

**DISTRIBUTED FINITE ELEMENT ANALYSIS  
USING A TRANSPUTER NETWORK**

**Final Report  
July 1989**

**NASA SBIR #NAS3-25422**

**SPARTA**

*James Watson  
James Favenesi  
Albert Danial  
Joseph Tombrello  
Dabby Yang  
Brian Reynolds  
Ronald Turrentine*

ORIGINAL CONTAINS  
COPY

**Rensselaer Polytechnic Institute**

*Mark Shephard  
Peggy Baehmann*

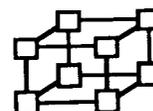
**Prepared For  
NASA LEWIS RESEARCH CENTER  
CLEVELAND, OH 44135**

**Advanced Data Processing Laboratory  
SPARTA, Inc.**

**4901 Corporate Drive  
Huntsville, AL 35805  
(205) 837-5200**



**SPARTA, INC.**



**MULTIPROCESSOR  
DEVELOPMENT  
GROUP**

## TABLE OF CONTENTS

Executive Summary	1
I Introduction	4
II System Design	6
Hardware	6
Configuration	7
Communication	7
III Distributed Finite Element Method	13
The Sparse Solver	14
The Profiler	15
Parallel Assembly of the Global Stiffness Matrix	16
Boundary Conditions	18
Parallel Jacobi-Conditioned Conjugate Gradient Solver	18
Effects of Bandwidth	19
The Out-of-Core Solver	24
The Substructures Method	25
Steps to Solution for Substructuring on Transputers	27
Secondary Storage Hardware Considerations	30
IV Pre- And Post-Processing	35
Finite Quadtree	35
NASTRAN Interface	36
XPOST	36
Adaptive Analysis	37
V XPFEM: A Transputer Based Finite Element Workstation	50
VI Conclusions and Recommendations	65
Enhancements to Pre-Processor	65
Improved Boundary Conditions	65
Three-Dimensional Mesh Generation	66
Parallel Mesh Generation	66
Node Renumbering	66
Enhancements to the Solver	67
Increasing Solver Capacity	67
Fine-Tuning the Solver	68
Enhancements to Post-Processing	70
Plasticity and Zero Stress Contours	70
Parallel Contour Computation	70
Three-Dimensional Stress Contours	70
Additional Features	71

VII	References .....	73
	Appendix A: XPFEM User's Manual .....	75
	XPFEM Options .....	77
	New Problem .....	77
	Recover Old Problem .....	77
	Problem Type .....	77
	Workspace Grid .....	77
	FINITE QUADTREE Geometric Modeler .....	77
	Material Properties .....	78
	Analysis Attributes .....	78
	FINITE QUADTREE Mesh Generation .....	80
	Output for Analysis .....	80
	Read NASTRAN File .....	81
	2D/3D Solver .....	81
	2D Post Processor .....	82
	3D Post Processor .....	84
	Exit .....	85
	Converting Binary Files to Text Files and Vice Versa .....	86
	Binary Input Files to Text Input Files .....	87
	Text Input Files to Binary Input Files .....	88
	Binary Solution Files to Text Solution Files .....	89
	Problems with EXE convert.tsr .....	90
	Appendix B: Solver Limitations .....	91
	Primary Memory Parameters .....	91
	Memory Usage as a Function of the Primary Memory Parameters .....	93
	Reconfiguring the Solver for Larger Problems .....	93
	Minimum Problem Size .....	94
	Appendix C: Jacobi-Conditioned Conjugate Gradient Algorithm .....	95
	Appendix D: XPDOS User's Manual .....	98
	XPDOS Command Shell .....	98
	Wildcards .....	99
	Listing Directories .....	99
	Deleting Files .....	100
	Copying Files .....	100
	Renaming and Moving Files and Subdirectories .....	101
	Creating Directories .....	101
	Removing Directories .....	102
	Changing Default Directory .....	102
	Changing Default Disk .....	103
	Displaying Free Space .....	103
	Displaying a File .....	103
	Obtaining Help .....	103
	Transferring Files between XPDOS and MS-DOS .....	103
	Executing MS-DOS Commands .....	104

Exiting from XPDOS .....	105
Appendix E: XPDOS Disk Library Routines and Error Codes .....	106
System Routines .....	106
Directory Access Routines .....	107
File Opening .....	110
General Routines .....	111
Direct Access .....	112
Binary Access .....	112
Status Codes .....	117
Appendix F: XPGRAPHICS User's Manual .....	124
System Routines .....	124
Color Routines .....	124
Point, Line, Polygon and Curve Routines .....	126
Screen and Window Routines .....	129
Text Routines .....	134
CRT Routines .....	137
Error Codes .....	139
Appendix G: Keywords Recognized by the NASTRAN Interface Module .....	140
Appendix H: Parallel Mesh Generation with FINITE QUADTREE .....	141

## LIST OF FIGURES

Figure 1, Deliverable Hardware .....	10
Figure 2, XPFEM Communication Scheme .....	11
Figure 3, Matrix-Vector Multiply Benchmark .....	12
Figure 4, Sample FE Model and Global Stiffness Matrix .....	21
Figure 5, P-Vector Overlap for Matrix-Vector Multiplication .....	22
Figure 6, P-Vector Dependence between Processors .....	23
Figure 7, Out-of-Core Configuration .....	31
Figure 8, A 167,000 Degree of Freedom Model .....	32
Figure 9, Equations for Partitioning Large FE Problems .....	33
Figure 10, The Large Model Condensed to Superelements .....	34
Figure 11, Quadtree Theory: Model Sectioned into Quadrants.....	39
Figure 12, Geometric Model .....	40
Figure 13, Boundary Discretization .....	41
Figure 14, Quadrants .....	42
Figure 15, Repositioning Quad Nodes Using Smoothing .....	43
Figure 16, XPFEM Input Files .....	44
Figure 17, 2-D Post-Processing .....	45
Figure 18, 3-D Post-Processing .....	46
Figure 19, Adaptive Analysis Algorithm Implemented in XPFEM .....	47
Figure 20, Bubble Functions .....	48

Figure 21, Error Analysis .....	49
Figure 22, Main Menu .....	53
Figure 23, Geometric Model with Mesh Parameters, Grid and Menu .....	54
Figure 24, Mesh with Refinements .....	55
Figure 25, Time to Assemble Global Stiffness Matrix .....	56
Figure 26, Time to Solve System of Finite Element Equations .....	57
Figure 27, Total Solution Time .....	58
Figure 28, Speed-Up for the Turbine Blade .....	59
Figure 29, Space Shuttle Main Engine Turbine Blade .....	60
Figure 30, Deformed and Undeformed SSME Turbine Blade .....	61
Figure 31, Stress Contours in Ring .....	62
Figure 32, Principal Stresses in Ring .....	63
Figure 33, Adaptive Analysis .....	64
Figure 34, Commercial XPFEM Workstation .....	72
Figure H.1, Twelve Colors Identify Independent Quadrants .....	144
Figure H.2, Minimum of Eight Colors Needed .....	145

## **EXECUTIVE SUMMARY**

The principal objective of this research effort was to demonstrate the extraordinary cost effective acceleration of finite element structural analysis problems using a transputer-based parallel processing network. This objective has been accomplished in the form of a commercially viable parallel processing workstation. The workstation is a desktop size, low-maintenance computing unit capable of supercomputer performance yet costs two orders of magnitude less.

To achieve the principal research objective, a transputer based structural analysis workstation termed XPFEM was implemented with linear static structural analysis capabilities resembling commercially available NASTRAN. Finite element model files, generated using the on-line preprocessing module or external preprocessing packages, are downloaded to a network of 32 transputers for accelerated solution. The system currently executes at about one third Cray X-MP24 speed but additional acceleration appears likely. For the NASA selected demonstration problem of a Space Shuttle main engine turbine blade model with about 1500 nodes and 4500 independent degrees of freedom, the Cray X-MP24 required 23.9 seconds to obtain a solution while the transputer network, operated from an IBM PC-AT compatible host computer, required 71.7 seconds. Consequently, the \$80,000 transputer network demonstrated a cost-performance ratio about 60 times better than the \$15,000,000 Cray X-MP24 system.

A number of significant developments were required to achieve the demonstration objective. These developments included:

### **Parallel Hardware System**

A network of thirty-two densely interconnected transputers, a large screen color graphics system and a network mass storage system were designed, separately implemented and integrated into an inexpensive host computer. The complete system fits on a desktop, yet delivers the computational power of a supercomputer.

## **Parallel Sparse Finite Element Solver**

A parallel, sparse implementation of the Jacobi-conditioned Conjugate Gradient (JCG) algorithm was implemented for solution of the finite element equations on the transputer network. The memory-efficient algorithm enables problems with over 20,000 degrees of freedom to be solved in the 32 Mbytes of distributed core memory. In addition, an out-of-core solver has also been defined for larger problems which cannot reside in the collective core memory of the transputer network.

## **Adaptive Analysis**

Adaptive analysis is an automated mesh refinement procedure for reducing finite element mesh discretization errors to a given tolerance. It consists of a series of solutions to the finite element problem, each with successively more refined meshes in regions of large error. The adaptive analysis capability implemented as part of this research effort has been demonstrated to converge efficiently on two-dimensional models. A three-dimensional capability would involve a similar implementation path. A suitable 3D mesh generator is in development and a 3D error criterion for driving the solution convergence is available. The three-dimensional capability was not defined as a part of this research as no new insight was apparent that would justify its significant implementation time.

## **Pre/Post Processing**

Graphic visualization of the finite element model, boundary conditions and applied loads, and the resulting deformed model and stresses are invaluable to engineering analysis. A graphics presentation capability has been implemented for this purpose. The graphics system permits visual, interactive creation and manipulation of models during preprocessing, shows the discretization of the model into elements during mesh generation, and displays the deformed model, stress contours, and principal stresses during post processing. Three-dimensional structures can be analyzed but model development and 3D stress contouring must be performed with third party software.

