate gradient method make use of direct solvers, but an analysis of such pre–conditioners is beyond the scope of this investigation.

		Two dimensions	Three dimensions
Order		p	p
Degrees of freedom per node, u		2	3
Additional nodes, a, $(p > 3)$		$\frac{1}{2}(p-2)(p-3)$	$\frac{3}{2}(p-2)(p-3)$ $+\frac{1}{6}p(p-1)(p-2)$ $-3(p-3)-1$
Number of nodes per element, n		$4p+a$	$4(3p-1)+a$
Average number of nodes per element, $\bar{n}$		$2p-1+a$	$3p-2+a$
Nodal Points / lattice nodes		$\frac{n}{(p+1)^2}$	$\frac{n}{(p+1)^3}$
Processor per node	Unassembled stiffness matrix	$u \times nu$	$u \times nu$
	Assembled stiffness matrix		
	Corner node, n_c	$[3(4p-1)+4a]u^2$	$[27(2p-1)+8a]u^2$
	Side node $(p > 1)$, n_s	$[(7p-1)+2a]u^2$	$[3(11p-5)+4a]u^2$
	Interior node $(p > 3)$, n_i	au^2	au^2
	Average $(p > 1)$, n_a	$\frac{2(p-1)n_s+n_c+n_i}{\bar{n}}u^2$	$\frac{3(p-1)n_s+n_c+n_i}{\bar{n}}u^2$
Processor per element	Unassembled stiffness matrix	$nu \times nu$	$nu \times nu$
	Assembled stiffness matrix:		
	Matrix elements corresponding to		
	(a). Corner nodes, r_c	1	1
	(b). Side nodes $(p > 1)$, r_s	$(2p-2)$	$(5p-5)$
	(c). Interior nodes $(p > 3)$, r_s	n_a	n_a
	in assembled stiffness on		
	all processors.		
	Sparse representation	$n_c r_c + n_s r_s + n_i n_a$	$n_c r_c + n_s r_s + n_i n_a$
	Dense representation	$(r_c + r_s + r_i)n_c$	$(r_c + r_s + r_i)n_c$
	Average per processor[a]	n_c	n_c
	Processor spatial utilization	1	1
	Processor temporal utilization	$\sim (\frac{2p+a}{4p+a})$	$\sim (\frac{3p-1+a}{4(3p-1)+a})$

Table 2: Storage requirements and processor utilization for Serendipity finite elements in two and three dimensions.

[a] In dense representation.

	Type of communication	Two dimensions	Three dimensions
Processor per node	Intra–element	$2(n-1)u$	$2(n-1)u$
	Inter–element	$4u$	$6u$
Processor per element	Lagrange elements	$4(p+1)u$	$6(p+1)^2u$
	Serendipity element	$4(p+1)u$	$24pu$

Table 3: Element transfers per processor for brick elements in two and three dimensions.

One processor per finite element: For this elementary object only inter–element communication is required. Without assembly of the global stiffness matrix, communication is required for the assembly of the residual vector after a local sparse matrix–vector multiplication. For Serendipity elements of order p, $4(p+1)u$ and $6 \times 4pu$ communications per processor are required in two and three dimensions respectively. The corresponding numbers for Lagrange elements of order p are $4(p+1)u$ and $6(p+1)^2u$ in two and three dimensions respectively.

One processor per nodal point per finite element Both intra–element and inter–element communication are required to compute a matrix–vector product. Intra–element communication of nodal values is an *all–to–all broadcasting* [20] among the processors representing nodes on the same element. This operation can be performed concurrently within an element, as well as for different elements. Assuming that communication is performed one dimension at a time, it is easily shown that the total number of element transfers in sequence is $2 \times (n-1)u$. With bidirectional communication, these numbers reduce by a factor of 2. With concurrent communication on all ports the communication time is further reduced by the number of Boolean cube dimensions required for embedding a finite element [20].

Inter–element communication is required for assembling the residual for an iterative solver, or in the explicit assembly of the global stiffness matrix. All nodal points shared between elements participate in the inter–element communication. For this elementary object representation, the inter–element communication requires at most $4u$ and $6u$ nearest neighbor communications in two and three dimensions respectively. Table (3) gives the number of data element transfers needed for a processor in two and three dimensions.

5.4.1 Comparing Lagrange and Serendipity elements

For both Lagrange and Serendipity elements, an unassembled stiffness matrix formulation is the simplest to implement. This implementation results in no loss of performance or storage efficiency, if there are sufficiently many physical processors to allow for a vir-

	order p	virtual processor ratio	maximum deg. of freedom
Processor per node	1	8	786444
	2	4	294921
	3	1	147504
	4	1	61575
Processor per element, unsym.	1	2	393216
Processor per element, sym.	1	4	786432

Table 4: The maximum number of degrees of freedom that fits in 512 Mbytes of storage as a function of the order of three dimensional Lagrange elements.

tual processor ratio of one. A nodal point per finite element per processor allows for the use of higher order elements in the construction of the finite element mesh. Due to the limited amount of storage per processor, the current version of the Connection Machine system can only store elemental stiffness matrices for elements of order one when the elementary object is a finite element. Symmetry is difficult to exploit with one nodal point per finite element per processor, and not doing so increases the storage requirements for the stiffness matrix by a factor of ~ 2. The concurrency with one nodal point per finite element per processor is one to two orders of magnitude greater than the degree of concurrency obtained when there is a finite element per processor. In addition, the communication bandwidth required between a pair of processors is up to one order of magnitude lower for one processor per nodal point compared to one processor per element.

