Bell Laboratories

Document Cover Sheet

for
[X] Technical Memorandum
[ ] Internal Memorandum
[ ] Technical Correspondence

For help in completing this sheet, see ***Instructions for Completing Document Cover Sheet*** *(Form E-9272).*

Title: Finite Element Algorithms for Multiprocessors Using Distributed Variables

Author's Date: 6/15/84

Author(s)	Location	Ext.	Dept.
E. P. DeBenedictis	HO 4E610	5742	11353
D. N. Shenton *	*Electrical Engineering Department - Carnegie-Mellon University Pittsburgh, PA		

Document No.(s)

Dept.	Date yr/mo/day	Seq.	Category	Software Suffix	Filing Case No.(s)	WPN
11353	840615	09	TM	_	39394	311306-5000

Keywords: Distributed Computing, Algorithms, Numerical Analysis, Complexity Theory

MERCURY Announcement Bulletin Sections (check all that pertain):

[ ] CHM — Chemistry and Materials
[X] CMM — Communications
[X] CMP — Computing
[X] ELC — Electronics
[ ] LFS — Life Sciences
[ ] MAS — Mathematics and Statistics
[ ] PHY — Physics

ABSTRACT By using a finite element program as an example, this document presents programming techniques and a complexity theory for message passing multiprocessors. The document describes algorithms, using distributed variables, for all the computationally intensive tasks in a finite element program. The complexity theory predicts the performance of the algorithms on a general multiprocessor. Finally, the overall performance is used to evaluate the suitability of an architecture for this particular application.

The finite element material in this document is obtained from a finite element program developed by one of us (Shenton) at CMU. This program, targeted for magnetic field computation, is adaptive and uses Delaunay triangulation and complementary finite element methods. The computational model is a multiple instruction-multiple data (MIMD) message passing computer with up to 100,000 computational nodes. The basis of the programming techniques are distributed variables, a device developed

Page Arrangement

Pages of Text 25. Other Pages 1. Total 26.

No. Figs. 7. No. Tables 0. No. Refs. 4.

Mailing Label

Initial Distribution Specifications

Complete Copy		Cover Sheet Only	
Addressees (by Name, or by BTL Organization and/or Level)	Company, If Other Than BTL	Addressees (by Name, or by BTL Organization and/or Level)	Company, If Other Than BTL
1135 DH 1135 SUPV G. S. Indig 113 EXD		A. A. Penzias J. W. Tukey P. W. Andersson 11 EXD 45 EXD 52 EXD 54 EXD 55 EXD 59 EXD	11 DIR 113 DPH 1127 DPH 1135 MTS 1135 STA/TA

(If additional space is needed, attach another page.)

☐ **Software** (any nontrivial executable computer program routine, whether in source or object code, or any material from which such a routine can be readily derived)

Proprietary Classification

Unless otherwise indicated below, this document will be classified BELL LABORATORIES PROPRIETARY, as it is now marked on the front of the cover sheet. If you want one of the other classifications listed below, check the appropriate box, cross out the restrictive marking on the front of the cover sheet, and if needed, replace it with the appropriate marking and explanation.

Classification **Approval**

☐ BTL PROPRIETARY — NOTICE ______ Supervisor

☐ BTL PROPRIETARY — PRIVATE ______ Director

☐ No marking

☐ **Government Security Classified** (e.g., CONFIDENTIAL, SECRET, TOP SECRET). See *BTL Security Handbook.*

ABI Distribution

To expedite the movement of documents to ABI, Director-level action is requested regarding items (1) and (2), below, when the document is first distributed. In those cases where approval is not provided on the cover sheet, approval will be sought when a request is received from ABI, with consequent delay in filling the request.

Indicate whether the document:

(1) Contains network planning information, customer proprietary information, or nongeneric software for use in ABI products or services that BTL may not furnish to ABI. ☐ Yes ☒ No

(2) May be supplied on request to ABI R&D organizations. ☒ Yes ☐ No

6/19/84

Department Head - 11353 Director D. O. Reudink

Author Signature(s)

E. P. DeBenedictis D.N. Shenton

Complete if this document supersedes or amends an earlier one:

Earlier Document Number ______ Author ______

Filing Case No. ______ Date ______

For Use by Recipient of Cover Sheet:

To get a complete copy of this document:

1. Be sure your correct location is given on the mailing label on the other side.
2. Fold this sheet in half with this side out.
3. Check the address of your local Internal Technical Document Service if listed; otherwise, use HO 4F-112. Use no envelope.
4. Indicate whether microfiche or paper copy is desired.

Internal Technical Document Service

(X) HO 4F-112 ( ) ALC 1B-102
( ) IH 7K-101 ( ) MV 1D-40
( ) WH 3E-204 ( ) CB 1C-338

Please send a complete ☐ microfiche ☐ paper copy of this document to the address shown on the other side. UNIX™ or GCOS users may order copies via the *itds* system; for information, type "man itds" after logon.

Document Cover Sheet
(Continuation)

Title: Finite Element Algorithms for Multiprocessors Using Distributed Variables	**Author's Date:** 6/15/84

Author(s) (Continued)	Location	Ext.	Dept.

Document No.(s) (Continued)

Dept.		Date yr/mo/day		Seq.	Cate-gory	Software Suffix	Filing Case No.(s)	Charging Case No.(s)
_ _ _ _ _	–	_ _ _ _ _ _	–	_ _	_ _	_		
_ _ _ _ _	–	_ _ _ _ _ _	–	_ _	_ _	_		

Complete Copy (Continued)		Cover Sheet Only (Continued)	
Addressees (by Name, or by BTL Organization and/or Level)	Company, If Other Than BTL	Addressees (by Name, or by BTL Organization and/or Level)	Company, If Other Than BTL

E-9271-C (7-83) *If additional space is needed for the Abstract, use the reverse side of this sheet.*

Abstract (Continued)

by the other of us (DeBenedictis) at AT&T Bell Laboratories. The complexity theory includes the effects of computation speed, bandwidth, and latency to predict the performance of a program over a wide range of message passing multiprocessor architectures.

AT&T Bell Laboratories

subject: **Finite Element Algorithms for Multiprocessors Using Distributed Variables**

WPN 311306-5000
Filing 39394

date: **June 15, 1984**

from: **E. P. DeBenedictis**
HO 11353
4E-610 x5742

D. N. Shenton
E. E. Department
Carnegie-Mellon University

11353-840615-09-TM

TECHNICAL MEMORANDUM

By using a finite element program as an example, this document presents programming techniques and a complexity theory for message passing multiprocessors. The document describes algorithms, using distributed variables, for all the computationally intensive tasks in a finite element program. The complexity theory predicts the performance of the algorithms on a general multiprocessor. Finally, the overall performance is used to evaluate the suitability of an architecture for this particular application.

The finite element material in this document is obtained from a finite element program developed by one of us (Shenton) at CMU. This program, targeted for magnetic field computation, is adaptive and uses Delaunay triangulation and complementary finite element methods. The computational model is a multiple instruction-multiple data (MIMD) message passing computer with up to 100,000 computational nodes. The basis of the programming techniques are distributed variables, a device developed by the other of us (DeBenedictis) at Bell Labs. The complexity theory includes the effects of computation speed, bandwidth, and latency to predict the performance of a program over a wide range of message passing multiprocessor architectures.

1. Introduction

Finite element programs approximate the solution to a problem defined in a continuous two (or higher) dimensional region by using a mesh of discrete subregions called finite elements. A frequently used finite element is a triangle with solution values defined at the vertices and midpoints of the sides. Figure 1 illustrates such a triangle. Within a single triangle the continuous function is represented by a biquadratic equation.