## Validation

Accuracy of finite element software implemented on XPFEM was verified with three tests:

1. Element stiffness matrices computed by XPFEM were compared to the matrices produced by COSMIC NASTRAN.  
Result: stiffness terms match COSMIC NASTRAN to five significant figures.
2. The displacement solution for a NASA-provided 3D turbine blade problem, as computed by a Cray X-MP24 running COSMIC NASTRAN, was compared term-by-term to the solution produced by XPFEM.  
Result: solutions are identical to five significant figures when the tolerance on XPFEM's iterative solver is  $1.0E-7$  or lower.
3. Rectangular models were deformed along one axis. The amount of lateral contraction computed by XPFEM was compared to the value predicted by linear elastic theory and to the value produced by COSMIC NASTRAN.  
Result: Lateral contraction agreeing with Poisson's ratio was computed by both XPFEM and COSMIC NASTRAN.

These tests do not rigorously prove the accuracy of XPFEM solutions, but do give sufficient confidence that the basic element generation, stiffness matrix assembly and solution algorithms function properly.

## I. INTRODUCTION

The Phase I feasibility study of a transputer based finite element solver concluded that a network of transputers, coupled with efficient parallel finite element code, could result in an engineering analysis tool of tremendous power [1]. The objective of this Phase II research effort was to design, develop, and implement a transputer-based finite element workstation with the extraordinary performance predicted by the Phase I study. This objective was achieved.

The performance of local-memory distributed processors depends to a large extent on how efficiently the processors can communicate. For this reason, the first task in the development of the finite element workstation involved designing a network that would allow the processors to communicate optimally. The other pieces of the workstation, namely the host interface, high-resolution graphics and fast external storage, were then integrated into the network to form a complete hardware system. The hardware in XPFEM and its arrangement is described in Section II, System Design. This section also explains the transputer communication scheme and how it evolved.

Much research on parallel finite element solvers has been done in recent years [2-5]. This research provided significant insight and guidance to the design of XPFEM; however, many of the proposed parallel solvers had limitations that precludes their use in a viable engineering workstation. The goal for XPFEM was to create a parallel solver that was fast, efficient and could solve large finite element problems. Several new methods had to be developed to achieve this goal. Section III, Distributed Finite Element Method, consists of two parts: *The Sparse Solver* and *The Out-of-Core Solver*. *The Sparse Solver* describes algorithms developed for and implemented in XPFEM which subdivide a finite element problem, assemble the stiffness matrix in parallel without communication, and solve the resulting finite element equations in-core using a conjugate gradient solver. *The Out-of-Core Solver* discusses a concept for implementing an out-of-core capability into XPFEM.

A finite element solver is of limited value to a structural analyst unless he or she has a convenient means of entering and modifying problems, and interpreting the results. Section IV, Pre/Post Processing, describes the mesh generator, NASTRAN

interface and post processor modules which provide input, output and interpretation capabilities.

Section V describes the integrated XPFEM workstation. It contains screens from a sample work session including model creation, mesh generation, and post-processing.

Many new ideas occurred to the researchers during the development of XPFEM. They include such diverse subjects as faster and more efficient parallel solvers, better storage schemes, methods of implementing advanced structural analysis features, and parallel pre- and post-processing. Although beyond the scope of this research effort, these ideas would be invaluable in a commercial structural analysis workstation. They are discussed in Section 6, Conclusions and Recommendations.

The eight appendices give detailed information on the topics presented in the report. They contain User's Manuals for the XPFEM workstation and the XPDOS transputer disk operating system, a description of the Jacobi-conditioned conjugate gradient solver, limitations of the solver, descriptions and interfaces of XPDOS and XPGRAPHICS, the graphics library for XPFEM, a report of a parallel mesh generation method, and limitations of the NASTRAN interface.

## II. SYSTEM DESIGN

Designers of parallel programs must thoroughly consider all aspects of system hardware, configuration and communication details as preliminaries to developing efficient parallel software. Significant effort was devoted to each of these preliminaries during this research project.

### **Hardware**

Figure 1 shows the hardware components of the delivered prototype transputer based workstation. The original design concept proposed a system of forty transputers but due to innovations in the network topology, a system of thirty-two T800-20 transputers configured in two tetrahedra was determined to be more cost-effective. Each transputer has four 20 Mbit/second bi-directional links, operates at 20 MHz, is rated at 1.5 Mflops, and has 1 Mbyte of RAM. The thirty-two transputers were mounted on eight PC-XT/AT sized plug-in cards with four transputers per card.

High resolution graphics capabilities were also incorporated into the workstation. These graphics capabilities were developed using INMOS B408 and B409 transputer modules as a graphics unit. The B408 module functions as a drawing and image storage unit. It contains a T800 transputer and 2.25 MBytes RAM and is linked to the B409 module via a pixel bus. The B409 module functions as a display and synchronization driver. The combined B408/B409 modules are capable of resolutions of 512 x 512 with five pages of memory, 768 x 768 with two pages of memory or 1024 x 768 with one page of memory.

Mass storage was made available to the transputer network via a pair of SCSI disk drives. This capability was necessary to implement effective out-of-core solution techniques. The SCSI drives were designed by a third party OEM exclusively for this project. They are 100 MByte Winchester Conner hard disks mounted on one PC-AT plug-in card. The disks are controlled by T222 transputers and interface to the network through transputer links. Transfer times are in the range of 0.25 MBytes/sec. A SPARTA-developed transputer disk operating system permits directory, file, and read/write operations to be performed with convenient routines (Appendices D and E).

## Configuration

Three different configurations--pipeline, doubly re-entrant grid, and linked tetrahedra--were considered as viable candidates for the workstation network. The pipeline was of interest because of the simplicity in its communication scheme. The merits and potential of the doubly re-entrant grid were determined during the Phase I feasibility study [1]. The tetrahedral connection scheme, first suggested by Rodgers [6], evolved as a viable configuration for iterative solution methods. Results from matrix-vector multiplication benchmarks indicated that the linked tetrahedra concept provided potential expandability, communication flexibility, optimized link usage, and resulted in the shortest average communication path for global scalar summations and vector exchanges, the dominant communication operations in conjugate gradient-based iterative solution methods. Because of these results, the tetrahedral scheme was chosen as the workstation network topology.

A schematic of the tetrahedral configuration is shown in Figure 2. In this configuration transputers are linked into clusters of four. Each transputer is attached to the remaining three transputers in a cluster using three of its integrated links. The remaining free link on each transputer is used to join clusters into tetrahedra with four clusters constructing one tetrahedron. At each vertex of a tetrahedron one transputer exists with a free link. These free links permit multiple tetrahedra to be networked, thus providing network expandability. As indicated by Figure 2, the workstation network consists of two interconnected sixteen-processor tetrahedra.

Results from matrix-vector multiplications performed on the tetrahedral network are displayed in Figure 3. These results show that the network is capable of sustaining 35.5 MFlops during matrix-vector multiplication.

## Communication

Conjugate gradient-based iterative techniques for solving linear systems are dominated by matrix operations such as matrix-vector multiplies and dot products. Due to global scalar summations and global vector exchanges, local-memory processing networks require significant communications to perform matrix operations. These communication requirements impose a system overhead that must be minimized to achieve maximum speed-up. The communication scheme developed for the

transputer based workstation was coordinated with the system configuration and appears to provide extremely efficient global summations while maintaining flexibility for improvements and expandability.

The communication scheme as implemented on the two tetrahedra network is outlined in Figure 2. This design, which requires five communication exchanges for global scalar summation, was the result of two revisions to earlier methods which used dynamic switching. Dynamic switching involves the use of specialized link hardware to change the link connections while the processors are running. It was thought to have advantages over a static network because messages between widely separated processors could be routed in a single step by directly joining the two remote processors whenever they needed to communicate.

The dynamic communication scheme consisted of three static exchanges among transputer clusters on the same tetrahedron and one dynamic exchange between corresponding transputers on different tetrahedra. A series of INMOS C004 link switches were used to dynamically alter the link connections at the vertices of the tetrahedra. The dynamic scheme was implemented but resulted in slow matrix-vector multiplication due to the delays needed to prepare each reconfigure. These results, and predictions from INMOS representatives that future transputers would be able to route messages in static configurations much faster than current transputers [7], led to a revision of the communications code so that only static exchanges took place.

The first revision needed seven communications steps to find global sums. It used the unmodified configuration with only one link between adjacent tetrahedra. This communication plan was implemented and used to achieve an early prototype distributed JCG solver on the two tetrahedra network; however, the seven communication exchanges at each iteration of the solution process resulted in an average computation-to-communication ratio of 1:2.

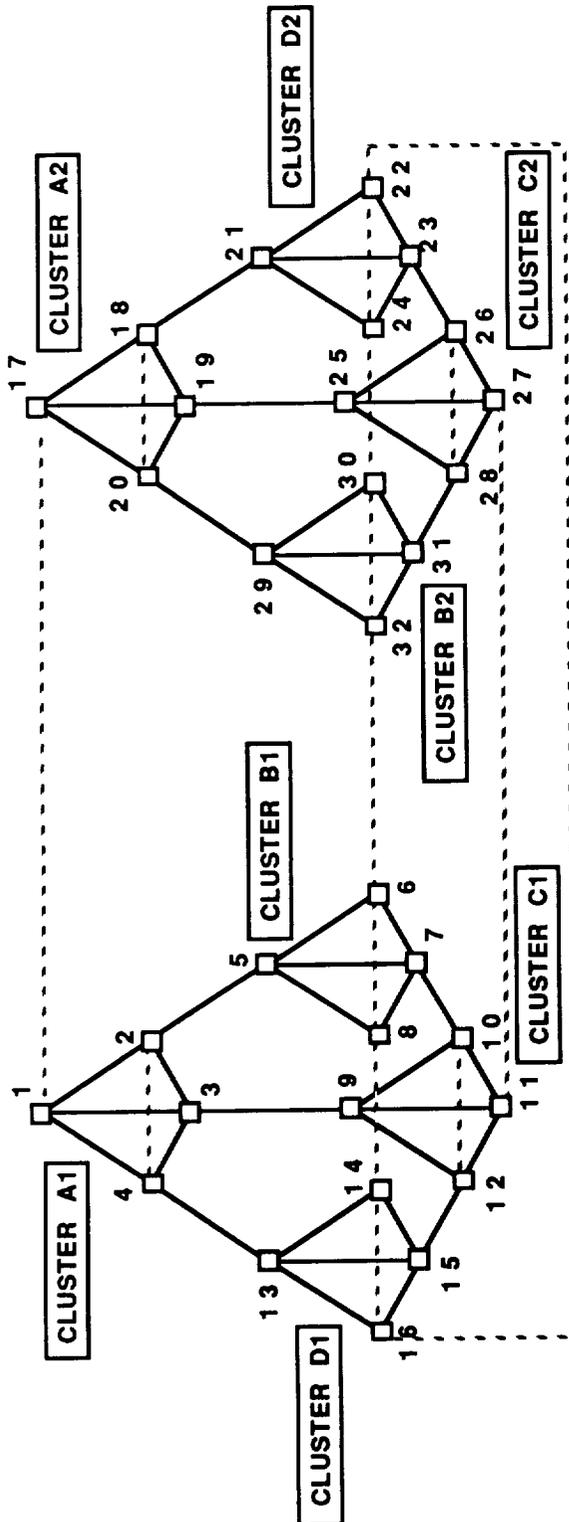
The final five-step communication scheme was a hybrid of the dynamic switching and seven-step schemes. Like the dynamic switching scheme, it fully utilizes all links in the network, including all four links of the vertex transputers of the tetrahedra. Like the seven-step scheme, all links are statically configured. Using the five-step scheme and the JCG algorithm, the average computation-to-communication ratio increased to 2:1.