The data parallel implementation of the finite element method described here uses one nodal point per finite element as the elementary object. No explicit assembly of the global stiffness matrix is performed. The primary intent for this selection is to study the influence of higher order finite elements on the performance of the data parallel implementation. Lagrange elements were chosen for this investigation because of the existence of the lattice emulation capability on the Connection Machine system. A more complex data structure [15,2,3] is required to realize any benefits from using Serendipity elements. The maximum order of the elements, the number of virtual processors per physical processor for this element order, and the total number of degrees of freedom are summarized in Table 4 for three-dimensional Lagrange elements. The numbers reported in this table correspond to a primary storage of 512 Mbytes. With the use of auxiliary storage system, the number of degrees of freedom can be increased substantially.

5.5 Stiffness matrix generation

The global stiffness matrix is obtained as the sum of the elemental stiffness matrices. Contributions to a matrix element of the global stiffness matrix is obtained from all elemental stiffness matrices that share nodal points. The elemental stiffness matrix is evaluated by integrating over the volume[3] of the finite element. This integration is performed by Gauss quadrature, wherein the finite element is first mapped onto a standard element of regular shape.

The computation of an elemental stiffness matrix can be summarized by the following Gauss quadrature rule:

$$[K]^{(\mathrm{el})} = \sum_{\alpha} [N'(\mathbf{x}_\alpha)]^T [C(\mathbf{x}_\alpha)] [N'(\mathbf{x}_\alpha)] J(\mathbf{x}_\alpha) \omega(\mathbf{x}_\alpha) \tag{24}$$

where the summation is over all quadrature points $\mathbf{x}_\alpha$, $J(\mathbf{x}_\alpha)$ is the determinant of the transformation Jacobian evaluated at the quadrature point, and $\omega(\mathbf{x}_\alpha)$ is the weight corresponding to the quadrature point. Two possible methods for computing the elemental stiffness matrices have been investigated for the case where each processor of the Connection Machine system represents one nodal point per finite element. The two approaches are

- Each processor is assigned a unique quadrature point. The contribution of all the quadrature points to the elemental stiffness matrix are evaluated concurrently. Finally the volume integral is evaluated through a reduction operation on each element, concurrently.
- Quadrature is performed sequentially on each processor. The matrix–matrix product in Equation (24) is performed concurrently.

On a sequential computing system, the pseudo–code for evaluating an elemental stiffness matrix has the following structure:

```
loop over all quadrature points
    evaluate Jacobian and shape function
      derivatives for all quadrature points
    loop over rows in elemental stiffness matrix
        loop over columns in elemental stiffness matrix
            evaluate contribution to entry (row, column)
              of the elemental stiffness matrix
        end loop
    end loop
```

[3] area in two–dimensions.

end loop

Of the two algorithmic approaches introduced above, the former approach assigns a unique quadrature point to each processor. This corresponds to performing the quadrature loop concurrently. The latter approach corresponds to performing one of the two inner loops concurrently.

5.5.1 Concurrent quadrature

The first algorithm for computing the elemental stiffness matrices concurrently performs Gauss quadrature (represented by summation in Equation (24)) in parallel. Whenever the integral is estimated by complete Gauss quadrature, that is, the number of Gauss quadrature points is equal to the number of nodal points per finite element full processor utilization is achieved. The same interpolation function is evaluated concurrently at the different quadrature points in a step of the algorithm, and all other operations required for the evaluation of the function subject to quadrature are also uniform across processors. The contribution of all quadrature points to one matrix element of the elemental stiffness matrix is computed concurrently on all processors forming the finite element. A '+'–reduction is then required to complete the quadrature, and store the result at the appropriate processor. This method can be summarized by the following code fragment:

do concurrently over all quadrature points for all finite elements
  evaluate Jacobian and shape function
    derivatives for all quadrature points
  loop over rows in elemental stiffness matrix
    loop over columns in elemental stiffness matrix
      evaluate contribution of all quadrature
        points to entry (row, column)
      '+' **reduction** over all quadrature points
      store on processor containing the row of
        the elemental stiffness matrix
    end loop
  end loop
end do

By performing several evaluations before performing the '+'–reduction, the communication efficiency can be increased by replacing the '+'–reduction by an all–to–all

reduction on the set of processors representing the finite element. The above algorithm for generating the elemental stiffness matrices for the finite element mesh requires $O(n^2)$ parallel arithmetic operations to be performed on every processor, where n is the number of nodes in one finite element. In addition, u^2n^2 reductions, or u^2n all–to–all reductions, have to be performed over each finite element. Each of these reduction operations requires $O(\log(p+1))$ communications [20].

5.5.2 Sequential quadrature

As the order of the finite element (p) increases, the communication time in the algorithm outlined above begins to dominate the time required to generate the elemental stiffness matrices. The number of data elements that need to be communicated per processor ($O(n^2)$). The second algorithm for the concurrent generation of the elemental stiffness matrices alleviates the communication problem by performing the Gauss quadrature sequentially on all processors and generating the rows (or columns) of the elemental stiffness matrix concurrently as follows:

```
do concurrently over all rows of the elemental stiffness matrix
    loop over all quadrature points
        evaluate Jacobian and shape function derivatives
        loop over columns in elemental stiffness matrix
            evaluate contribution to entry (row, column)
        end loop
    end loop
end do
```

No communication is required for the above algorithm. In addition, the parallel arithmetic complexity of this scheme is also $O(n^2)$. However, since this algorithm performs the Gauss quadrature sequentially every processor representing a node of the finite element has to compute the Jacobian and the shape function derivatives for all quadrature points on the element. This implies that some redundant computations are performed by every processor. The overhead from the redundant computations is of order $O(\frac{1}{n})$ and reduces rapidly as the order of the finite element increases. For example, for a three dimensional finite element mesh composed of linear elements, the overhead due to redundant computations is approximately 10 %. This overhead reduces to less than 4 % for quadratic elements and less than 2 % for cubic elements.