When a region is divided into many triangles, the density of triangles in a particular area determines the precision with which the continuous function is modeled. Figure 2 illustrates the triangles around a triangular wedge of dielectric material. Note the higher density of triangles near

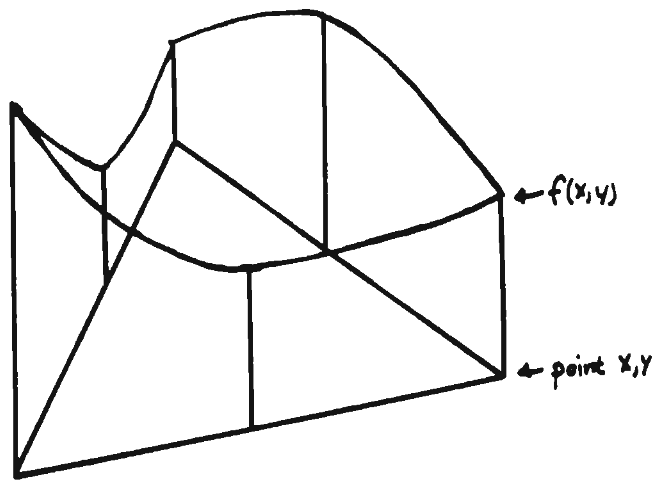

Figure 1: A Triangular Finite Element

the points of the dielectric material.

A finite element problem is solved as a single matrix equation of the form $A\vec{x} = \vec{b}$. The numbers in the solution vector $\vec{x}$ are really the coefficients of the biquadratic equation defining the triangles, but can be thought of as the values of the approximated solution at the points (vertices and midpoints of the sides) of the mesh. If there are n points in $\vec{x}$, A is a (sparse) $n \times n$ matrix. A is called the stiffness matrix, and its entries represent dependencies between points. The vector $\vec{b}$ is the boundary conditions.

The numerical method of solving $A\vec{x} = \vec{b}$ involves iteration. The Incomplete Cholesky Conjugate Gradient (ICCG) decomposes A into the product of LL^t, where L has the same sparsity pattern of A. The conjugate gradient method is used iteratively to find $\vec{x}$ from an initial arbitrary value. The mathematical complexity of ICCG is given as $n^{1.2}$, where n is the dimension of the vector.

The solution of the system of equations $A\vec{x} = \vec{b}$ provides the answer for the specified mesh. In the adaptive procedure [Cendes 82], two approximate solutions are derived, the difference between them providing an element by element measure of the accuracy of the solution. By refining those elements having the largest errors and recomputing the solution iteratively, finite element meshes having a uniform error density are obtained.

1.1 Multiprocessor Programming

The multiprocessor algorithm presented here divides the entire region into contiguous subregions, one for each computational node, of the multiprocessor. The nodes execute instructions independently, but must communicate intermediate values at various points in the processing to correctly execute the algorithm.

Figure 3 shows the mesh of figure 2 broken into eight pieces. Note that the triangles are divided to equalize the quantity in each node while retaining a compact shape.

A computational node contains data on all the points in and at the boundary of its region.

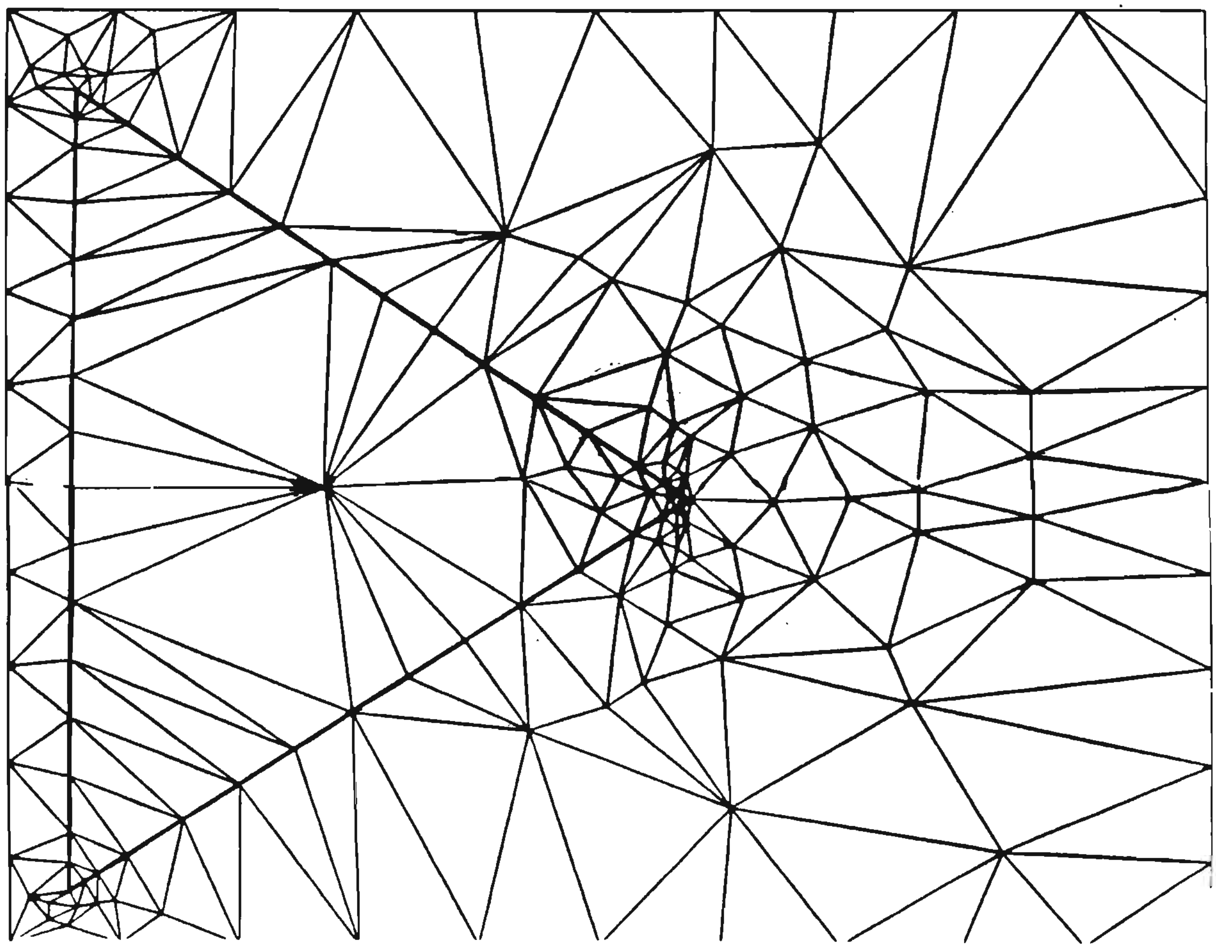

Figure 2: Triangularization of a Region

Boundary points are duplicated between computational nodes. A computational node contains all the A-matrix elements between its points. The structure of the A matrix follows the mesh closely. Each row and column of A represents a point (vertex or midpoint of a side) of a triangle, and the elements of A represent dependencies between points in the mesh. Two points in the mesh are dependent only if there is a triangle that contains both of them.