The five step global communication scheme works equally well for vectors as well as scalars. However, it was observed during the development of the solver that global vector communication is superfluous if the finite element model bandwidth does not extend across the full matrix. Since banded matrices are the norm in finite elements, *local* rather than *global* vector exchange methods were investigated. An interesting result surfaced: pipeline communication between nearest neighbors is much faster than a global vector exchange. The number of communication steps for vector exchanges now becomes a function of the bandwidth and differs from problem to problem--tightly banded models will need fewer vector communication steps than poorly banded models. For reasonably banded problems, local vector exchanges on a pipeline increased the average computation-to-communication ratio to 8:1. A fortunate consequence of the tetrahedral topology is that it contains a pipeline within it, so no hardware modifications were necessary to gain the benefits of local pipeline communication.

Two primary conclusions were reached during the communications research. Firstly, as mentioned above, the experiences at SPARTA suggest that static, rather than dynamic, configurations result in a more efficient communication design. Secondly, local pipeline communications are more effective than tetrahedral communications when exchanging vectors to adjacent or nearly adjacent transputers.

<b>Component Description</b>	<b>Quantity</b>
<b>INMOS IMS B403-3 T800 Transputer (20 Mhz - 1Mbyte - 3 cycle DRAM)</b>	<b>32</b>
<b>INMOS INS B008 Motherboard</b>	<b>10</b>
<b>INMOS IMS B405-3A T800 Transputer with 8 Mbyte DIT module</b>	<b>1</b>
<b>CSA SCSI Winchester Drive Control</b>	<b>2</b>
<b>100 Mbyte SCSI Winchester Disk</b>	<b>2</b>
<b>INMOS IMS B403-3 T800 Graphics Tram</b>	<b>1</b>
<b>INMOS IMS B409 Graphics Controller</b>	<b>1</b>
<b>NEC Multisync II Color Monitor</b>	<b>1</b>
<b>Expansion Chassis</b>	<b>2</b>

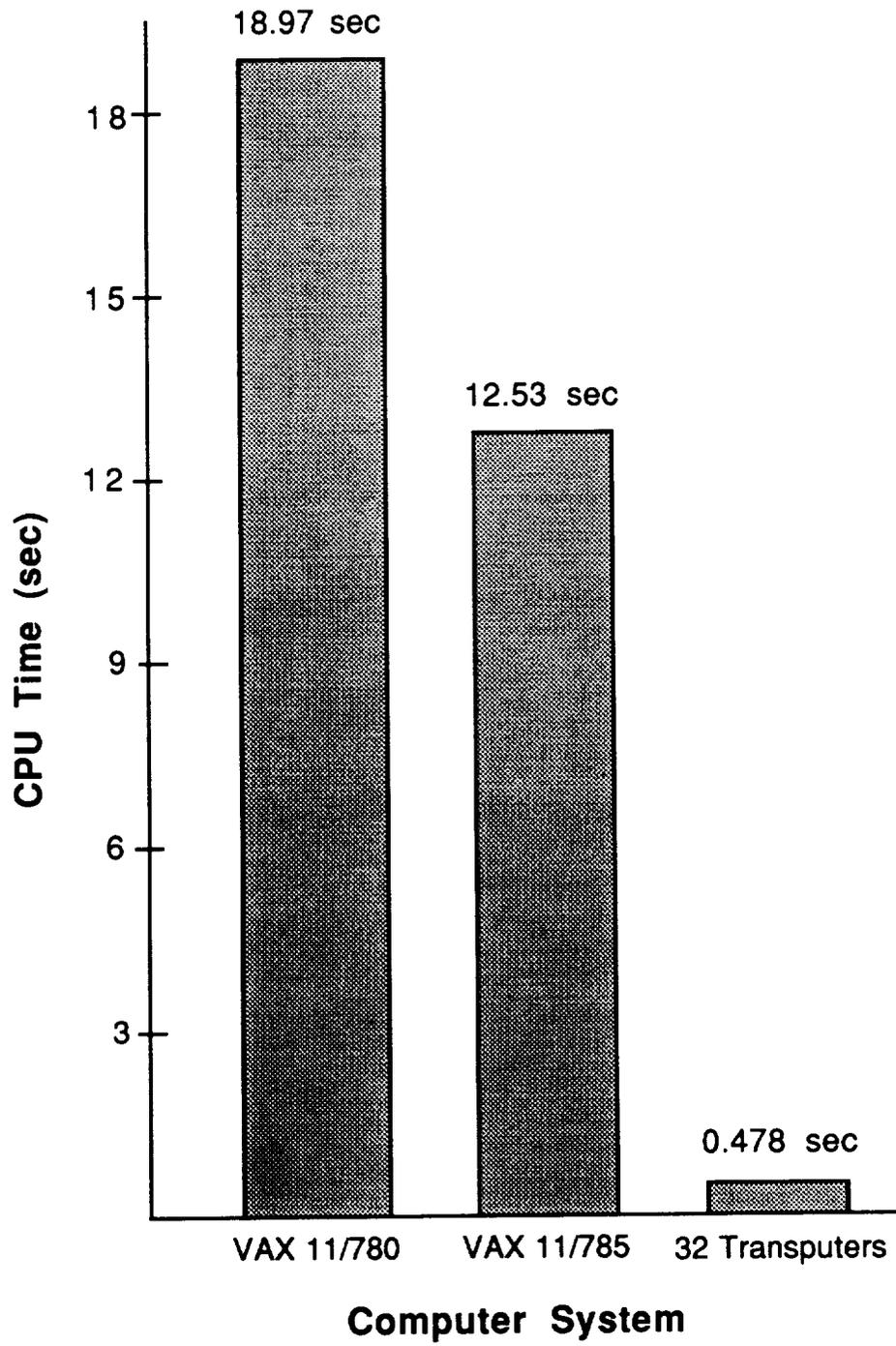
**FIGURE 1**  
**DELIVERABLE HARDWARE**



GLOBAL SCALAR SUMMATIONS	GLOBAL VECTOR DISTRIBUTIONS
1.) CLUSTER SUM ( $a1 = x1 + x2 + x3 + x4$ )	1.) CLUSTER EXCHANGE $(V1 = [x1, \dots, x4, 0, 0, \dots, 0])$
2.) INTRA-TETRODE EXCHANGE $(a1 + b1, a1 + c1, a1 + d1)$	2.) ADJACENT CLUSTER EXCHANGE $(V1 = [x1, \dots, x4, 0, \dots, 0, x17, x18, x19, x20, 0, \dots, 0])$ $V2 = [x1, \dots, x8, 0, \dots, 0])$
3.) CLUSTER SUM ( $S1 = a1 + b1 + c1 + d1$ )	3.) CLUSTER EXCHANGE
4.) INTER-TETRODE EXCHANGE BETWEEN PROCESSORS 1 AND 17, 6 AND 32, 11 AND 27, 16 AND 22	4.) INTRA-TETRODE EXCHANGE $(V1 = UNCHANGED)$ $V2 = [x1, \dots, x20, 0, \dots, 0, x29, x30, x31, x32])$
5.) CLUSTER BROADCAST	5.) CLUSTER EXCHANGE

0489-035/01

FIGURE 2  
XPFEM COMMUNICATION SCHEME



**FIGURE 3**  
**Matrix-Vector Multiply for 2784 Terms**

### III. DISTRIBUTED FINITE ELEMENT METHOD

Finite element analysis consists of four distinct tasks:

1. Pre-processing (problem definition, geometric modeling, and mesh generation),
2. Generation of finite element equations,
3. Solution of the system of finite element equations,
4. Post-processing (calculation of stresses, stress contours, strain energy density, et cetera).

Each of these tasks has varying degrees of inherent parallelism available for exploitation on a distributed network.

The core of the finite element method consists of the computation, assembly and solution of the system of finite element equations. The computation of the finite element equations can be done either an element at a time by computing and assembling element stiffness matrices, or a node at a time by computing and assembling stiffness terms corresponding to individual nodes. The nodal approach enables the stiffness terms to be computed and assembled entirely in parallel *without communication*, and results in near linear speed-up (Speed-up is defined as the computation time for one processor divided by the computation time for N processors.) for any problem size if the workload is distributed evenly. For solution of the resulting system of finite element equations, direct methods based on Gaussian elimination are more general but iterative methods such as the conjugate gradient and Lanczos methods are more adaptable to distributed networks and are applicable to a majority of finite element problems due to the symmetric, positive definite characteristics of finite element matrices.

A relevant issue to the solution of the resulting system of equations is the distinction between in-core and out-of-core techniques. For moderate size (20,000 degree of freedom) problems, the available network core memory of 32 Mbytes is enough to store the entire model. However, as problems increase in size, only a fraction of the model can reside in-core; therefore, techniques for employing out-of-core memory during the solution process become necessary. In the remainder of this section, the development and implementation of an in-core sparse solution technique

and the descriptions of an out-of-core substructuring technique on the tetrahedral network are described.