	Conc. quad.	Seq. quad.
Arithmetic	$O(n^2)$	$O(n^2)$
Data communication	$u^2n^2 \times O(\log(p+1))$	0
Redundant computations	0	$O(n)$

Table 5: Arithmetic and communication complexity for elemental stiffness matrix generation by concurrent and sequential quadrature.

5.5.3 Comparison and discussion of elemental stiffness matrix generation

Table (5) summarizes the arithmetic and data communication involved for the two stiffness generation algorithms outlined above. The redundant computations reported in Table (5) assume that the number of quadrature points used to estimate the volume integral is the same as the number of nodal points per finite element. As the order of the finite elements increases, the number of quadrature points needed to evaluate the volume integral exactly is typically less than the number of nodal points per finite element. In general, a polynomial of degree $2r-1$ in one direction requires r quadrature points in that direction for exact integration. For example, a polynomial of order five ($p = 5$) requires only nine quadrature points in two–dimensions (3×3 quadrature rule) and twenty–seven quadrature points in three–dimensions ($3 \times 3 \times 3$ quadrature rule). Therefore, the amount of redundant computational effort performed by the second algorithm is really a fraction of n.

The storage requirement for the two algorithms discussed above is the same. Both algorithms require temporary matrices of size $u \times n$ to store the shape function derivatives and the global coordinates of the nodal points. In addition, two $u \times u$ matrices are required on every processor to compute the Jacobian and its inverse.

The computational effort spent in the generation of the elemental stiffness matrices becomes significant when the stiffness has to be computed several times during the analysis. This is the case for a non–linear analysis where typically more than half the computational effort is spent in evaluating the elemental stiffness matrices. In addition, an implicit dynamic analysis also requires the computation of the elemental stiffness matrices repeatedly. On the Connection Machine system, with several thousand processors, the time spent in generating the elemental stiffness matrices using the above algorithm is reduced significantly and is virtually independent of the number of finite elements composing the finite element mesh.

5.6 Solving the system of linear equations

The system of linear equations that results after accounting for the global interaction between the finite elements is *sparse* in a matrix representation. The sparsity reflects

the structure of the model, and in the two applications labeling all the nodal points along one dimension before any point in the next dimension (natural ordering) yields a matrix with non-zero $u \times u$ blocks along some of the diagonals. Traditionally the system of equations is solved using a band matrix solver. Parallel band matrix solvers based on a nested dissection ordering [10] are described in [17,19,5]. The solution time is proportional to the bandwidth and the logarithm of the size of the system. For an arbitrary sparse system finding a good elimination ordering is a very hard problem. A previous investigation [6] concluded that a nested dissection ordering often yields an ordering that has a higher degree of concurrency, but results in a higher fill–in than a minimum degree ordering.

A direct solver requires global communication, at least in some of the steps. If a multi–frontal method [7,8] (based on minimum degree or nested dissection ordering), is used for the solution of the system of equations, then the first step involves local interactions, but as the elimination process proceeds the interactions extend over a larger domain. The more balanced the elimination tree, the further apart in the initial ordering are the data elements involved in the final elimination stages. For a regular lattice the distance in the linear ordering is $\sqrt{N}$ for two–dimensional problems, and $N^{\frac{2}{3}}$ in three–dimensions. A systolic algorithm making uniform use of the processors laid out as a two–dimensional lattice for a two–dimensional application is described in [29].

Recently, the development of pre–conditioning techniques for the conjugate gradient method has resulted in an increased use of iterative techniques. For an iterative technique, there is no fill–in, and the data structure created for the representation of the problem can also be used in the solution process. In the context of the finite element method, this implies that an explicit assembly of the elemental stiffness matrices into a global stiffness matrix is not necessary.

For both iterative and direct–solution techniques, the most important issue involved is the interaction between two finite elements sharing a node. In iterative schemes this interaction appears as a sparse matrix–vector multiplication, which can be evaluated as a summation over all finite elements in the mesh

$$[K]\{U\} = \sum_i [K^{(i)}]\{U^{(i)}\}, \tag{25}$$

where $[K^{(i)}]$ is the elemental stiffness matrix and $\{U^{(i)}\}$ is the displacement vector corresponding to the i–th finite element. Assembly of the global stiffness matrix is avoided by the use of Equation (25). Clearly, communication between elementary objects is required to accumulate the inner–product necessary for most iterative solvers. The implementation of the finite element method described in this article uses the conjugate gradient method [13] with a diagonal pre–conditioner to solve the linear system. A description of the algorithm can be found in [11]. One of the main reasons for restricting attention to a diagonal pre–conditioner was an effort to gain experience with the essential characteristics of data parallel programming in the context of the finite element

method. A detailed study of the influence of pre–conditioners to the conjugate gradient method in a data parallel environment is in progress. Briefly, the computations of the conjugate gradient method are

initialize $\{U\}$
repeat until convergence
    compute residual : $\{r\} = \{F\} - [K]\{U\}$
    apply boundary conditions
    compute acceleration parameters
    evaluate new estimate for $\{U\}$
end loop

For the implementation described in this article, one processor of the Connection Machine system represents one nodal point per finite element. The rows of the elemental stiffness matrix corresponding to a nodal point were stored on the processor representing the nodal point. The evaluation of the sparse matrix–vector product requires *all–to–all broadcasting* for subsets of processors representing nodal points on a finite element, and *reduction over shared nodes* since the elemental stiffness matrices are not assembled into a global stiffness matrix. In addition, the conjugate gradient method requires global communication during each iteration for the evaluation and broadcast of the acceleration parameters [18].