1.2 Overall Program Structure

Although the multiprocessors under discussion are MIMD machines, there is some similarity between the structure of the algorithms and a single instruction-multiple data (SIMD) machine. The proposed program of the MIMD multiprocessor includes making a SIMD machine with a matrix operation instruction set. The following matrix operations are done by this virtual SIMD machine with a single or fixed sequences of commands:

(1) vector-scalar multiplication
(2) vector inner product

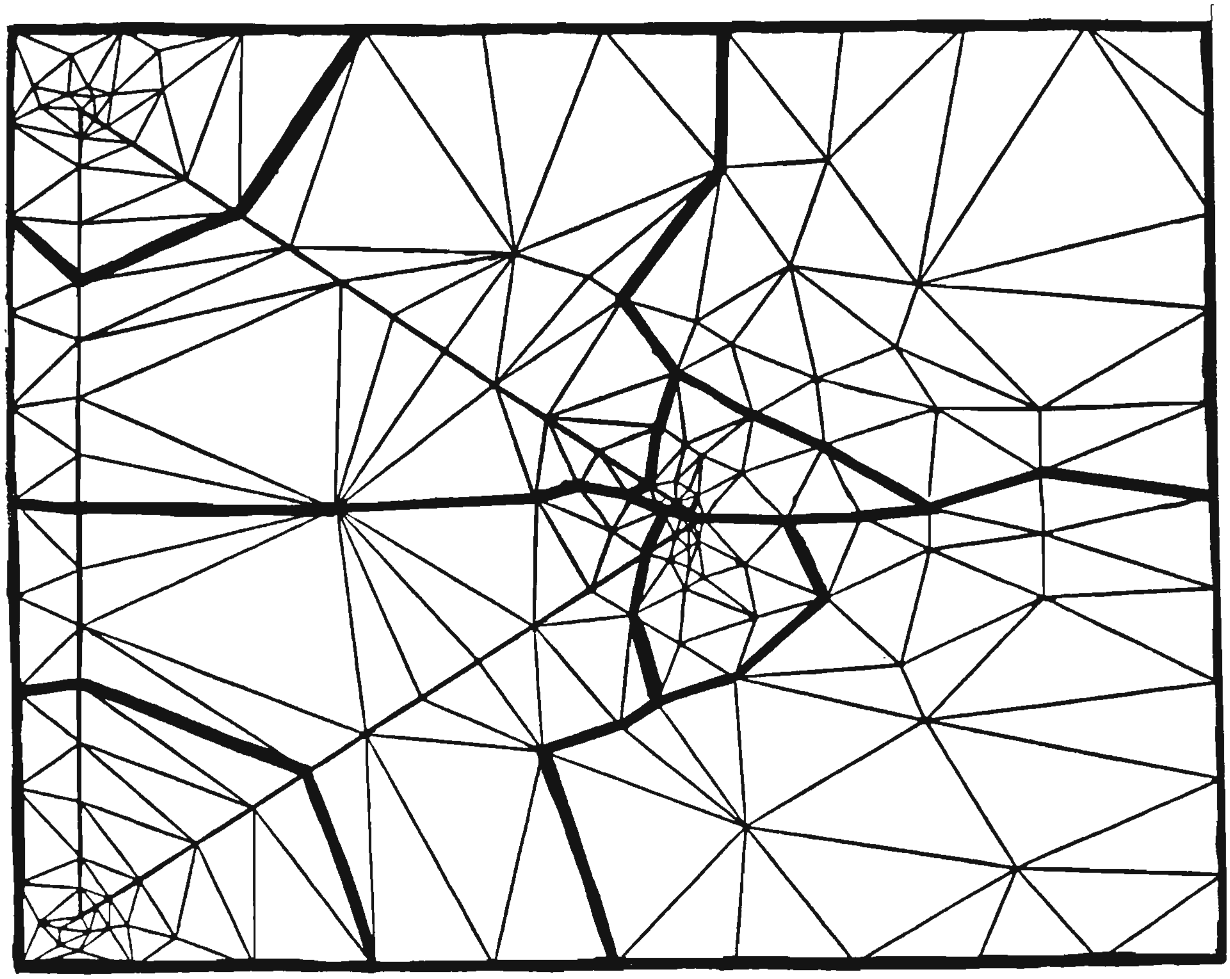

Figure 3: Partitioning of a Region

(3) matrix-vector multiplication
(4) solving $L\vec{x} = \vec{b}$, for $\vec{x}$, where L is lower triangular, $\vec{x}$ and $\vec{b}$ are vectors
(5) the incomplete cholesky decomposition $LL^t = A$, where A is given

The complete algorithm for a finite elements program involves repeated matrix, vector, and scalar operations along with a computationally insignificant amount of control. A conventionally coded control program manipulates this highly parallel processor programmed only for matrix operations. The control program, resident in just one computational nodes, would manage graphical input and output of the problem and its solution, generate matrix operation commands for the rest of the processor, and do its own (scalar) control computations for convergence and related issues.

1.3 Algorithm Complexity

The crucial question about algorithms for multiprocessors is how efficient they are in execution. Since multiprocessors are a rapidly advancing technology, efficiency measurements on existing hardware is of marginal utility. The approach used here is to describe the performance of an algorithm in terms of four generic multiprocessor performance parameters.

The four parameters are: I, the instruction time within each computational node, F, the floating point time, M, the time to send a message, and L, the message transfer time from writer to reader. The measures I and F are the measures used in conventional complexity theory. Sometimes, as here, one of I and F dominate and the other can be ignored. M is related to the bandwidth of the message passing network. L is the message latency, or the amount of time a message is within the message passing network.

An possibly fruitful exercise, which is done here, is to consider the design of a multiprocessor that would execute these algorithms efficiently. By analyzing the performance of the algorithms in terms of the four performance parameters it may become evident that certain relative values of the parameters would be optimal.

2. Distributed Variables

The reader is familiar with the role of variables in conventional programming. Distributed variables are the extension of many of the characteristics of conventional variables to MIMD computers.

A variable is a conceptualization of a temporary value in a program. When desired a variable has a name such that reference anywhere to the name refers to the same conceptual object. The names of variables can be changed, however, by passing the variable to a subroutine, supporting the concept of a variable as a conceptual object, rather than just a name.

Each language employs a set of interactions with its variables; each language has a self-consistent set, but different languages have distinctly different operations. In Fortran-type languages, a variable is the address of an area in memory. A Fortran-type variable is read by the name appearing in expressions and written by appearing to the left of an equals sign. In Smalltalk or Simula, variables can refer to objects, where interactions are functios calls on objects.

Within a single computational node, a distributed variable has a name like a Fortran variable. A distributed variable retains its meaning when used on different computational nodes, however. When invoking a subroutine, for example, on a different computational node, distributed variables used as arguments refer to the same conceptual object when referenced by either the calling program or the subroutine.

Since shared memory not available on message passing machines, global variables, that can be refered to anywhere merely by knowing its name, are generally disallowed. Variables must get around through other means, such as function invocation, or through pre-existing distributed variables.

2.1 Mailbox Type Distributed Variables

The Mailbox variable probably corresponds most closely to the common interpretation of a message passing channel. The allowed interactions with a Mailbox variable are putting a value into it and taking one out. The values are typed when the variable is declared; e.g. values could be characters, other distributed variables, or structures. Generally speaking the variables will come out in the order they went in, although this concept is unclear when there are multiple readers or writers. Unlike the channels in many conventional operating systems, there is no opening or closing of Mailbox variables; all that is needed to interact with one is its name. Mailbox variables have a queue for values, the size is specified when the variable is created, but must always be greater than zero.

2.2 Other Types of Variables

A possibly expected variation on the Mailbox variable is the Broadcast variable. A Broadcast variable may have a several readers, each of which will get every value put into the variable. Like Mailbox variables, anybody can write a value into a Broadcast variable.

A probably unexpected variation on the Broadcast variable is the Command variable, or a Broadcast variable with stronger synchronization. Interactions with distributed variables have two synchronization events; Mailbox and Broadcast variables use only one. A read from a Command variable synchronizes when the data is available, and again when the reader has completed any processing associated with that data. A writer to a Command variable therefore wait until its value has been read and used by all the readers.

There is a reverse version of the Broadcast variable that has considerable application. A broadcast has the property that one value goes in and many come out; in a reverse broadcast, many values go in and one comes out. A reverse broadcast variable has an associated operation, such as addition, that is applied to the written values to produce the value that is eventually read. The Sum type distributed variable is reverse broadcast on input and broadcast on output. A Sum variable adds all the values written and then the sum can be read by multiple readers.

2.3 Extensibility of Types

With an abstractive concept with as many variations as distributed variables, much of the power is derived from the programmer being able to select the variations most appropriate to his task.