## THE SPARSE SOLVER

This section describes parallel algorithms which were designed to solve moderate size finite element problems in-core efficiently and with minimal storage. The algorithms are necessarily more complex than conventional, sequential finite element (FE) algorithms, but the increase in performance more than justifies the additional complexity: a single transputer with 8 Mbytes of memory can solve problems with over 10,000 degrees of freedom, and a network of 32 transputers with 1 Mbyte each can solve problems with over 38,000 degrees of freedom. The collection of algorithms which analyze, assemble and solve the FE problem in-core is referred to as the *sparse solver*. The following sections describe the profiler, the assembler and the solver, which comprise the main parts of the sparse solver. These sections will refer to Figures 4-6 which illustrate a sample model and show the form of the model's *global stiffness matrix* (GSM).

### Node Rows versus Degree-of-Freedom Rows

All mention of *rows* and *columns* of the GSM, both in this section and in the occam code, refer to nodes, not to the actual degrees of freedom. For example, for a 3D FE model with 12 nodes, there will be 36 degrees of freedom organized in consecutive sets of x, y and z displacement terms for each node. The notation used here, however, will consider the GSM as having 12 rows, not 36. Each "row" actually consists of x, y and z displacement vectors, so one row contains 3 different arrays. For a three-dimensional problem, the intersection of a row and a column consists of a submatrix with nine terms (x, y and z from the row times x, y and z from the column) referred to as a "patch." The size of a patch differs with the dimensionality of the FE model: two-dimensional patches have four terms (the square of the problem dimension) and three dimensional patches have nine terms. This view of the GSM simplifies many factors related to maintaining information on sparsity. An incidental benefit is that short integers can be used as indices for problems with up to 65,534 degrees of freedom in two-dimensional problems, and up to 98,301 degrees of freedom in three-dimensional problems. If degrees of freedom were indexed directly (rather than indirectly through nodes), short integers could handle problems with only

32,767 degrees of freedom. Due to the heavy reliance on indexing tables, short integers liberate significant amounts of memory that would otherwise be occupied by the more conventional 32 bit integers.

### **The Profiler**

In order to write a sparse solver, general characteristics of the GSM such as the extent of sparsity, the patterns of non-zeros, variations in bandwidth, and the effect of node and element numbering on sparsity must be identified. To assist in determining characteristics of FE models, a program was written to search through model connectivity information to determine where non-zeros will appear. In addition, since renumbering codes create models numbered in an orderly fashion, this program, called a profiler, was designed to recognize patterns that appear due to node numbering. The profiler produces information about non-zeros in each row of the GSM. (A non-zero refers to a non-zero patch, not a single term. A non-zero will occur wherever one node shares the same element with another node. Figure 4 shows that nodes 6 and 9 are both in element 6, so there will be a non-zero patch in row 6, column 9 and row 9, column 6. Row 6 will contain zeros in columns 1, 2, 3, 4, 8 and 10 since node 6 does not share a common element with nodes 1, 2, 3, 4, 8 or 10.) The profiler keeps track of the position of each non-zero and, if consecutive patches are non-zero as well, it will group them together in a "segment" (Figure 4). There are two segments in row 6: the first one starts in column 5 and is 3 patches long and the second one is a singleton and is in column 9. The profiler revealed some interesting properties of the NASA turbine blade model (1575 nodes, 1025 eight noded brick elements). Although numbered well in most regions, the model has a bandwidth spanning 90% of the matrix in certain regions. The turbine blade model node numbering also results in many consecutive patches, indicating that storage of segment information (i.e., the starting column number and the length) is sufficient to index the entire GSM.

Profiling is performed in parallel. The host processor serving the network evenly divides the nodes among the remote processors. Remote nodes compute profiling information on their nodes and return the data to the host. From the profiling data, the host processor determines how to subdivide the finite element problem so that equation solving, the most time consuming step, is optimally load-balanced. The

problem is then redistributed to the network for assembly and solution of the finite element equations.

### **Parallel Assembly of the Global Stiffness Matrix**

Computation of the element matrices and their subsequent summation into the GSM are computationally intensive scalar operations. Most of these operations, however, are independent processes and are thus readily adaptable to parallel processing. Element stiffness matrix computations, for example, are independent tasks and can be performed in parallel with linear speed-up. Summation of the element matrices into a GSM, however, is not inherently parallel because of the data dependencies of GSM terms; a single term in the GSM typically receives contributions from several elements. Rather than distributing elements to processors, a new approach was taken: routines were devised to compute patches of the GSM directly, rather than computing and assembling element matrices. In this manner, processors can work independently without performing redundant calculations and without communicating. The only penalty is in the slight redundancy of storage of element data since some elements' parameters and node coordinates may be stored on more than one processor.

#### **Computation of the GSM by Patches -- Example**

The computation of stiffness terms and their assembly into the global matrix will be illustrated by an example. Consider the FE model and its related GSM in Figure 4. Assume that there are four processors, and that processor 0 (P0) has rows 1 - 3, processor 1 (P1) has rows 4 & 5, processor 2 (P2) has rows 6 - 8 and processor 3 (P3) has rows 9 & 10. Each processor simultaneously begins to compute the first patch on that processor, i.e., P0 computes the patch for node 1 on itself; P1 computes the patch for node 4 against node 2; P2 computes the patch for node 5 against node 6; and P3 computes the patch for node 9 against node 4.

A patch of the GSM is computed in the following manner: First, data for all elements which use the two nodes involved (one node representing the row, and the other representing the column) is needed. P0 will need element 1 for data on node 1, P1 will need element 2 for data on nodes 2 and 4, P2 will need element 6 for data on nodes 5 and 6, and P3 will need elements 4 and 5 for data on nodes 4 and 9. Next, for

each element involved, the local node corresponding to the global node must be determined. With the local node numbers, special element stiffness routines are invoked which return only those stiffness terms relating the two local nodes. For example, P0 must determine which local node the global node 1 represents in element 1. If global node 1 is element 1's local node 3, for example, P0 would then call the special element stiffness routine which will compute only the stiffness terms related to local node 3 acting on itself. P3 would have to determine the local node numbers for global nodes 4 and 9 in element 4 (and element 5). If global node 9 is element 4's local node 1, and global node 4 is element 4's local node 2, processor P3 will call the element stiffness patch routine which computes only the stiffness terms relating local nodes 1 and 2.

In this example, P3 is the only processor with more than one element affecting the processor's first patch. After finding the stiffness terms from element 4, P3 repeats the procedure for element 5 (determine which local nodes the global nodes 4 and 9 correspond to, then compute the stiffness terms only for those two local nodes acting on each other) and sums the stiffness terms from elements 4 and 5 together into one patch.

Processors compute all the patches on one row, then proceed to their next row. Since processors are load balanced by the number of patches, processors will take approximately the same amount of time to compute and assemble their portions of the GSM.

At first glance, the nodal assembly approach appears to have substantially higher indexing overhead than element assembly. Benchmarks, however, have shown that overhead differences between the two methods are negligible in terms of assembly times. The high efficiency of this method for the scalar operation of stiffness matrix assembly is evidenced in the fact that *XPFEM generates and assembles global stiffness matrices nearly two times faster than a Cray X-MP24*. The benchmark turbine blade model took 13.9 seconds to assemble on the Cray, and only 7.4 seconds on XPFEM.

## **Boundary Conditions**

The sparse solver supports applied loads and prescribed displacements. Boundary condition information is stored in two ways: first, the rows and columns of constrained degrees of freedom are zeroed, and the diagonal term is assigned unity. Second, a table of boolean variables is maintained to tell whether or not a degree of freedom is constrained. Using this information, the matrix-vector multiply routine can skip entire rows which would otherwise be wasted on multiplies involving only zero terms.

## **Parallel Jacobi-Conditioned Conjugate Gradient Solver**

The Jacobi-Conditioned Conjugate Gradient (JCG) method uses the inverse of the GSM's diagonal as a preconditioning matrix for the Conjugate Gradient iterative technique. Jacobi preconditioning is desirable in parallel systems for several reasons: the preconditioning matrix is trivial to obtain (other preconditioning methods such as Incomplete Cholesky Decomposition use derivatives of Gaussian elimination-based direct solvers which are difficult to load balance efficiently in parallel), takes little extra storage (the diagonal matrix need only be stored as a vector), and provides good convergence for a minimal computational expense.

There are several ways of decomposing the JCG method on multiple, local memory processors. To find the task distribution appropriate for the application, the implementer must make decisions which strike a balance between memory usage, computation, communication, problem size, and the number of processors anticipated. For small systems of equations to be solved on few processors, for example, each processor can store complete copies of each of the six working vectors thereby saving on communication, but performing redundant computations. The trade-off between communication and computation changes with problem size; time taken by redundant computations grows more quickly than time taken by communication.

The design goal for the sparse solver was to solve very large problems on many processors. For this reason, the intermediate variables in the JCG method were distributed so that only one working vector is stored in its entirety on each processor, no redundant computations are performed, and three global sums are exchanged at

each iteration (two scalar sums and one vector sum). Appendix C outlines the algorithm and variables.

### **Effects of Bandwidth**

Conventional banded matrix solvers such as those used by NASTRAN attempt to simultaneously minimize both storage and computational requirements by storing, and operating on, only the stiffness terms that fall within the bandwidth of the FE matrix. The scheme works well for models like beams and cantilevers which can be made with tight bands, but often results in wasteful storage and computation of zero terms when complex models are built.

#### **Bandwidth Has Minimal Affect on Storage Requirements**

Unlike conventional banded solvers, the storage requirements of the sparse solver in XPFEM is only marginally affected by bandwidth. Only non-zero terms of the GSM are stored, regardless of the bandwidth, so the memory required by stiffness terms remains constant. The pointer tables, however, may require proportionately more memory (typically 25% of the GSM memory requirements) when the bandwidth is excessively large since the rows in a poorly banded matrix are usually highly segmented.