- **all–to–all broadcasting:** With the above choice for the elementary object and the storage scheme, the evaluation of the sparse matrix–vector product in each conjugate gradient iteration requires that every processor representing a nodal point on a finite element receives the values of the displacements from all other nodal points on that finite element. This operation is termed "all–to–all" broadcasting and can be performed concurrently for all finite elements, and concurrently within each finite element. The displacement values are accumulated into a vector in each processor. The vector length is $n \times u$, the same as the size of the elemental stiffness matrix. Once the vector containing the elemental displacements has been accumulated, every processor in the lattice performs a simple "on–processor" matrix–vector multiplication. The final result on every processor in the lattice is the contribution of one finite element to the local residual corresponding to the nodal point represented by the processor.

- **assembly–of–shared–nodes:** To obtain the global residual, the local residuals from elements that share a node must be added together. This is analogous to the assembly of small right hand side vectors to obtain the global right hand side vector. Since the assembly procedure ('+' reduction) is required only for faces/edges that are shared; only nearest neighbor communication is required for this primitive.

	Processor per node	Processor per element
Vector length	u	nu
Number of SAXPY operations	nu	nu
Number of floating–point operations	$(2nu-1)\times u$	$(2nu-1)\times nu$

Table 6: Floating–point operations for matrix-vector multiplication on two and three dimensional Lagrange elements when no explicit assembly of the global stiffness matrix is performed.

When one processor represents one finite element, each processor performs a $nu \times nu$ matrix–vector multiplication. In the one processor per nodal point per finite element data representation, each processor performs a $u \times nu$ matrix–vector multiplication. Table (6) shows the arithmetic complexity for the two possible choices of the elementary object when no symmetry assumptions are made.

6 Applications

The three–dimensional finite element method implemented on the Connection Machine system was used to analyze two applications:

1. A square cantilever plate, fixed on one end and with a distributed force applied to the free end.

2. A long block with a square cross section. The boundary conditions for this application simulated a plane strain uniaxial loading. The discretization of the physical domain ensured that a Connection Machine system with 32K physical processors was completely utilized.

The schematic and the boundary conditions of the set–up of the first application are summarized in Figure (3). The plate dimensions were assumed to be 10 units long, 1 unit thick, and 400 units wide and discretized by first order (eight–node) iso–parametric elements, yielding a total of 4000 finite elements, 8822 nodal points, and 26466 degrees of freedom. The above schematic and loading conditions are similar to one of the Ansys benchmarks reported by Wagner and Swanson [28].

A Connection Machine system (CM–2) with 16K physical processors was used for the simulation. The processors were configured as a $32 \times 2 \times 1024$ lattice. The split finite element mesh yields a virtual processor ratio of 4.

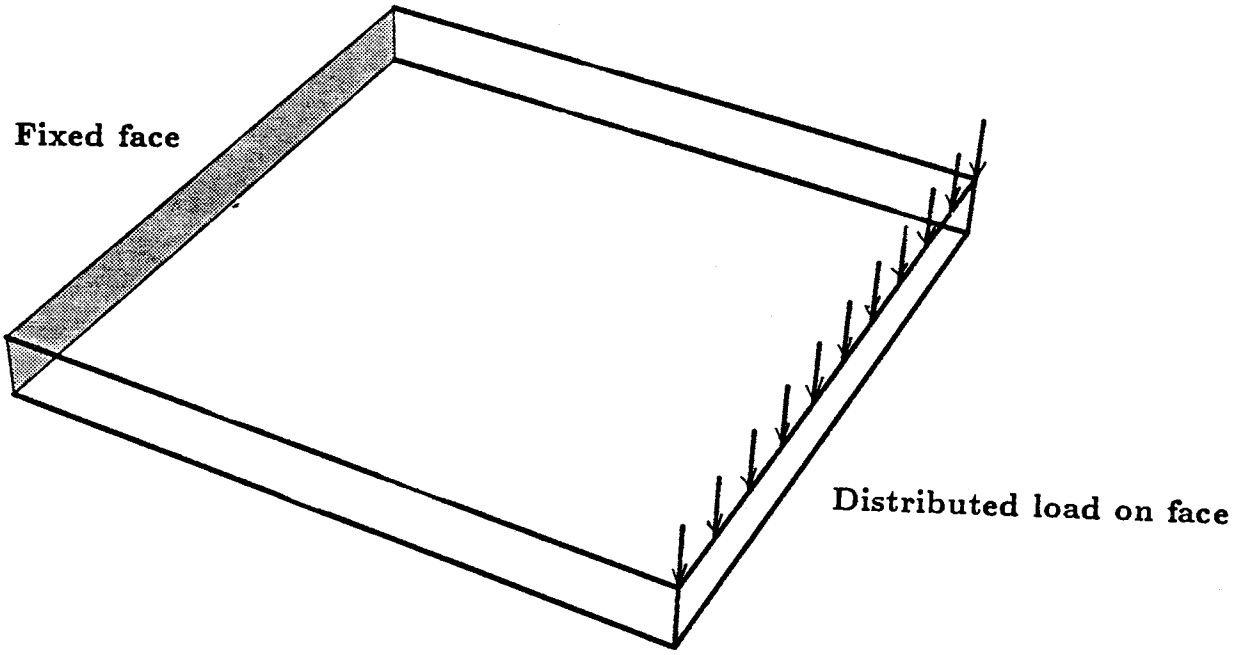

Figure 3: A square cantilever plate – geometry and boundary conditions.