It is possible to describe the characteristics of a distributed variable type through such things as state transition tables, making formal descriptions of existing and proposed variables possible. It may also be possible to build programming systems that operate on dynamic definitions of the lowest level distributed variable characteristics.

2.4 Performance of Distributed Variables

Even without knowledge of how distributed variables will be used, it is possible to find upper bounds on some aspects of their performance. The dominant execution cost in a distributed variable is the number of messages that get passed between nodes, and minimums for these can be investigated.

The time required to pass a message to a known destination node is assumed to be constant as a function of the number of nodes in the system. Theoretically, however, the message latency must be a slowly increasing function of time. Present hardware is limited by switching events, which scale as $\log_2 n$. Were speed of light delays to be significant, message latency would vary as $n^{\frac{1}{3}}$. These functions vary slowly enough that approximation by a constant is reasonable.

A Mailbox variable can be as efficient as the underlying hardware only when used in certain ways. If one node were usually the writer and another usually the reader, a appropriately designed Mailbox variable would adapt and send messages to the right place before they are requested. If a Mailbox has many readers at once, it is probably most efficient to store the messages at a central location and have readers send request messages there. Under these circumstances, a Mailbox read would require a request message and a data message for each value transferred.

The spatial distribution of information as in a Broadcast variable requires at least $\log_2 n$ messages. Distribution through a tree structured network of message relays is probably most efficient. Sequential message delays in a tree structured distribution is $\log_f n$, where f is the fanout; whereas sequential transmission would require n sequential delays.

3. *Data Representation*

A graphical representation of vectors and sparse matrices is used as in figure 4. Point data structures represent vector elements or rows and columns of matrices. An attribute of point i is x_i, the i'th element of vector $\overline{x}$. Arcs represent the non-zero matrix elements; matrix element A_{ij} is represented by an arc between point i and point j.

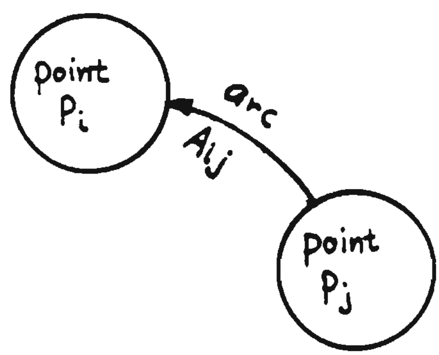

Figure 4: Graphical Representation of Vectors and Matrices

3.1 *Data Structures*

The data structures for points and arcs are shown below:

triangle data structure

attribute	*type*	
sides[3]	pointer to triangle	*links to other triangles*
vertex[3]	pointer to point	*links to vertex points*
midpoint[3]	pointer to point	*links to midpoints*
arc[6,6]	pointer to arc	*links to arcs*

point data structure

attribute	*type*	
x	floating point	*vector element*
b	floating point	*vector element*

arc data structure

attribute	*type*	
head	pointer to point	*point at head of arc*
tail	pointer to point	*point at tail of arc*
a	floating point	*matrix element*
l	floating point	*lower triangular matrix element*

The highest level data structure is the triangle. The topology of the mesh is represented by the connections between the triangle structures. Each triangle has pointers to the three other triangles sharing sides.

The point data structure represents the vertices or midpoints of the sides of the triangles. The b and x attributes of a point data structure represent vector elements in certain matrix-vector operations. Each triangle has three mesh points at its vertices (which define the triangle) and three points representing the midpoints of the sides. The triangle data structure has six pointers to data structures representing these points.

The non-zero elements in the stiffness matrix are represented by the arc data structure. Generally, the ij'th element of a matrix is represented by an arc from the j'th to i'th point (i.e. an arc with the head pointer addressing the i'th point and tail pointer addressing the j'th point). If a matrix is symmetric, as is the stiffness matrix, the head and tail pointers are interchangeable, allowing one arc to represent two elements. If the matrix is lower triangular, as is the matrix resulting from the incomplete Cholesky decomposition, the value of the element is zero in the opposite direction of the arc.

3.2 Boundaries and Shared Points

At the boundaries between regions, the structure described above is amended. Each computational node has a complete and consistent description of its region. Where two nodes join, which is always along a side of a triangle, points are duplicated. The program that sets up the points must create the duplicate points and verify that their data is consistent.

Arcs between two points, both on a boundary, are duplicated in two nodes. The value of the matrix element is the sum of the values associated with all such arcs. See figure 5.

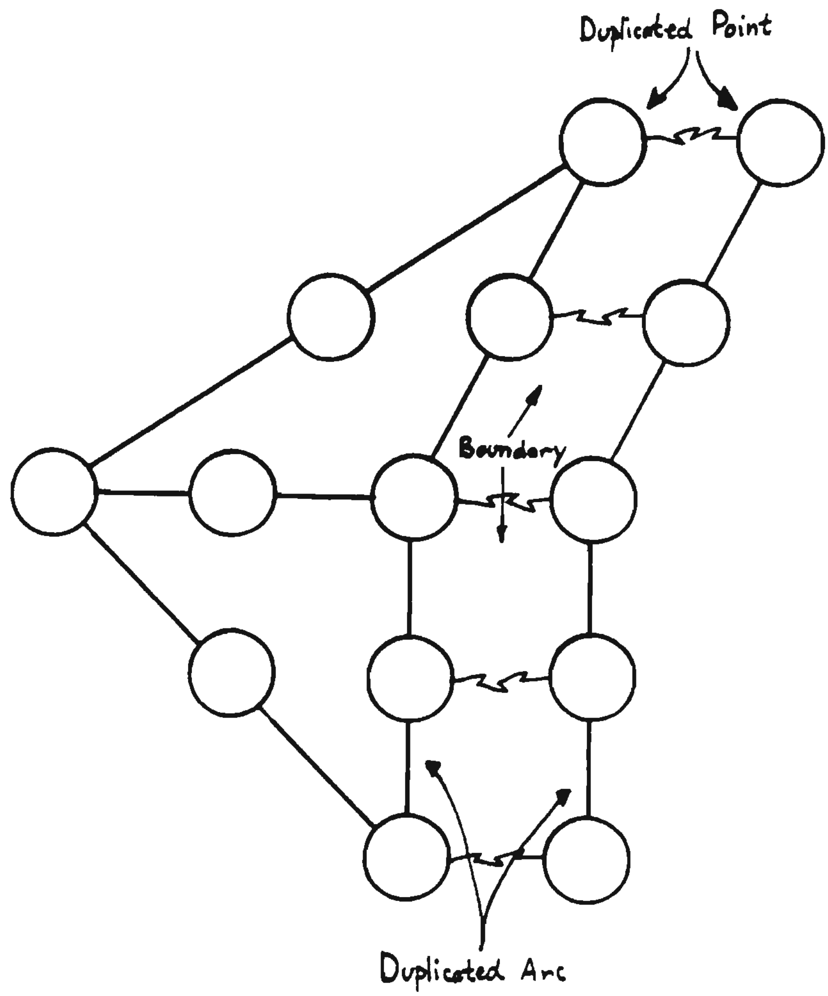

Figure 5: Boundaries and Shared Points and Arcs

3.3 Example Values

The performance of a multiprocessor on the algorithms that follow depends on how many triangles are in the memory of each processor. It is assumed here that a problem always fills memory, and scaling occurs by increasing the number of nodes (and hence adding memory indirectly.) The quantity of triangles is equal to the available data memory divided by the average data size per triangle.

We designate the quantity of triangles as T, the quantity of mesh points as P, and the quantity of arcs A. Analysis of existing meshes suggests the following ratios are typical of a two dimensional adaptive finite element mesh:

Assertion
$\frac{P}{T} = 2$ *average of 2 points per triangle*
$\frac{A}{P} = 15$ *average of 15 arcs per point*
End of Assertion

Engineering considerations suggest that the appropriate memory size if 500 K bytes per computational node with perhaps 350 K bytes available for data storage; implying a typical figure of 400 triangles per node.