#### **Bandwidth Does Not Affect Computational Requirements**

The computational effort in the sparse solver, like the memory required by the GSM, is also independent of bandwidth since operations are carried out only on non-zero terms. The number of floating point operations required for a poorly banded matrix identically equals the number required for a tightly banded matrix.

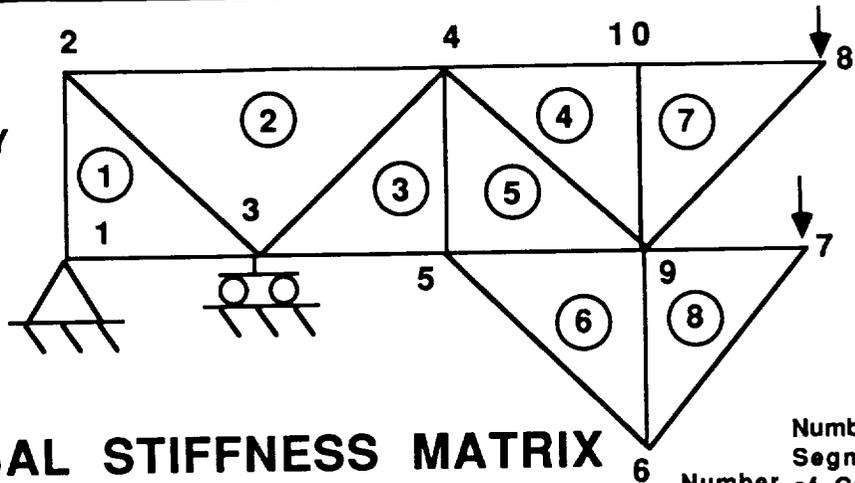
#### **Bandwidth Determines Amount of Communication**

Although the amount of computation does not increase, bandwidth does govern overall solution time. At one point in the JCG iteration loop, processors exchange portions of the projection vector,  $\{p\}$ , before they perform the matrix-vector multiply (Figure 5). If the bandwidth is large, the distance, i.e., the number of processor links, each portion of  $\{p\}$  must travel is also large and the amount of communication time

increases (Figure 6). The dependence of solution time on bandwidth is critical. Tightly banded problems usually run at speed-ups of 28 (XPFEM has 32 processors), but problems whose band extends across more than half of the matrix rarely achieve speed-ups higher than 18.

The original NASA turbine blade model had an enormous bandwidth (it extended across 90% of the matrix in places) and resulted in unsatisfactory parallel performance--the resulting solution time of 96 seconds corresponds to a speed-up of only 14--well below 50% parallel efficiency. After a crude renumbering algorithm cut the bandwidth to 32% of the matrix, equation solution time dropped to 73 seconds corresponding to a speed-up of 23. NASTRAN's highly effective renumbering code reduced the bandwidth on this problem to just 13% of the matrix. Since XPFEM does not have a similar algorithm, NASTRAN's renumbering of the turbine blade model was used by XPFEM to further demonstrate bandwidth effects. With NASTRAN's renumbering the turbine blade problem took 64 seconds to solve giving a speed up of 27. Renumbering had negligible effects on matrix assembly times.

THE FINITE ELEMENT  
MODEL WITH *BOUNDARY*  
CONDITIONS AND  
EXTERNAL LOADS



THE GLOBAL STIFFNESS MATRIX

	1	2	3	4	5	6	7	8	9	10	Number of non-Zeros	Number of Consecutive non-Zeros
1	Shaded	Shaded	Shaded								3	1
P0 2	Shaded	Shaded	Shaded	Shaded							4	1
3	Shaded	Shaded	Shaded	Shaded	Shaded						5	1
P1 4		Shaded	Shaded	Shaded	Shaded				Shaded	Shaded	6	2
5			Shaded	Shaded	Shaded	Shaded			Shaded		5	2
6					Shaded	Shaded	Shaded		Shaded		4	2
P2 7						Shaded	Shaded		Shaded		3	2
8								Shaded	Shaded	Shaded	3	1
P3 9				Shaded	7	2						
10				Shaded				Shaded	Shaded	Shaded	4	2
											44	

EACH SHADED BOX, OR PATCH, REPRESENTS 4 TERMS (THE SQUARE OF THE DIMENSIONALITY OF THE PROBLEM) FOR ALL COMBINATIONS OF DEGREES OF FREEDOM BETWEEN 2 NODES. THE ACTUAL STIFFNESS MATRIX HAS 400 TERMS.

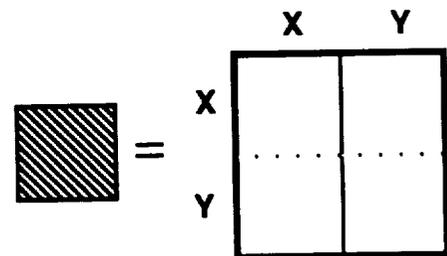
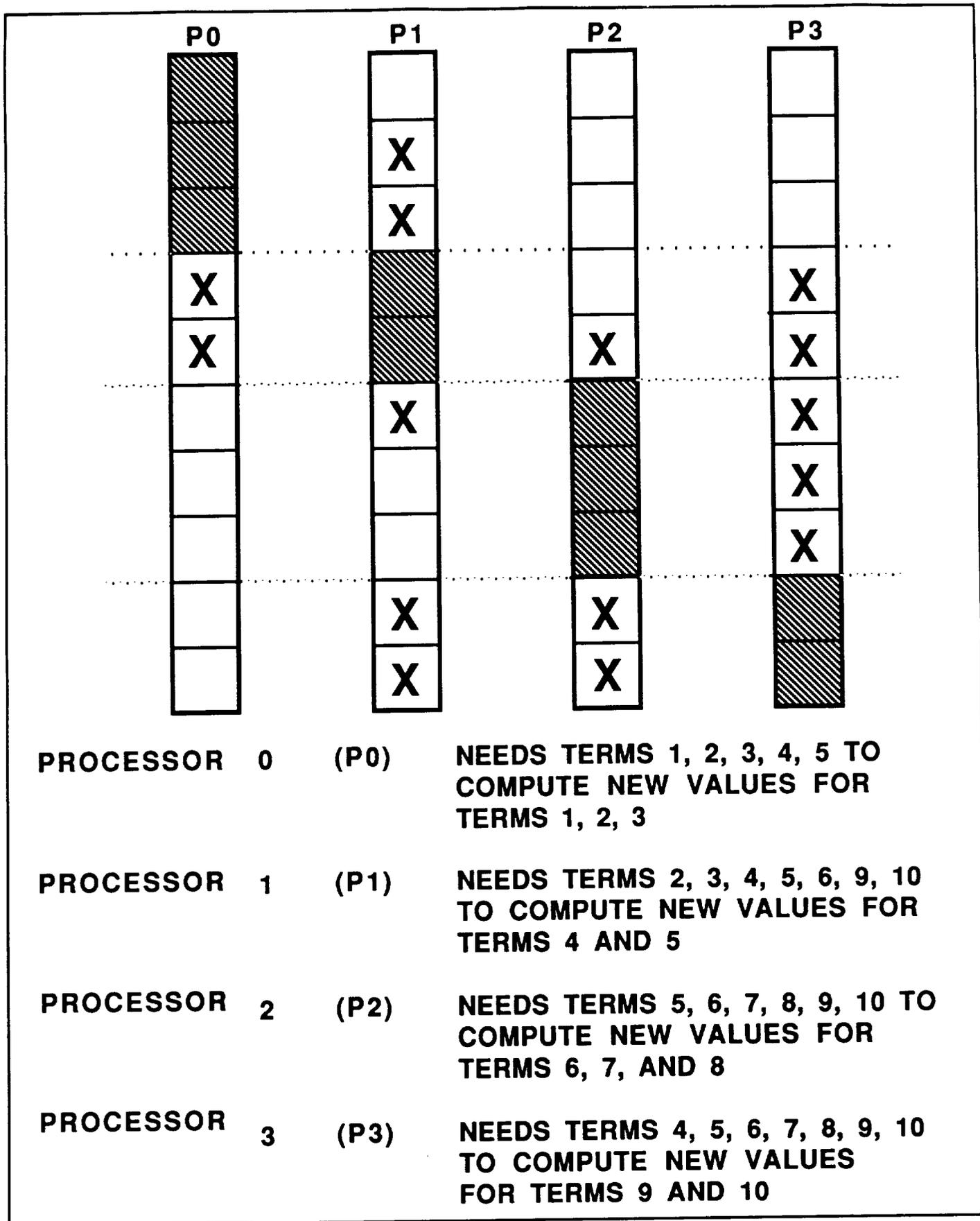


FIGURE 4



0689-007/03

**FIGURE 5**  
**P-VECTOR OVERLAP FOR MATRIX-VECTOR MULTIPLICATION**

P0 NEEDS DATA FROM P1

P1 NEEDS DATA FROM P0, P2, P3

P2 NEEDS DATA FROM P1, P3

P3 NEEDS DATA FROM P1, P2

	P0	P1	P2	P3
P0	○	X		
P1	X	○	X	X
P2		X	○	X
P3		X	X	○

2 PIPELINE COMMUNICATION STEPS NECESSARY TO EXCHANGE SEGMENTS OF P-VECTOR. ALL SENDS AND RECEIVES IN A GIVEN STEP ARE PERFORMED SIMULTANEOUSLY

	P0	P1	P2	P3
STEP 1:	SEND TERMS 1, 2, 3 TO P1  RECEIVE 4, 5 FROM P1	SEND TERMS 4, 5, TO P0 AND P1  RECEIVE 1, 2, 3 FROM P0  RECEIVE 6, 7, 8 FROM P2	SEND 6, 7, 8 TO P1 AND P3  RECEIVE 4, 5 FROM P1  RECEIVE 9, 10 FROM P3	SEND 9, 10 TO P2  RECEIVE 6, 7, 8 FROM P2
STEP 2:	IDLE	RECEIVE 9, 10 FROM P2	SEND 9, 10 TO P1 SEND 4, 5 TO P3	RECEIVE 4, 5 FROM P2

FIGURE 6

0689-007/02

P-VECTOR PIPELINE COMMUNICATION EXCHANGE

## THE OUT-OF-CORE SOLVER

Out-of-core finite element solution methods for networks of microprocessors pose significant design and implementation problems. The most severe restriction is the relatively slow access time of external storage devices which are on the same cost scale as the microprocessors themselves. Extremely fast external storage devices, such as the solid-state secondary storage used in Cray computers, are available but are prohibitively expensive for workstation-level computer systems.