6.1 Generation of elemental stiffness matrices

The elemental stiffness matrices were generated concurrently using the algorithm performing the quadrature for a matrix element of the elemental stiffness matrix sequentially. Table (7) shows the CPU timings as a function of the virtual processor ratio and the floating–point rate for a Connection Machine system with 64K physical processors. Measurements were made on a 16K configuration with 32–bit floating–point hardware option operating at a clock rate of 7.0 MHz. As the virtual processor ratio increases from one to eight, the projected rate increases from ~ 1.1 Gflops s^{-1} to ~ 1.8 Gflops s^{-1}. This rate is close to the peak performance attainable from the current version of *Lisp. As the order of the finite element increases, the computational effort required to evalu-

Virtual processor ratio	CPU time (s)	Projected GFlops s^{-1}
1	0.334	1.135
2	0.514	1.475
4	0.890	1.704
8	1.675	1.810

Table 7: Measured performance for single–precision stiffness matrix generation for linear three dimensional iso–parametric elements.

Order of finite elements	CPU time (s)	Projected GFlops s^{-1}
1	0.33	1.12
2	3.57	1.12
3	19.58	1.13

Table 8: Measured performance for single–precision stiffness matrix generation for three dimensional iso–parametric elements of different orders at a virtual processor ratio of one.

ate the elemental stiffness matrices increases as $O(n^2)$. Table (8) summarizes the CPU times recorded for the generation of the elemental stiffness matrices for different order of the finite elements.

6.2 The conjugate gradient solver

The global stiffness matrix that results from the finite element method is often ill–conditioned. The condition number of the stiffness matrix increases as the number of nodal points increase. Therefore, to preclude the possibility of increasing the number of iterations required for convergence because of round–off, the loading situations described above were simulated using double–precision floating–point operations. A double–precision floating–point hardware option is not yet available (projected delivery early 1989). Double–precision floating–point operations are performed in software and therefore are quite slow compared to the single–precision floating–point operations, which use hardware.

The rate of convergence of the conjugate gradient solver with a diagonal pre–conditioner is shown in Figure 4. This figure shows two sets of simulations, one corresponding to a Poisson ratio of 0.3 and the other corresponding to a Poisson ratio of 0.0. The non–zero Poisson ratio in the first simulation results in a three–dimensional displacement field. The second simulation reduces to a series of two–dimensional problems, that is, the component of the displacement field in the width direction is very small as compared to the other two displacement components. For both simulations, nearly 2500 iterations are required for the ℓ_2-norm of the normalized residual to reach a value of 1.0×10^{-3}, $\sim$ 4000 iterations for a normalized residual of 1.0×10^{-5}, and $\sim$ 4500 iterations for a normalized residual of 1.0×10^{-8} for the square cantilever plate with 26466 degrees of freedom.

To investigate the influence of discretization on the convergence rate of the conjugate gradient iteration, the geometry of the square plate and its discretization in the x–y plane (length–thickness plane) was kept fixed. The number of elements in the z direction (width dimension) were varied from 1 to 400. Figure (5) shows the number of iterations required for a residual of 1×10^{-8}. With a simple diagonal pre–conditioner the number

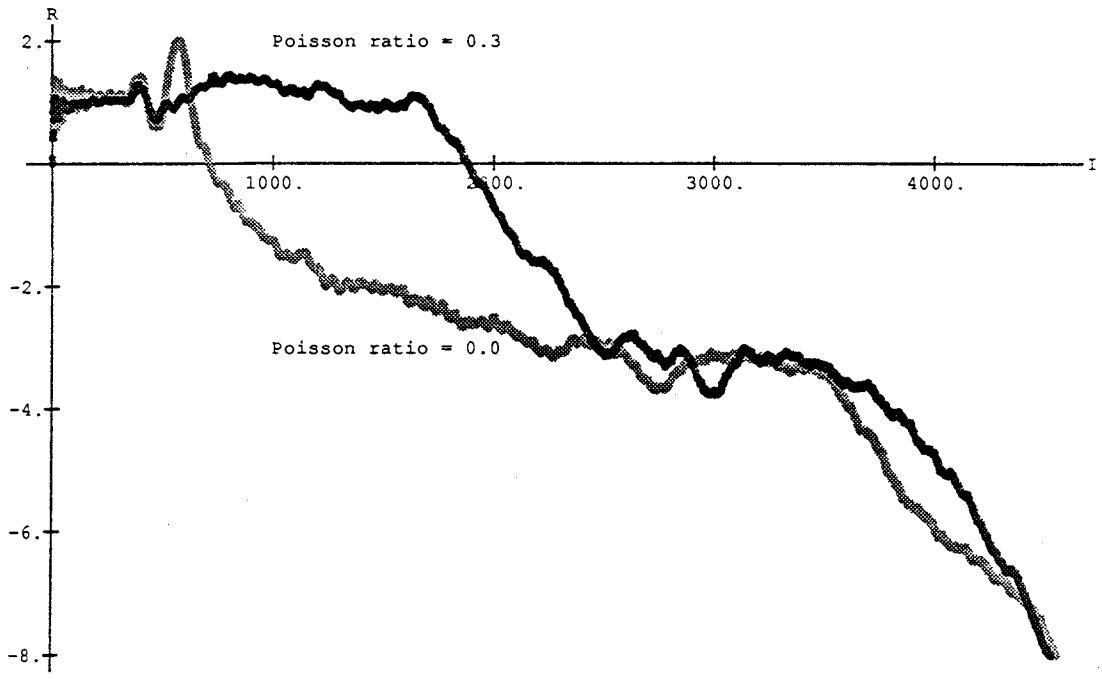

Figure 4: The ℓ_2-norm of the normalized residual as a function of the conjugate gradient iteration for the square cantilever plate (Mesh discretization : $10 \times 1 \times 400$); I is the conjugate gradient iteration number and R is the logarithm (base 10) of the normalized global residual.