Example Values
$T = 400$ *triangles per node*
$P = 800$ *points per node*
$A = 12000$ *arcs per node*
End of Example Values

Given that the subregions in each node are compact, the perimeter of a region approximates the square root of its area, and certain relations result. The maximum block number is limited similarly.

Example Values
$4\sqrt{P} = 113$ *points on a boundary*
$8\sqrt{P} = 226$ *arcs on a boundary*
$B = 20$ *maximum block number*
End of Example Values

3.4 A Block Form of a Finite Elements Matrix

The theoretical amount of concurrency for several of the algorithms explored here is vastly improved by finding a block form of the stiffness matrix. The block form that can be employed, illustrated in figure 6, has diagonal submatrices on the diagonal of the main matrix. When looking for block forms, the goal if to divide the matrix into few large blocks.

To specify a block form of a matrix expressed graphically it is only necessary to assign a block number to each point. If we want the diagonal blocks to be diagonal themselves we must assure that no point has an arc to another point with the same block number.

3.4.1 Bounds on Block Number Size

It can be proven that there exists a block number assignment for the two dimensional mesh used here where the largest block number is eight.

An algorithm to assign all points to eight blocks is outlined: Each of the eight blocks is assigned a color and the block assignment is equivalent to doing two map-coloring operations on the points. The first map coloring operation assigns four of the eight colors to only the vertex points. The coloring is done such that no two points in the same triangle have the same color. The second map coloring operation treats the vertex points as holes in the map and colors the midpoints. The second coloring uses four different colors from the first coloring and assigns colors so no two points in the same triangle have the same color.

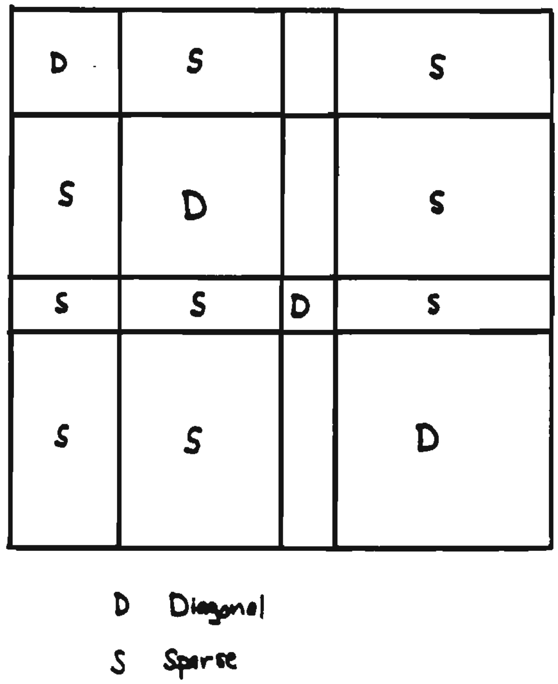

Figure 6: Matrix Interpretation of Block Numbers

3.4.2 An Algorithm to Find a Block Numbering

Algorithm A. (Block number assignment.) Given a set of points $P_n = P_1 \ldots P_p$ and arcs $A_n = A_1 \ldots A_a$, assign block numbers to the points.

A1: [Iterate.] Do A2 for all P_n in any order.

A2: [Assign.] Set $P_n \leftarrow$ the smallest block number different from all its neighbors.

End of Algorithm

This algorithm can be executed in parallel as long as the points being numbered do not share an arc.

4. Algorithmetic Complexity Issues

The study of algorithms for conventional computers is a substantially more mature field than that for multiprocessor algorithms. The differences between the two fields is in the model used to evaluate the efficiency of an algorithm. The running time of an algorithm for a conventional machine is measured by total floating point computations. By contrast, it is proposed here that the running time on a multiprocessor depends on three things: (1) the time required to communicate intermediate results, (2) the time required to resolve sequential dependencies that prevent computational nodes from operating in parallel, and (3) total floating point operations.

Other authors have proposed two efficiency measures for parallel programs; speedup and efficiency [Jordan 82]:

Definition

N	*number of computational nodes*
T_i	*time to run with i computational nodes*
$S = \frac{T_1}{T_N}$	*speedup*
$E = \frac{S}{N}$	*efficiency*

End of Definition

The speedup is the amount faster a program runs on a multiprocessor compared to a conventional computer. Speedup is limited by the number of nodes in the multiprocessor. In practice, any speedup of order N is excellent. Efficiency is the fractional utilization of the nodes of the multiprocessor.

4.1 Specific Performance Measures

The multiprocessor is a collection of N computational nodes and a message passing system. Each computational node is modeled as a conventional Von-Neumann computer, with floating point, and with a message passing IO interface. The performance of a multiprocessor is measured here by the following:

Definition

I	*instruction execution time (not used here)*
F	*floating point execution time*
M	*message send or receive time*
L	*latency in message transmission*

End of Definition

The instruction execution time can be measured by having one computational node execute conventional instructions and measuring the average time. The floating point rate is measured similarly, but depending on the application, a mix of regular instructions may be executed also but not counted. The message send or receive time is measured by having every computational node send messages and measuring the average time. Note that on some multiprocessors, the message time will be limited by system software and on others by the bandwidth of the interconnect. The message latency is measured by having two nodes send a message back and forth continuously. L is the average time between message transmissions (on two processors.)

Although the performance parameters may in theory take any values, the following relations are true of any sensible system:

Assertion	
$2 \leqslant N \leqslant 100{,}000$	*computational nodes*
$F > I$	*floating point slower than instructions*
$M >> F$	*message time much slower than floating point*
$L >> M$	*latency much greater than message time*
End of Assertion	

In this document the quality of multiprocessor algorithms is measured as an expression in terms of the performance parameters. Unfortunately, such an expression, while being general, may not provide the necessary insight into the parallel nature of the algorithm. To remedy this the execution time is evaluated in terms of sample values of the performance parameters. These values, based on a proposed design [DeBenedictis 84] that uses expected 1987 technology.

Example Values	
N = 4096	*cost of $4M for the CPU*
I = 500nS	*2 MIP instruction rate*
F = 2uS	*.5 MFLOP floating point rate*
M = 100uS	*10,000 messages per second*
L = 1mS	*1 mS queuing delay in network*
End of Example Values	

Given these sample performance measures, the mix of the various operations that matches the design of the machine most effectively is:

Example Values	
200	*instruction executions*
50	*floating point operations*
1	*message transmissions and receptions*
.1	*sequentially dependent messages*
End of Example Values	

5. *Programming*

The overall program structure for adaptive finite elements uses heuristics that are currently a research topic. Aside from the heuristics, however, well known algorithms are employed.

The heuristic techniques, not discussed further here, sets up two complementary finite element problems representing upper and lower bounds of the actual solution. The heuristics also refine the mesh based on the two solutions.

The computationally intensive part of the program is the solution of the matrix problem $A\overline{x} = \overline{b}$, where A is a matrix of the form described earlier. The equation $A\overline{x} = \overline{b}$ is solved by the ICCG method [Cendes ??].

Algorithm ICCG. (Incomplete Cholesky Conjugate Gradient.) Given iteration limit n, matrix A, boundary conditions $\vec{b}$, and accuracy limit ϵ, compute $\vec{x}$ where $A\vec{x} = \vec{b}$.

ICCG1: [Initialize.] Set $\gamma \leftarrow 1$,
$\vec{p} \leftarrow 0$,
$\vec{x} \leftarrow$ arbitrary vector.

ICCG2: [Compute incomplete Cholesky.] Compute L where $LL^t = A$.

ICCG3: [Iterate.] Do ICCG4...ICCG6 for $i=1...n$.

ICCG4: [Compute.] Set $\vec{r} \leftarrow \vec{b} - A\vec{x}$.

ICCG5: [Terminate.] If $\mathrm{norm}(\vec{r}) < \epsilon\, \mathrm{norm}(\vec{b})$ terminate algorithm.