To obtain reasonable performance from an out-of-core solver using common external storage devices such as SCSI hard disk drives, the use of an efficient parallel solution method that minimizes disk access is required. These requirements directly conflict with each other: Iterative solvers are efficient in parallel but require frequent access to the externally-stored stiffness matrix; iterative block solvers need a moderate amount of disk access but are less efficient on multiple processors due to load balancing difficulties; finally, direct solution methods need less disk access than iterative methods but are also less efficient in parallel. To resolve these conflicts, a substructuring technique adapted from [8] is proposed which takes advantage of XPFEM's in-core conjugate gradient solver, can solve hundred thousand degree-of-freedom problems efficiently, and requires only minimal disk access.

The proposed out-of-core parallel solution method is a variant of the substructuring technique used by regular sequential solvers to increase solver capacity. It exhibits three desirable features needed by a network of microprocessors:

1. Disk access time is a small fraction of total solution time, enabling relatively slow, low-cost disk drives to be used.
2. The method can be made extremely efficient in parallel by having processors decompose individual substructures.
3. An iterative solver can be used to solve the condensed finite element problem efficiently on the distributed network.

The substructure concept and its implementation on a network of transputers are described below.

## The Substructure Method

The substructure method is a technique for solving a large finite element problem by replacing it with a set of equivalent problems, each with a much smaller number of degrees of freedom [9,10]. The method consists of partitioning the full finite element model into substructures, then eliminating degrees of freedom internal to each substructure. The substructures can then be viewed as *superelements* which have only boundary nodes. If the substructures contain many internal nodes relative to boundary nodes, the size of the reduced set of equations will be several orders of magnitude smaller than the original equations. The displacements of a superelement's boundary nodes can be determined by solving the reduced system of equations with appropriately modified boundary conditions. Once these displacements are calculated, each substructure can be treated as a separate finite element problem with applied displacement boundary conditions.

To see how this reduction can be accomplished, suppose that a finite element problem has been subdivided into a number of substructures. Letting the superscript  $a$  denote a particular substructure, the finite element equations for the node displacements of the substructure have the familiar form

$$\mathbf{K}^a \mathbf{x}^a = \mathbf{f}^a$$

where  $\mathbf{x}^a$  is the displacement vector,  $\mathbf{f}^a$  is the force vector, and  $\mathbf{K}^a$  is the stiffness matrix of the substructure. Assembling these sets of equations over all the substructures yields the complete set of equations for the displacements in terms of the applied forces :

$$\mathbf{K} \mathbf{x} = \mathbf{f}$$

Now consider the matrix equation for the substructure  $a$ . Let the subscript 1 denote boundary nodes and subscript 2 denote interior nodes of the substructure. Then the equations for the displacements  $\mathbf{x}^a$  can be written in the form :

$$\mathbf{K}^{a_{11}} \mathbf{x}^{a_1} + \mathbf{K}^{a_{12}} \mathbf{x}^{a_2} = \mathbf{f}^{a_1} \quad (1)$$

$$\mathbf{K}^{a_{21}} \mathbf{x}^{a_1} + \mathbf{K}^{a_{22}} \mathbf{x}^{a_2} = \mathbf{f}^{a_2} \quad (2)$$

Solving (2) for the interior displacements,  $x^{a_2}$ , and substituting into (1) yields the following matrix equation for the boundary nodes of the substructure:

$$[ K^{a_{11}} - K^{a_{12}}(K^{a_{22}})^{-1}K^{a_{21}} ] x^{a_1} = f^{a_1} - K^{a_{12}}(K^{a_{22}})^{-1}f^{a_2}$$

Note that  $x^{a_2}$ , the interior displacements, have been removed from the finite element equation. Assembling these equations over all substructures yields a matrix equation

$$K' x' = f'$$

in which the unknowns  $x'$  are the displacements at the boundaries of the substructures and  $K'$  and  $f'$  are the modified global stiffness matrix and applied force vector.

Once the displacements for the boundaries of the substructures are calculated, the displacements for the interior nodes of each substructure can be determined by one of two methods. Either use the explicit formula

$$x^{a_2} = (K^{a_{22}})^{-1}(f^{a_2} - K^{a_{21}}x^{a_1})$$

or solve the substructure finite element equations

$$K^a x^a = f^a$$

using the boundary solution as applied displacements. The second method appears to be more appropriate since  $(K^{a_{22}})^{-1}$  may be too large to store for each substructure. At this stage, the condensed substructures, or superelements, can be treated as *independent* finite element problems and can be distributed to a network of processors to be solved concurrently. If the superelements themselves are too large to be solved in-core, the entire substructuring procedure may be reapplied to groups of superelements to reduce the problem sizes even further.

## Steps to Solution for Substructuring on Transputers

The steps below outline the out-of-core substructure solution method for the network of 32 transputers in XPFEM. In addition to the 1 Mbyte of RAM on each processor, eight 200 Mbyte hard disk drives, each with a 2 Mbyte RAM buffer, are required. Each cluster of four transputers shares one disk drive. Further, dynamic link switching is used to alternate between the pipeline/cluster-to-disk configuration (Figure 7) and the tetrahedral configuration (figure 2) needed for solution. Note that, unlike in the sparse solver, dynamic switching will not hinder solution for out-of-core problems because the durations between reconfigures are orders of magnitude greater (i.e. several seconds) than those in the in-core solver.

To more fully explain the substructuring concept, an example problem will be described in detail. The example consists of a 2D finite element model of a square plate with 288 linear quadrilateral elements in 288 rows. There are 83,521 nodes (167,042 degrees of freedom) and 82,944 elements (Figure 8).

**Step 1: Subdivide the original model into substructures.** This step is done by the host processor. Ideally, each substructure will have a high ratio of interior nodes to boundary nodes. For the example model, a substructure with ideal dimensions would be a square with 18 elements to a side, giving 361 nodes and 324 elements. The full model would then be subdivided into  $82,944/324 = 256$  substructures. A substructure will then have 289 interior nodes and 72 boundary nodes. One substructure is sent to each processor.

After distributing the substructures found in Step 1, the host processor becomes idle, and the remote processors perform the remaining steps in parallel:

**Step 2: Compute a stiffness matrix for the substructure.**

**Step 3: Reorder the stiffness matrix to separate internal and external (boundary) degrees of freedom (Figure 9).**

**Step 4: Compute LU decompositions of the  $K_{22}$  internal stiffness submatrix.** An LU decomposition requires  $(1/6)(n \times n)(\text{half-bandwidth})$  matrix operations, where one matrix operation consists of a floating point multiply and a floating point add. For  $n = 578$  internal degrees of freedom and a half-bandwidth of 60, the number of matrix operations is  $3.3E+6$ . A T800 can sustain 0.5 Mflops on matrix operations, so each processor will take about 7 seconds for decomposition.

**Step 5: Eliminate internal degrees of freedom by assembling a superelement with nodes only on the boundary.** This step consists of finding the inverse to an upper triangular matrix and two matrix-vector multiplies (refer to the equation at the bottom of Figure 9). This step takes  $n^2m + bmn + n^2b$  matrix operations where  $b =$  the half-bandwidth,  $m =$  external degrees of freedom and  $n =$  internal degrees of freedom. In the example problem,  $n = 578$ ,  $m = 144$  and  $b = 60$ ; therefore, the number of matrix operations is approximately  $7.3E+7$  and will take about 146 seconds on one T800.

**Step 6: Send the superelement stiffness matrix to the transputer disk interface to be written to disk.** The superelement stiffness matrices for the example problem have  $(144)^2 = 20,736$  terms, or 165,888 bytes. During this phase, one link from each transputer in a cluster of four is joined to the transputer disk controller. All four transputers simultaneously send their data to the disk controller which stores the data in its 2 Mbyte buffer. The transputers are then delayed only by the amount of time it takes to transmit the decomposed matrix over a link, which is less than 1 second. The disk controller starts writing as soon as it begins to receive data, and will continue to do so after the transputers have resumed computations.

Steps 2-6 are repeated until each substructure has been condensed into a superelement. With 82,944 elements in the original model, 256 elements per superelement and 32 processors, 10 cycles of Steps 2-7 are needed to condense the entire model into superelements. The time for each cycle includes:

- 5 seconds for assembling a substructure stiffness matrix
- 7 seconds to perform a LU decomposition on the internal node submatrix
- 146 seconds to compute the superelement stiffness matrix by eliminating internal degrees of freedom
- 1 second to write the superelement stiffness matrices to disk

Each cycle then takes about 159 seconds, and 10 cycles take 1590 seconds.

**Step 7: If the condensed problem is small enough, solve it in-core with XPFEM's sparse solver. If it is still too large, form substructures out of the superelements, condense them into a next higher level of superelements and repeat Steps 1-7.** After eliminating interior nodes in the example problem, there will be 19 rows and 19 columns of nodes in the model (Figure 10). Rows and columns have 289 nodes each, so there are now 9,537 nodes (nodes at intersections of a row and column are counted only once). This problem *can* be solved in the collective core memory of all 32 processors with the sparse solver described in Section III.1. It is important to note, however, that global stiffness matrices (GSM) generated from superelements will be much more full than the GSM's generated from regular elements. This reduces the size of the problem which can be solved in-core with the sparse solver. Solution time for the relatively dense in-core problem with 9,537 nodes is estimated at 1200 seconds.