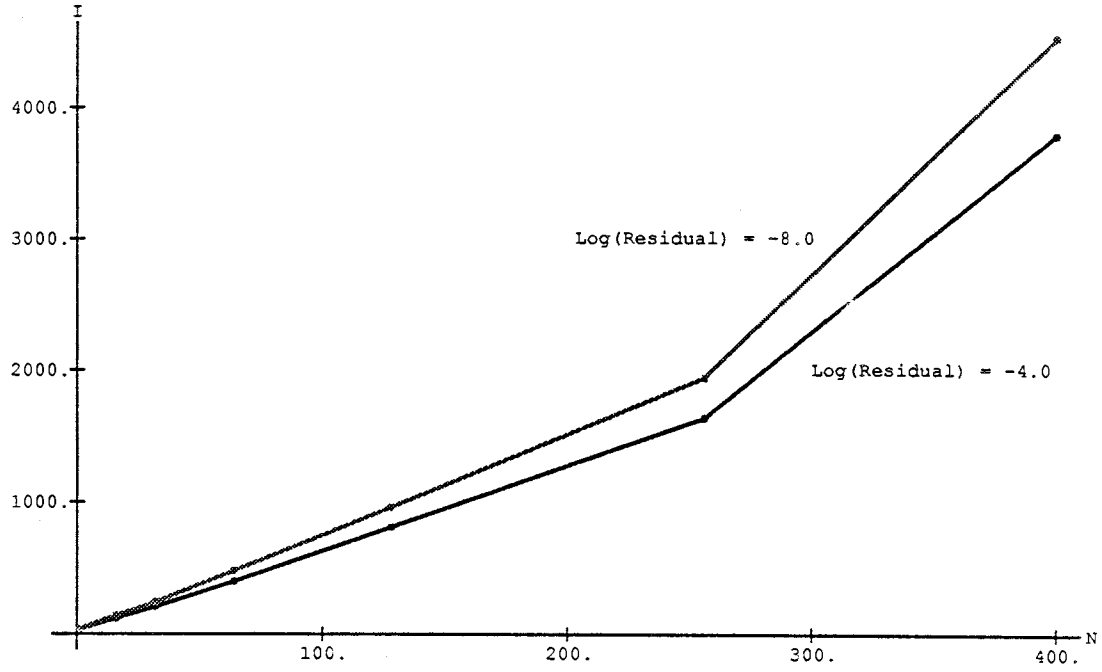

Figure 5: Number of conjugate gradient iterations (I) required for the normalized global residual to reach a value less than 1×10^{-4} and 1×10^{-8} as a function of the number of linear brick elements (N) in a $10 \times 1 \times N$ discretization.

Figure 6: Number of conjugate gradient iterations (I) required for the normalized global residual to reach a value ≤ 1.0 for various discretizations (N) in the width dimension.

of iterations required for convergence grow faster than linearly as the number of degrees of freedom increase in the width dimension. From Figure (4) it is also clear that the normalized residual is > 1.0 for a significant portion of the iteration process. Figure (6) shows the number of conjugate gradient iterations required for the normalized residual to achieve a value of than ≤ 1.0 as a function of the width discretization. The number of conjugate gradient iterations required for the normalized residual to fall below 1.0 is linearly dependent on the number of elements in the width dimension. The need for a better pre–conditioner is obvious.

The CPU time taken by the iterative solver for each conjugate gradient iteration can be divided into three parts:

$$t_{\text{iteration}} = t_{\text{all–to–all}} + t_{\text{assembly}} + t_{\text{computation}},$$

where $t_{\text{all–to–all}}$ is the communication time for a segmented "all–to–all" broadcasting, t_{assembly} is the time spent in assembling the residual vector, and $t_{\text{computation}}$ is the time spent in computing the sparse matrix–vector product, the acceleration parameters and updating the solution. Table (9) shows the CPU time for the three main sections of the solver during one iteration of the conjugate gradient method. The timings reported in this table are for double–precision floating–point operations. Consequently, the communication timings reported are also for 64–bit words.

Based on the timings obtained for the generation of the elemental stiffness matrices, the time taken on a Connection Machine system with double–precision floating–point

Virtual processor ratio	"all–to–all" broadcasting	Assembly	Computations	Total for one iteration
1	0.013	0.015	0.292	0.320
2	0.022	0.027	0.573	0.622
4	0.042	0.050	1.130	1.222

Table 9: CPU time (in s) for the three main sections of the conjugate gradient solver using double–precision (software) for three–dimensional eight node iso–parametric finite elements.

Virtual processor ratio	"Speed–up" factor
1	12.7
2	16.4
4	18.8

Table 10: Estimated "speed–up" factor for the conjugate gradient method on a Connection Machine system equipped with double–precision floating–point hardware.

hardware is estimated by defining a speed–up factor

$$\text{speed–up} = \frac{t_{\text{dp}}}{2t_{\text{sp}}}$$

where t_{dp} is the CPU time for stiffness matrix generation in double–precision with no floating–point hardware and t_{sp} is the CPU time for stiffness generation in single–precision with floating–point hardware. The CPU times for the stiffness generation are particularly useful for estimating the speed–up factor because there is no communication in this segment of the formulation. Moreover, the computations that are needed to evaluate the elemental stiffness matrices are sufficiently intensive so that the initial set up do not influence the timings. The estimated speed–up factors as a function of the virtual processor ratio are shown in Table(10). Therefore, the CPU time for a conjugate gradient solver with double–precision floating–point hardware, and a virtual processor ratio of one, is expected to be in the range of 0.04–0.05 s per iteration. These numbers predict that the total CPU time required by the conjugate gradient solver with a diagonal pre–conditioner, for the finite element mesh described above, is in the range 250–300 s, which compares very favorably with the CPU times reported by Wagner and Swanson [28], despite the poor convergence with the diagonal pre–conditioner.

The physical domain for the second application was chosen such that the primary storage of Connection Machine system was fully utilized. A finite element mesh with 1 element in the x–direction, 1–element in the y–direction, and 16384 elements in the z–direction was constructed so that the total number of degrees of freedom were 196620.