ICCG6: [Compute.] Set $\vec{r} \leftarrow L^{-1}\vec{r}$,
$\theta \leftarrow \vec{r} \cdot \vec{r}$,
$\beta \leftarrow \frac{\theta}{\gamma}$,
$\gamma \leftarrow \theta$,
$\vec{r} \leftarrow L^{-t}\vec{r}$,
$\vec{p} \leftarrow \vec{r} + \beta\vec{p}$,
$\delta \leftarrow \vec{p}A\vec{p}$,
$\alpha \leftarrow \frac{\gamma}{\delta}$,
$\vec{x} \leftarrow \vec{x} + \alpha\vec{p}$.

End of Algorithm

The matrix operations described above for one iteration of the ICCG, as a function of n, the actual number of iterations, is:

Operation	*Number*
Vector-Scalar Multiplication	5n
Vector Inner Product	1n
Matrix-Vector Multiplication	2n
Back Substitution	2n
Incomplete Cholesky	1

5.1 Vector-Scalar Multiplication

To compute $\vec{x}$ where $\vec{x} = c\vec{b}$, $\vec{x}$ and $\vec{b}$ are vectors and c is a scalar, the elements of $\vec{x}$ are defined by $x_n = cb_n$.

The operations can be carried out in any order or all at once. In practice, all the computational nodes operate in parallel, doing their operations serially in any convenient order.

Algorithm V-Uniprocessor. (Vector-Scalar Multiplication.) Given a scalar c, and vectors $\vec{x}$ and $\vec{b}$ defined by points $P_1 \ldots P_p$, compute $\vec{x} = c\vec{b}$.

VU1: [Iterate on points.] Do VU2 for $i = 1 \ldots p$.

VU2: [Multiply.] Set $x_i \leftarrow cb_i$.

End of Algorithm

The distribution of the information directing all the computational nodes to do a vector-scalar multiplication, as well as the scalar, is an issue. The required sequence of events is: (1) the decision to do a vector-scalar multiplication, and the scalar is generated at a centralized location, (2) this command is distributed to all computational nodes, (3) the nodes do the multiply in parallel, (4) information indicating that the operation has been completed in all nodes is delivered back to the centralized location.

(A notational comment on multiprocessor algorithms: These algorithms typically have several parts. There is usually one instantiation of a control part, and many instantiations of a node part. When the data structures are set up, a process is started on each node containing data. These processes execute the node parts of the algorithms.)

Algorithm V-Control. (Vector-Scalar Multiplication.) Given a Command type distributed variable S, and the scalar c, direct other computational nodes to compute $\vec{x} = c\vec{b}$.

VC1: [Distribute scalar.] Write c to S.

VC2: [Wait.] Wait for S to acknowledge write.

End of Algorithm

Algorithm V-Node. (Vector-Scalar Multiplication.) Given a Command variable S, points $P_1 \ldots P_p$, containing vectors $\vec{x}$ and $\vec{b}$, compute x where $\vec{x} = c\vec{b}$.

VN1: [Get scalar.] Read c from S.

VN2: [Do uniprocessor algorithm.] Do algorithm V-Uniprocessor.

VN3: [Acknowledge.] Acknowledge reading from S.

End of Algorithm

The time required to distribute a command and acquire information about its completion in a system with n computational nodes, is asymptotically $\log_2 n$. For both the time is $2\log_2 nL$, where L is the message latency time.

Measure	*Parameterized*	*Example Timings*
Distribution	$2\log_2 NL$	24 mS
Operation	PF	1.6 mS
Total		26 mS
Total not including distribution		1.6 mS

Performance figures show that the time is dominated by the time to distribute the command to all the processors. It is, however, often possible to eliminate the command distribution delay by overlapping the execution of one command with the distribution of the next. Command distribution delay will not be included in the future.

5.2 Vector Inner Product

The dot product, $x = \vec{a} \cdot \vec{b}$, where x is a scalar, $\vec{a}$ and $\vec{b}$ are vectors with n elements, is defined to be: $x = \sum_{i=1}^{n} a_i b_i$

The multiprocessor algorithm first computes the dot product of the elements in each computational node and stores the result in a temporary. The second phase adds all the temporaries.

Phase 1: $T_j = \sum_{\text{point } i \text{ in node } j} a_i b_i$

Phase 2: $x = \sum_{\text{all } j} T_j$

The phase 1 operations are straightforward. All the data necessary for the operations in each computational node are already there. The time to complete phase 1 in each computational node is related to the quantity of points and, since all the nodes operate in parallel, the efficiency of the operation will approach 100%.

The phase 2 operations involve a distributed sum that is done by a Sum type distributed variable. The main program declares and initializes a variable of type Sum and then passes this variable to each of the computational nodes. Each time a distributed sum is done the nodes write a value into the Sum variable, and the main program reads the sum of all these values.

Algorithm I-Control. (Inner product.) Given a sum type distributed variable I, direct computation of an inner product.

IC1: [Start nodes running.] Run I-Node on nodes.

IC2: [Get answer.] Read answer from I.

End of Algorithm

Algorithm I-Node. (Inner product.) Given a set of points $P_1 ... P_p$, with vector elements for $\vec{a}$ and $\vec{b}$, help compute $x = \vec{a} \cdot \vec{b}$.

IN1: [Initialize.] Set $S \leftarrow 0.0$.

IN2: [Iterate on points.] Do IN3 and IN4 for $i = 1...p$.

IN3: [Local summation.] Set $S \leftarrow S + a_i b_i$.

IN4: [Write to distributed variable.] Write S to I.

End of Algorithm

Measure	*Parameterized*	*Example Timings*
Local summation	2PF	3.2 mS
Distributed summation	$\log_2 NL$	12 mS
Total		15 mS

5.3 Matrix-Vector Multiplication

The uniprocessor algorithm for calculating $\vec{x} = A\vec{b}$, where $\vec{x}$ is the unknown vector, $\vec{b}$ is a vector, and A is a matrix, is straightforward.

Algorithm M-Uniprocessor. (Matrix-vector multiplication.) Given a set of points, $P_1 ... P_p$, compute $\vec{x} = A\vec{b}$. (Initially $\vec{x} = 0$.)

MU1: [Iterate on arcs.] Do MU2 and MU3 for all i and j such that A_{ij} exists.

MU2: [Compute.] Set $x_i = x_i + b_i A_{ij}$.

MU3: [Symmetric arcs.] Set $x_j = x_j + b_j A_{ji}$.

End of Algorithm

The difficulty in programming this algorithm on a multiprocessor is that some of the point data structures are duplicated between computational nodes. Fortunately, the uses made of these duplicated points is straightforward, and the multiprocessor code is best described as a variation of the uniprocessor code.

The result of executing the uniprocessor code on a multiprocessor is correct except for duplicated boundary points. The vector elements represented by boundary points would be incorrect in each copy. The correct value for the vector elements is the sum of all the values in the duplicated points.

The multiprocessor algorithm executes the uniprocessor algorithm and then sums the boundary vector elements. The correct boundary vector elements are distributed to every duplicated point.

Algorithm M-Multiprocessor. (Matrix-vector multiplication.) Given a set of points, $P_1 \ldots P_p$, some of which are boundary points, and a set of arcs, $A_1 \ldots A_a$, compute $\vec{x} = A\vec{b}$.

MM1: [Do uniprocessor algorithm.] Do M-Uniprocessor.

MM2: [Iterate on boundary points.] Do MM3 for all i such that P_i is a boundary point.

MM3: [Summation output.] Write x_i to pntsum(P_i).

MM4: [Iterate on boundary points.] Do MM5 for all i such that P_i is a boundary point.

MM5: [Summation input.] Read from pntsum(P_i), put result in x_i.