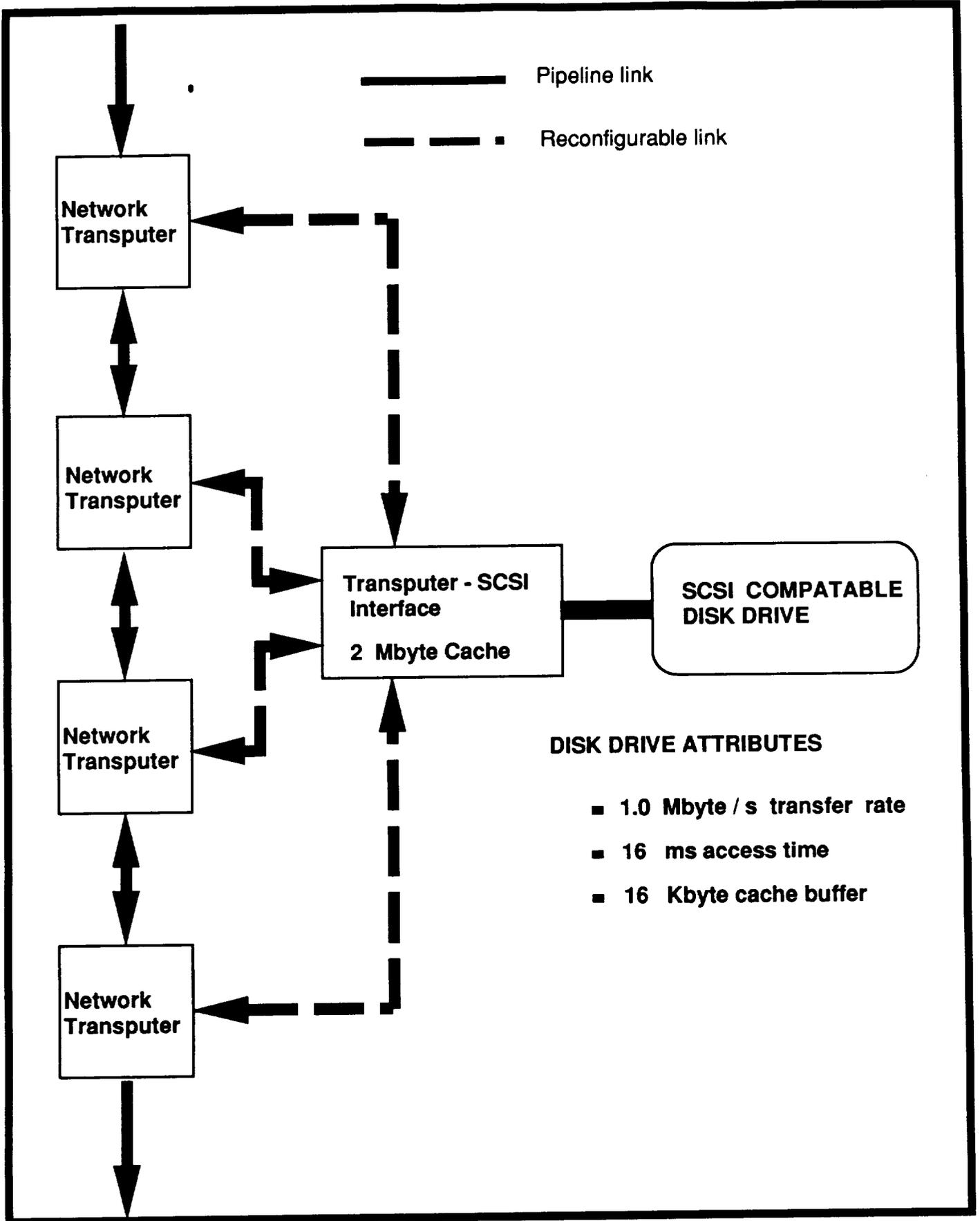
**Step 8: Write the displacement solution to disk.**

**Step 9: Compute displacements of eliminated nodes one substructure at a time.** Each processor recomputes the stiffness matrix and sets up a finite element problem using the known displacements from Step 7 as applied displacements to the boundary of the substructure. The separate substructure finite element problems are small (between 300 and 400 nodes) and are computed quickly on each processor. Solution time for such small problems on one processor is estimated to be less than 40 seconds.

Total solution time for the sample problem of 167,042 degrees of freedom, using a substructure/superelement solution technique on 32 transputers with 8 hard disk drives is then estimated to be under 2900 seconds, or 48.3 minutes.

## **Secondary Storage Hardware Considerations**

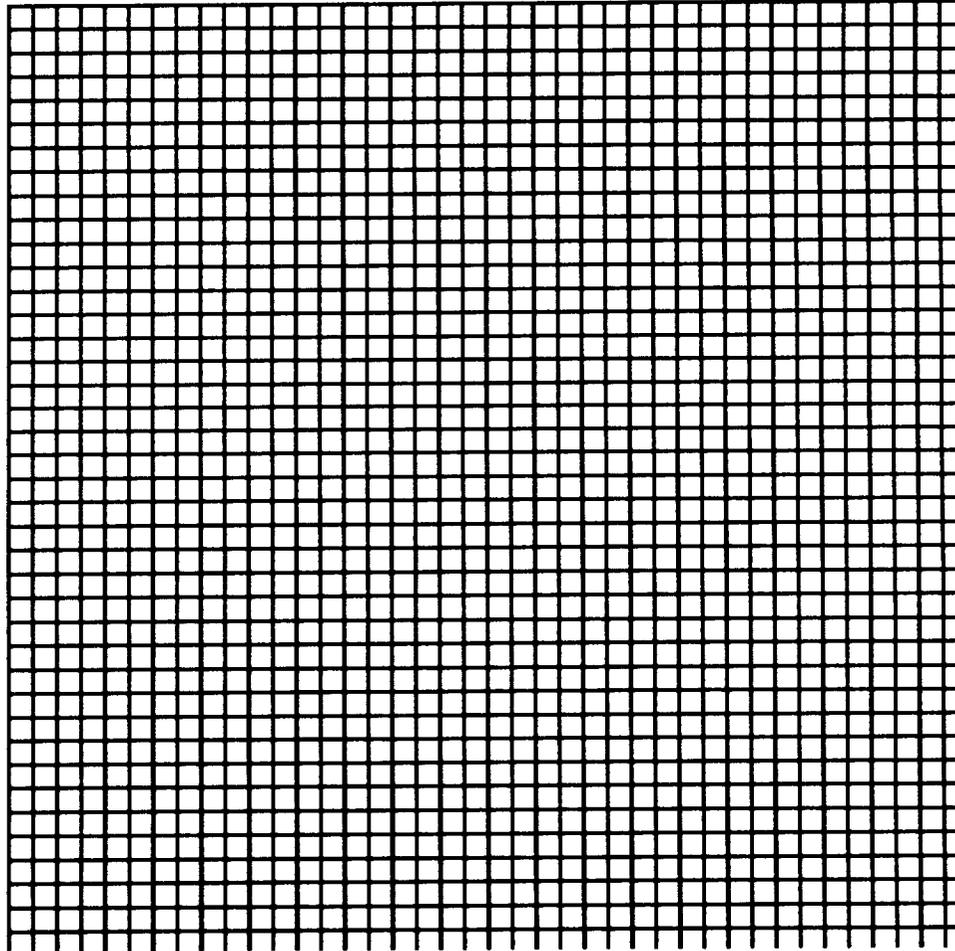
In order to implement the out-of-core substructuring technique as described above, it will be necessary to use a more advanced transputer disk drive interface. The new interface would contain a 32-bit T800 with 2 Mbytes of memory, 4 bidirectional transputer links for communication with the network and a SCSI controller for communication with a SCSI compatible disk drive. With 2 Mbytes of onboard memory the interface could accommodate a fully functioning disk operating system program and have approximately 500 Kbytes of memory per transputer link to be used as I/O buffers. The disk operating system would be capable of executing disk access requests received from any of the four transputer links. With this configuration, as many as four transputers can simultaneously access buffered I/O from a disk drive at rates approaching 2 Mbytes/second. For the purposes of the substructure solver, one interface and one 200 Mbyte Winchester disk drive would be required for every four transputers in the solver network. The basic configuration is illustrated in Figure 7.



**FIGURE 7**  
**NETWORK DISK DRIVE CONFIGURATION for OUT-of-CORE SOLVER**

288 elements across

288  
elements  
vertically



**Figure 8**

**A large model with 83,521 nodes and 82,944 elements. This problem cannot be solved in-core.**

The FE matrix equation  $[K]\{x\} = \{F\}$  may be partitioned as follows:

$$\begin{bmatrix} \mathbf{K}_{11} & \mathbf{K}_{12} \\ \mathbf{K}_{21} & \mathbf{K}_{22} \end{bmatrix} \begin{Bmatrix} \vec{\mathbf{x}}_1 \\ \vec{\mathbf{x}}_2 \end{Bmatrix} = \begin{Bmatrix} \vec{\mathbf{F}}_1 \\ \vec{\mathbf{F}}_2 \end{Bmatrix}$$

$( )_1 = \text{Boundary term}$

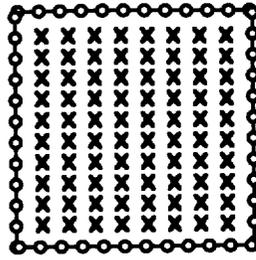
$( )_2 = \text{Interior term}$

Interior variables,  $\vec{\mathbf{x}}_1$ , can be eliminated by rewriting the partitioned equations as:

$$[\mathbf{K}_{11} - \mathbf{K}_{12} (\mathbf{K}_{22}^{-1}) \mathbf{K}_{21}] \vec{\mathbf{x}}_1 = \vec{\mathbf{F}}_1 - \mathbf{K}_{12} (\mathbf{K}_{22}^{-1}) \vec{\mathbf{F}}_2$$