This geometry was used to simulate plane strain uniaxial stress loading on a Connection Machine system with 32K physical processors. To accommodate the "split" version of the finite element mesh, the Connection Machine system was configured as a $2 \times 2 \times 32768$ lattice of physical processors. This machine configuration corresponds to a virtual processor ratio of four. The CPU times recorded for the generation of the elemental stiffness matrices and the time per iteration for the conjugate gradient solver are very close to the timings reported above for the $10 \times 1 \times 400$ discretization of the square cantilever plate.

7 Conclusions

The domain discretization, the evaluation of the elemental stiffness matrices and the solution of the linear system of equations can be performed with a high degree of concurrency. From the analysis of data parallel algorithms for the finite element method, and its implementation on the Connection Machine system the following conclusions can be made:

- The degree of concurrency obtained with a processor assigned to a nodal point per finite element is a factor of $\sim (p+1)^3$ greater than the degree of concurrency obtained when a processor corresponds to a finite element, for three-dimensional brick elements. For first order elements ($p = 1$), the degree of concurrency is a factor of 8 greater in the three dimensional case and a factor of 216 greater for fifth order elements.

- The storage requirement per processor is a factor of $\sim (p+1)^3$ higher for a processor per finite element compared to one processor per nodal point per element. Symmetry of the stiffness matrix is more easily exploited with one processor per finite element. With a local storage of 8K bytes per processor the maximum order of brick elements that can be used when one processor represents one nodal point per element is four, but only one when a processor represents a finite element. The total storage requirement is the same for the two choices of the elementary objects.

- The maximum number of degrees of freedom that can be represented in the primary storage of the Connection Machine system model CM–2 with 512 Mbytes of storage is $\sim 780,000$ with three–dimensional first order brick elements, and $\sim 60,000$ for fourth order elements.

- When the elemental stiffness matrices are not explicitly assembled into a global stiffness matrix the computations for all nodes are the same for all elements of the same order.

- Lagrange type elements can make effective use of the lattice emulation capability. Serendipity elements require a more complex mapping for good processor and storage utilization.

- When one processor is assigned to a nodal point per finite element, data elements of the elemental stiffness matrices can be computed concurrently without communication by performing the quadrature for a single matrix element sequentially, but the quadrature for different matrix elements concurrently. For the higher level languages the peak arithmetic speed of $1.1 - 1.8$ Gflops s^{-1} is achieved at a clock rate of 7 MHz.

The two applications described in this article assume that one processor represents one nodal point per finite element. The domain for the two applications were discretized by brick elements. The discretization was computed using an algorithm by Steger and Sorensen and Jacobi iteration. The poor convergence rate of the Jacobi method was of no consequence because of extremely good initial guesses for the nodal coordinates.

For the solution of the linear system, a conjugate gradient method with a diagonal pre–conditioner was used. In the context of a data parallel environment an iterative solver has several advantages, namely

- the data structure used for the evaluation of the elemental stiffness matrices can also be used by the solver. There is no fill-in.

- the solution algorithm is highly concurrent.

- the sparse matrix–vector product only involves local interactions in the physical domain.

With Lagrange type elements, the lattice address mode and lattice communication primitives of the Connection Machine system the time per conjugate gradient iteration for linear brick elements for a finite element mesh with $\sim 400,000$ degrees of freedom is 1.25 s when the double–precision floating–point operations are performed in software. The corresponding time per conjugate gradient iteration is projected to be about 0.15 s with double–precision floating–point hardware (currently only single–precision floating–point hardware is available). The elemental stiffness matrices were not assembled into a global stiffness matrix. Instead the residuals were assembled during each step of the iteration process. The performance for the stiffness matrix computation was in the range 1.1 - 1.8 Gflops s^{-1}, and for the iterative solver the performance was in the range 0.5 - 0.7 Gflops s^{-1} for single–precision, hardware supported floating–point arithmetic.

Further research is currently directed towards finding optimal pre–conditioners for the system of equations in a data parallel environment. Research is also directed for

implementing direct solvers to be used in the pre–conditioning phase, or for the complete solution. The ability to efficiently handle multiple right hand sides is yet another motivation for a careful study of direct solvers.

8 Acknowledgements

A two dimensional version of the finite element code formed the starting point for this work. Special thanks go to Fabian Walaffe for many stimulating discussions. The two dimensional code was implemented by Fabian Walaffe and Amaury Fonseca, both of MIT and Thinking Machines Corp. A graphics interface for the two–dimensional code was produced by Luis Ortiz of Yale University and Thinking Machines Corp., and Alan Ruttenberg of MIT and Thinking Machines Corp.

References

[1] Guy E. Blelloch. Scans as primitive parallel operations. In *The 1987 International Conference on Parallel Processing*, pages 355–362, IEEE Computer Society Press, 1987.

[2] M.Y. Chan. *Dilation-2 Embeddings of Grids into Hypercubes.* Technical Report UTDCS 1-88, Computer Science Dept., University of Texas at Dallas, 1988.

[3] M.Y. Chan. The embeddings of grids into optimal hypercubes. 1988. manuscript.

[4] Thinking Machines Corp. *Connection Machine Model CM–2 Technical Summary.* Technical Report HA87-4, Thinking Machines Corp., 1987.

[5] Jack Dongarra and S. Lennart Johnsson. Solving banded systems on a parallel processor. *Parallel Computing*, 5(1&2):219–246, 1987. (Report ANL/MCS-TM-85, November 1986), YALEU/DCS/RR-519.

[6] Iain S. Duff and S. Lennart Johnsson. *The Effect of Orderings on the Parallelization of Sparse Codes.* Technical Report YALEU/DCS/RR-, Argonne Nat'l Laboratories and Yale Univ., Dept. of Computer Science, 1986.