End of Algorithm

The code above uses the new attribute called pntsum of the point structure to do a summation of the various partial sums associated with boundary vector values. The new definition of the point structure is shown below. When the mesh is created corresponding pntsum attributes must be initialized to the same instance of the distributed variable.

point data structure

attribute	*type*	
x	floating point	*vector element*
b	floating point	*vector element*
pntsum	type Sum distributed variable	*connects duplicate points*

Execution of the code illustrated is efficient, because the fraction of points requiring communication is small. Performance measures:

Measure	*Parameterized*	*Example Timings*
Multiply-add	2AF	48 mS
Communication	$4\sqrt{P}M$	11 mS
Latency	L	1 mS
Total		60 mS

5.4 Back Substitution

Concurrency in back substitution is derived mostly from the block numbering described earlier. Recall the conventional algorithm for solving $L\vec{x} = \vec{b}$:

$$x_j = \frac{b_j - \sum_{i=1}^{j-1} x_i L_{ij}}{L_{jj}}$$

The block numbers assigned to points in an earlier section partition the points into sets that can be computed concurrently. The multiprocessor iterates on block numbers; first all elements of $\vec{x}$ with block number 1 are computed, then block number 2, etc.

Consider computing x_j for a point with block number 1. By the assignment of block numbers, no point has an arc to another point with the same block number. Since block number 1 is the lowest block number, all L_{ij} arcs have the property that i > j, and the L_{ij} value is zero since it is above the main diagonal. The computation of these x_j is simply $x_j = \frac{b_j}{L_{jj}}$. These computations are independent.

Now consider the computation of the x_j with block number n > 1. Again, no arc connects to a point with the same block number. Some of the L_{ij} arcs connect to points with a higher block number, but the value of these arcs is zero. The rest of the arcs connect to points with a lower block number, the xj values for these points are required, but their values have already been computed. These computations are independent.

Algorithm B-Node. (Back Substitution.) Given a set of points, $P_1 ... P_p$, and a set of arcs, $A_1 ... A_a$, solve $A\overline{x} = \overline{b}$ for $\overline{x}$. (Initially $\overline{x}=0$).

BN1: [Iterate on block numbers.] Do BN2...BN11 for n=1...B.

BN2: [Iterate on selected arcs.] Do BN3 for all i and j such that A_{ij} exists, $i \neq j$, and P_i is in block n.

BN3: [Compute.] Set $x_i \leftarrow x_i + L_{ij}x_j$.

BN4: [Iterate on boundary arcs.] Do BN5 and BN6 for all i such that P_i is a duplicated point and P_i is in block n.

BN5: [Summation output.] Write $b_i - x_i$ to pntsum(P_i).

BN6: [Summation output.] Write L_{ii} to pntsum(P_i).

BN7: [Iterate on selected arcs.] Do BN8...BN11 for all i and j such that A_{ij} exists, $i \neq j$, and P_i is in block n.

BN8: [Check for boundary point.] If P_i is not a duplicated point, set $x_i \leftarrow \frac{b_i - x_i}{L_{ij}}$ and skip BN9...BN11 and continue the iteration.

BN9: [Summation input.] Read from pntsum(P_i), put result in A.

BN10: [Summation input.] Read from pntsum(P_i), put result in B.

BN11: [Compute.] Set $x_i \leftarrow \frac{A}{B}$.

End of Algorithm

The algorithms restructures the computation of x_j as shown below. When a vector element x_j is duplicated between several nodes, the numerator and denominator of the expression for x_j are computed separately based on the (locally consistant) data in each node. The numerators and denominators are then added and divided (redundantly) on each node.

$$x_j = \frac{\sum_{\text{all nodes}} [b_j - \sum_{\text{local}} x_i L_{ij}]}{\sum_{\text{all nodes}} [L_{jj}]}$$

BN2 and BN3 compute $\sum_{\text{local}} x_i L_{ij}$,

BN4...BN6 start the two distributed summations of $b_j - \sum_{\text{local}} x_i L_{ij}$ and L_{jj},

BN7...BN10 complete the distributed summation, and

BN11 computes the quotient.

Measure	*Parameterized*	*Example Timings*
Summation	$2\frac{A}{B}F$	2.4 mS
Communication	$\frac{4\sqrt{P}}{B}M$	560 uS
Latency for above	L	1 mS
Definition	$2\frac{P}{B}F$	160 uS
Total		4.1 mS
Total 20 iterations		82 mS

5.5 The Incomplete Cholesky Decomposition

The Cholesky decomposition of a matrix A is the matrix L where $A = LL^t$. L is derived by equating the elements of A with the product of a row of L and a column of L^t. The incomplete Cholesky decomposition of a sparse matrix A has the same sparsity pattern as the matrix A by definition.

Like back substitution, the incomplete Cholesky involves an iteration; unlike back substitution, the incomplete Cholesky updates values associated with arcs, not points.

A new notation is used to describe the L_{ij} arcs. In this algorithm, L_{ij} represents an L arc between a point in block i and another point in block j. There may be more than one arc satisfying this description (say there are m arcs), and these arcs are named $L_{ij}^1...L_{ij}^m$.

During the n'th iteration in the computation of the incomplete Cholesky, the $L_{ij}^1...L_{ij}^m$ elements are computed where $n = i+j$. (Iteration 1 does not exist, iterations are numbered from 2, where L_{11} is computed.) To illustrate, the table below shows the L_{ij} elements computed during the first few iterations:

Iteration number	L_{ij} computed
2	L_{11} (i.e. $L_{11}^{1} ... L_{11}^{m}$)
3	L_{21}
4	L_{31} L_{22}
5	L_{41} L_{32}
6	L_{51} L_{42} L_{33}
7	L_{61} L_{52} L_{43}

Recall the two formulas for computing the Cholesky:

$$L_{jj} = \sqrt{A_{jj} - \sum_{n=1}^{j-1} L_{jn}^{2}}$$

$$L_{ij} = \frac{A_{ij} - \sum_{n=1}^{j-1} L_{in} L_{jn}}{L_{jj}}$$

Note that the iterative order described above is valid for the incomplete Cholesky; the computation of any L_{ij} only involves other L_{kl}'s where $i+j > k+l$.

The summations in the Cholesky equations have a graph interpretation. The summation to calculate the L for an arc between two endpoints is the product along all paths through an intermediate point. Specifically, the product is used of any two arcs with tails on the endpoints of the arc and heads on an intermediate point. The calculation of L_{jj} elements, where the endpoints are the same, uses the square of any arc from the endpoint to any other point.

Figure 7 illustrates three points and their L arcs. The only product in figure 7 is the product of L_{21} and L_{31}; the product is needed to commute L_{32}. The task is to form all products of the L values that have tails at a common point. The product is added to the L arc between the two points that are at the heads of the two original L arcs.

Note the following about the computations for an L_{ij} arc: All the L_{in} L_{jn} pairs that must be multiplied are resident on the same computational node, and there is a (duplicated) copy of the L_{ij} arc on each of these computational nodes.

A variation of the back substitution strategy works acceptably. Sums are formed conventionally for all redundant copies of an arc. Following this a distributed sum is formed of the quantities $A_{ij} - \sum$ and L_{jj}. These sums are distributed back to all the duplicated arcs, which do the division.

The algorithm requires a temporary in the point data structure, called L_{jj}. Also required is a Sum type distributed variable connecting arcs between two duplicated points.