[7] Iain S. Duff and John K. Reid. The multifrontal solution of indefinite sparse symmetric linear systems. *ACM Trans. Math. Softw.*, 9:302–325, 1983.

[8] Iain S. Duff and John K. Reid. The multifrontal solution of unsymmetric sets of linear systems. *SIAM J. Sci. Stat. Comp.*, 5:633–641, 1984.

[9] Y.C. Fung. *Foundations of Solid Mechanics.* Prentice-Hall, 1965.

[10] A. George. Nested dissection of a regular finite element mesh. *SIAM J. on Numer. Anal.*, 10:345–363, 1973.

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

[12] M. Gurtin. Variational principles for linear elastodynamics. *Arch Nat. Mech. Anal.*, 16:34–50, 1969.

[13] M.R. Hestenes and E. Stiefel. Methods of conjugate gradient for solution of linear systems. *J. Res. Nat. Bur. Standards*, 49:409–436, 1952.

[14] W. Daniel Hillis. *The Connection Machine.* MIT Press, Cambridge, MA, 1985.

[15] Ching-Tien Ho and S. Lennart Johnsson. On the embedding of arbitrary meshes in Boolean cubes with expansion two dilation two. In *1987 International Conf. on Parallel Processing*, pages 188–191, IEEE Computer Society, 1987. (Report YALEU/DCS/RR-576, April 1987).

[16] S. Lennart Johnsson. Communication efficient basic linear algebra computations on hypercube architectures. *J. Parallel Distributed Comput.*, 4(2):133–172, April 1987. (Tech. Rep. YALEU/DCS/RR-361, Yale Univ., New Haven, CT, January 1985).

[17] S. Lennart Johnsson. *Fast Banded Systems Solvers for Ensemble Architectures.* Technical Report YALEU/DCS/RR-379, Dept. of Computer Science, Yale Univ., March 1985.

[18] S. Lennart Johnsson. Highly concurrent algorithms for solving linear systems of equations. In *Elliptic Problem Solving II*, Academic Press, 1983.

[19] S. Lennart Johnsson. Solving narrow banded systems on ensemble architectures. *ACM TOMS*, 11(3):271–288, November 1985. (Tech. Rep. YALEU/DCS/RR-418, Yale Univ., New Haven, CT, November 1984).

[20] S. Lennart Johnsson and Ching-Tien Ho. *Spanning graphs for optimum broadcasting and personalized communication in hypercubes.* Technical Report YALEU/DCS/RR-500, Dept. of Computer Science, Yale Univ., New Haven, CT, November 1986. Revised November 1987, YALEU/DCS/RR-610. To appear in IEEE Trans. Computers.

[21] S. Lennart Johnsson and Peggy Li. *Solutionset for AMA/CS 146.* Technical Report 5085:DF:83, California Institute of Technology, May 1983.

[22] C. L. Lawson, R.J. Hanson, D.R. Kincaid, and F.T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM TOMS*, 5(3):308–323, September 1979.

[23] Michael Metcalf and John Reid. *Fortran 8X Explained.* Oxford Scientific Publications, 1987.

[24] E M. Reingold, J Nievergelt, and N Deo. *Combinatorial Algorithms.* Prentice-Hall, Englewood Cliffs. NJ, 1977.

[25] J.L. Steger and R.L. Sorensen. Automatic mesh-point clustering near a boundary in grid generation with elliptic partial differential equations. *J. Comp. Physics*, 33:405–410, 1979.

[26] J.F. Thompson, F.C. Thames, and C.W. Mastin. Automatic numerical generation of body-fitted curvilinear coordinate system for field containing any number of arbitrary two-dimensional bodies. *Journal of Computational Physics*, 15:299–319, 1974.

[27] Alan S. Wagner. *Embedding Trees in the Hypercube.* PhD thesis, University of Toronto, 1987.

[28] L. Wagner and J.A. Swanson. A study of the finite element analysis on supercomputers and near supercomputers. In *Proceedings of the 2nd International Conference on Supercomputing*, pages 119–130, ICS, 1987.

[29] Patrick H. Worley and Robert Schreiber. *Nested dissection on a Mesh-Connected Processor Array.* Technical Report CLaSSiC-85-08, Stanford Univ., Cent. Large Scale Sci. Computation, 1985.

[30] David Young. *Iterative Methods for Large Linear Systems of Equations.* 1971, 1971.

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

[23] Michael Metcalf and John Reid. *Fortran 8X Explained.* Oxford Scientific Publications, 1987.

[24] E M. Reingold, J Nievergelt, and N Deo. *Combinatorial Algorithms.* Prentice-Hall, Englewood Cliffs. NJ, 1977.

[25] J.L. Steger and R.L. Sorensen. Automatic mesh-point clustering near a boundary in grid generation with elliptic partial differential equations. *J. Comp. Physics*, 33:405–410, 1979.

[26] J.F. Thompson, F.C. Thames, and C.W. Mastin. Automatic numerical generation of body-fitted curvilinear coordinate system for field containing any number of arbitrary two-dimensional bodies. *Journal of Computational Physics*, 15:299–319, 1974.

[27] Alan S. Wagner. *Embedding Trees in the Hypercube.* PhD thesis, University of Toronto, 1987.

[28] L. Wagner and J.A. Swanson. A study of the finite element analysis on supercomputers and near supercomputers. In *Proceedings of the 2nd International Conference on Supercomputing*, pages 119–130, ICS, 1987.

[29] Patrick H. Worley and Robert Schreiber. *Nested dissection on a Mesh-Connected Processor Array.* Technical Report CLaSSiC-85-08, Stanford Univ., Cent. Large Scale Sci. Computation, 1985.

[30] David Young. *Iterative Methods for Large Linear Systems of Equations.* 1971, 1971.

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