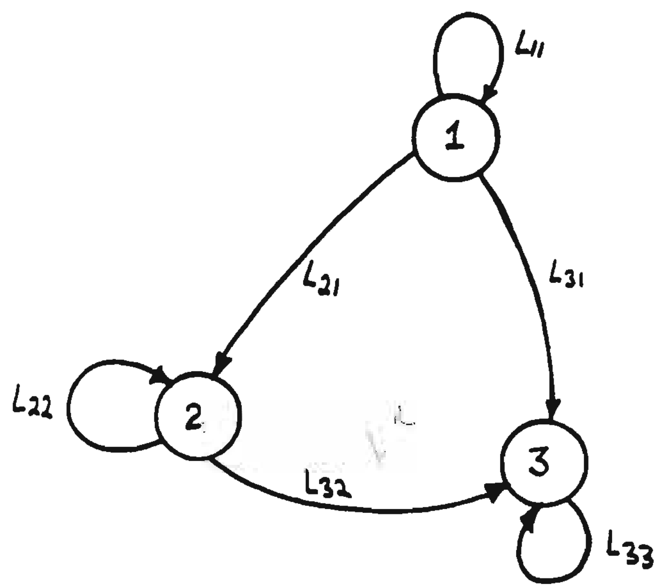

Figure 7: Block Structure for Incomplete Cholesky

point data structure

attribute	*type*	
x	floating point	*vector element*
b	floating point	*vector element*
pntsum	type Sum distributed variable	*connects duplicate points*
L_{jj}	floating point	*temporary*

arc data structure

attribute	*type*	
head	pointer to point	*point at head of arc*
tail	pointer to point	*point at tail of arc*
a	floating point	*matrix element*
l	floating point	*lower triangular matrix element*
arcsum	type Sum distributed variable	*connects arcs between duplicate arcs*

Algorithm C-Node. (Incomplete Cholesky.) Given a set of points $P_1 \ldots P_p$, solve $LL^t = A$ for L. (Initially all $L_{ij} = 0$.)

CN1: [Iterate on block numbers.] Do CN2...CN12 for n=1...2B.

CN2: [Iterate on selected arcs.] Do CN3...CN7 for all i and j such that L_{ij} exists and $block(P_i)+block(P_j) = n$.

CN3: [Iterate on paths of length 2.] Do CN4 for all k where both L_{ki} and L_{kj} exist.

CN4: [Compute.] Set $L_{ij} \leftarrow L_{ij}+L_{ki}L_{kj}$.

CN5: [Check for boundary point.] If either P_i or P_j is not a boundary point, skip CN6 and CN7 and continue the iteration.

CN6: [Summation output.] Write $A_{ij}-L_{ij}$ to $arcsum(L_{ij})$.

CN7: [Summation output.] If $i = j$ write L_{jj} to $arcsum(L_{jj})$.

CN8: [Iterate on selected arcs.] Do CN9...CN12 for all i and j such that L_{ij} exists and $block(P_i)+block(P_j) = n$.

CN9: [Non-boundary arcs.] If either P_i or P_j is not a boundary point, set $X \leftarrow A_{ij}-L_{ij}$, and if $i = j$ set $Y = L_{jj}$. Go to CN12.

CN10: [Boundary arcs.] Read from $arcsum(L_{ij})$, put result in X.

CN11: [Arcs of form L_{jj}.] If $i = j$ read from $arcsum(L_{jj})$ put result in Y.

CN12: [Compute.] If $i = j$ then set $L_{ij} \leftarrow \sqrt{X}$, otherwise set $L_{ij} \leftarrow \frac{X}{Y}$.

End of Algorithm

Measure	*Parameterized*	*Example Timings*
Local products (CN2...CN4)	$7\frac{A}{B}F$	8.4 mS
Communication (CN5...CN7)	$\frac{8\sqrt{P}}{B}M$	110 uS
Latency for CN6 and CN7 to CN10 and CN11	L	1 mS
Computation (CN8...CN12)	$\frac{A}{B}F$	1.2 mS
Total		11 mS
Total 2B (40) iterations		470 mS

6. Conclusions

It is informative to measure the performance of a multiprocessor on an entire ICCG iteration. The table below summarizes the ICCG algorithm in terms of the matrix operations discussed earlier.

The ICCG procedure is iterative, requiring $o(n^{2})$ iterations. When ICCG is used in a adaptive mesh, changes in the solution are minimal as the mesh is refined. In the table below, the number of iterations, n, is a constant 5 for the ICCG.

Operation	*Quantity*	*Time*
Vector-Scalar Multiplication	5n = 25	40 mS
Vector Inner Product	1n = 5	60 mS
Matrix-Vector Multiplication	2n = 10	600 mS
Back Substitution	2n = 10	820 mS
Incomplete Cholesky	1	470 mS
Total		2.0 S

It is interesting to observe the correlations between changes in the hardware parameters (F, M, L) and the overall rate of adaptive iterations. In the table below, the total time per iteration (2.0 S in the above table) was expressed as a function: T(F, M, L). The table below shows derivatives with respect to F, M, and L. Specifically, the coefficient of parameter X is $\frac{X}{T(F,M,L)} \frac{\partial}{\partial X} T(F,M,L)$ (Instruction executions, the coefficients of I were not evaluated in this document.)

Parameter	*Correlation coefficient*
I (instruction executions)	not measured (0)
F (floating point)	.71
M (messages)	.13
L (sequential messages)	.15

The table indicates that the performance of the machine on the whole problem is most sensitive to changes in the floating point rate (F). A 1% increase in the floating point time would result in a .71% increase in the execution time on this problem. A 1% increase of M or L would slow down the machine by .13% and .15% respectively.

The example hardware running this application is somewhat overdesigned for communication. This conclusion is based on the assumption that the incremental cost associated with changing the speed of communication and floating point is about equal, whereas the application is severely floating point limited. Were these costs known more accurately, a mathematical optimization problem could be set up. This conclusion is also reasonable considering that the example hardware was intended for general purpose programming, whereas matrix manipulations are unusually computation intensive.

The table below describes the algorithms discussed here with the conventional multiprocessor measures. The total floating point is the coefficient of F in the function T(F, M, L).

Measure	*Value*
Total Floating Point per Adaptive Iteration	2.9 G
Time per Iteration	2.0 S
Million Floating Point Operations per Second (MFLOPs)	1450
Speedup	2969
Efficiency	72%

A comparison of the program discussed here with similar programs for a conventional supercomputer (CRAY-1) is helpful. Programs for both computers require the programmer to design bottom-up from the hardware structure of the computer. The program described here requires the programmer to identify concurrency; CRAY-1 code requires the programmer to explicitly identify the vectors in the problem.

This multiprocessor approach has the following advantages over a vector processor approach:

(1) With appropriate selection of hardware, the $\frac{\text{cost}}{\text{performance}}$ ratio can be less, or the speed can be greater, or both.

(2) The finite element mesh can be arbitrary. A vectorized program would restrict the mesh in various ways, such as requiring the mesh to be rectangular.

(3) Multiprocessors can be constructed over a wider range of sizes than a vector processor.

Programs for vector processors are irregular in their performance. Vectorized algorithms run fast, and the rest run at the much slower scalar speed. One algorithm discussed here, computation of the incomplete Cholesky, cannot be vectorized. We believe this is because the vector concept is less general than the MIMD concept. Vector processors win, however, because they can be programmed as a scalar processor when necessary, whereas a MIMD machine can never be programmed as a single computer.

MIMD multiprocessors are widely recognized as having a superior computing potential, but examples of interesting programs have been lacking. This document presents one example of an interesting program; indeed finite elements is an important application area for present supercomputers. If a dozen documents of this type, each addressing a different problem, could be produced, general acceptance of MIMD parallel processors might follow.

Erik P. DeBenedictis

E. P. DeBenedictis

D. N. Shenton

Att.
References

7. References

[Cendes ??], "Notes on a ICCG Matrix Solution Package", Z. J. Cendes, unpublished document, Carnegie-Mellon University.

[Cendes 82], "Magnetic Field Computation Using Delaunay Triangulation and Complementary Finite Element Methods", Z. J. Cendes, D. Shenton, H. Shahnasser, journal article.

[DeBenedictis 84], "A Distributed Switch for Multiprocessor Computing Systems", E. DeBenedictis, in preparation, Bell Telephone Laboratories.

[Jordan 82], "A Guide to Parallel Computation and Some Cray-1 Experiences", T. L. Jordan, in "Parallel Computation", Garry Rodrigue, ed., Academic Press, 1982.