**Figure 9**  
Matrix Partitioning

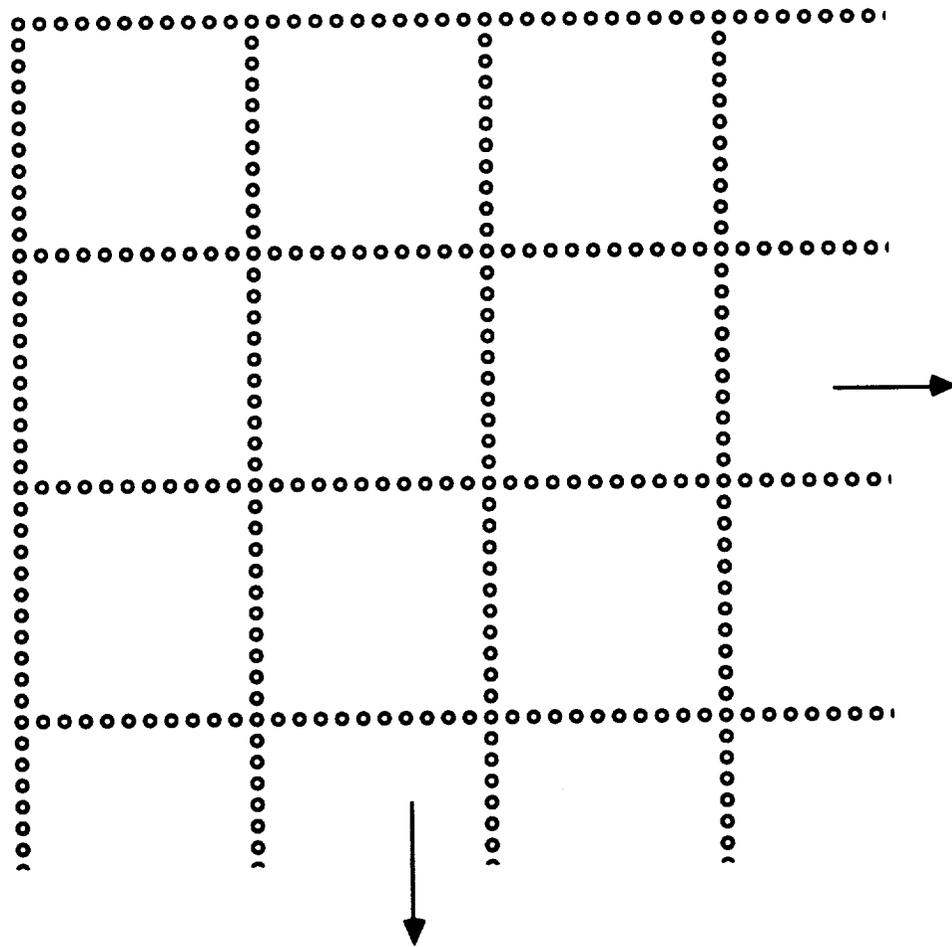
Each substructure has  
 $19 \times 19 = 361$  nodes and  
 $18 \times 18 = 324$  elements.  
 There are 72 nodes on  
 the boundary; the 289  
 interior nodes are  
 eliminated.



○ Boundary node (72)  
 X Interior node (289)

18 columns of superelements  
 horizontally (3 shown)

18 rows of  
 superelements  
 vertically  
 (3 shown)



**FIGURE 10**

The full model condensed into  
**9,537 Nodes, and 256 superelements.**  
 This problem **CAN** be solved in-core.

#### IV. PRE- AND POST-PROCESSING

Pre- and post-processing modules were developed for XPFEM to achieve an end-to-end transputer based finite element workstation. The pre-processing tasks involved implementing FINITE QUADTREE, an automated 2-D finite element mesh generator, in occam on the transputer development system, linking QUADTREE with the solution procedures, and developing a NASTRAN interface for the network. Post-processing tasks included procedures for displaying undeformed and deformed models, calculating stresses, and displaying principal stresses and stress contours. Additional post-processing procedures included a library of adaptive analysis routines for identifying mesh areas requiring refinement. These adaptive analysis routines consist of error calculations and mesh refinement operators.

##### FINITE QUADTREE

FINITE QUADTREE is a complete 2-D finite element pre-processing program which employs quadrants to develop a finite element mesh. Included in FINITE QUADTREE are modules for geometric modeling, creating a materials library, defining a physical problem using constraints and forces, meshing the geometric model, refining the mesh, and writing model files to disk.

FINITE QUADTREE meshes geometric models by enclosing the model in a rectangle and successively subdividing the rectangle into quadrants until the designated level of refinement is obtained. (Figure 11) This quadrant method results in extensive tree structures within QUADTREE. The lowest level of the tree is associated with the elements of the finite element mesh.

Geometric models created in QUADTREE are based on the geometric entities *vertex*, *edge*, and *face* (Figure 12). Using edge types such as lines, quadratics, and arcs, the geometry of an object can be specified precisely and manipulated easily in contrast to element based modelers. Figures 13, 14 and 15 demonstrate the QUADTREE meshing process before elements are constructed. In Figure 13 the boundary of the geometric model is discretized. Following this discretization, boundary quadrants and then interior quadrants are identified and inserted. Figure 14 shows the model sectioned into quadrants. These quadrants are repositioned, as in Figure 15, to simplify the creation of robust finite elements. The finite elements are

created with the quadrants using the quadrant vertices, midpoints, and quarter-points as nodes.

FINITE QUADTREE was developed in FORTRAN by researchers at Rensselaer Polytechnic Institute. The maturity of the occam compiler for transputers, the goal of implementing QUADTREE on transputers, and the concept of eventually developing a distributed mesh generator for a transputer network necessitated a translation of FINITE QUADTREE to occam. Using a FORTRAN-to-occam translator and restructuring the program by eliminating GO TO's and substituting the occam construct RETYPES for common blocks permitted FINITE QUADTREE to be implemented in its entirety on a 4 Mbyte T800 transputer development board. Graphics output from QUADTREE is accomplished either through the network graphics system (B408/B409 modules) or an EGA/VGA graphics card.

A conceptual design for distributed mesh generation was completed in this research effort. The design, which requires adaptation of FINITE QUADTREE using a coloring scheme, is described in Appendix H.

### **NASTRAN Interface**

To facilitate use of XPFEM, a NASTRAN interface was completed and integrated into the system. The interface receives a fixed-format COSMIC or MSC NASTRAN bulk data deck, analyzes the included cards, translates the data to XPFEM format, and generates the standard XPFEM input files for connectivity and constraints, nodes, and forces (Figure 16).

The current version of the NASTRAN interface is designed only for displacement solutions. Data deck cards recognized by the interface are listed in Appendix G. Unrecognized cards are echoed to screen but not written to file.

### **XPPOST**

Quantities of primary interest in finite element analysis (e.g., stresses) are often derived quantities, i.e., quantities which must be calculated from computational data. In structural analysis, the stresses are derived from the displacements, which are obtained from the solution of the finite element equations. Because of the derived

nature of relevant quantities and the discretized nature of finite element analysis, post-processing of results is required. This post-processing not only includes calculating derived quantities but also smoothing and effectively displaying data. XPPOST uses distributed processing and the enhanced graphics capabilities of the transputer network to provide several options for viewing models and stresses (Figures 17 and 18).

XPPOST requires connectivity, nodal coordinate, force, constraint and displacement data. Models are displayed in normalized coordinates at one-half to three-fourths screen size. Undeformed and deformed 2-D and 3-D models are displayed as wire-frame models. A homogeneous transformation matrix permits scaling, translation, and rotation of the displayed model. Orthogonal or perspective viewing is also implemented using the transformation matrix.

Stress options for 2-D models in XPPOST include principal stresses, maximum shear stress and stress contours for normal and shear stresses. The principal stress and stress contour routines are based on analytical equations for stresses at a point. For the principal stress option, normal and shear stresses at the centroid of each element are calculated and transformed to obtain the principal stresses, maximum shear stress, and angle of orientation. Using stress bars to indicate relative magnitude, these stress values and orientation can be displayed with or without the model. Options enable principal tensile, compressive, combined tensile and compressive, or shear stresses to be displayed.

Stress contours are obtained by smoothing calculated stress values at the integration points to the nodes for each element. The smoothing matrices are derived from a least squares analysis. Prior to nodal averaging the smoothed nodal values are used to calculate centroidal stress values. Contours are displayed by creating contour triangles consisting of two consecutive nodes of the element and the element centroid. A color look-up table provides the necessary fill information.

### **Adaptive Analysis**

The adaptive analysis capability implemented in XPFEM employs strain energy error analysis to successively refine a finite element mesh and reanalyze the resulting model until element errors meet a specified tolerance (Figure 19). The XPFEM

adaptive analysis capability is a practical design tool due to the combination of acceptable solution turnaround times (a result of the computational speed of XPFEM), automated mesh generation available in FINITE QUADTREE, and advanced error analysis algorithms employing bubble functions.

The error analysis algorithms of Baehmann [11] determine the areas of refinement. In these algorithms, errors within the interior of the finite elements are assumed significant and errors occurring at the finite element boundaries (i.e., "jump" errors) are assumed negligible. These assumptions permit cubic and quartic bubble functions which vanish on element edges (Figure 20) to be used in a higher order computation of element error (Figure 21). Element errors and total global error are computed in the energy norm. From the total global error and the calculated strain energy obtained from the finite element solution, an approximate relative error is computed (Figure 21). The exact relative error is unknown since, in general, the exact strain energy is unavailable.

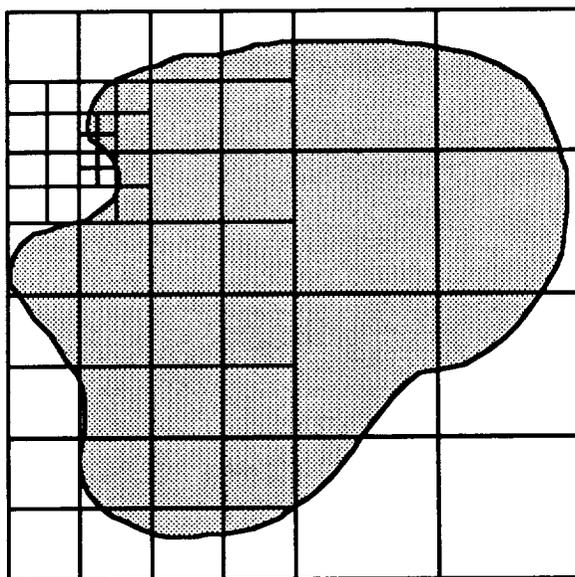
Mesh refinement is necessary if the approximate relative error exceeds a given tolerance. Areas of refinement are determined by comparing each element error to the average element error for the entire model. An element is identified for refinement if its error is larger than the average element error.

A robust adaptive mesh refinement scheme must consider two situations: models with singularities, and large versus small global error. By controlling the level of refinement for different refinement areas, the implemented adaptive analysis scheme accounts for both of these refinement situations to distribute relative error evenly among the finite elements.

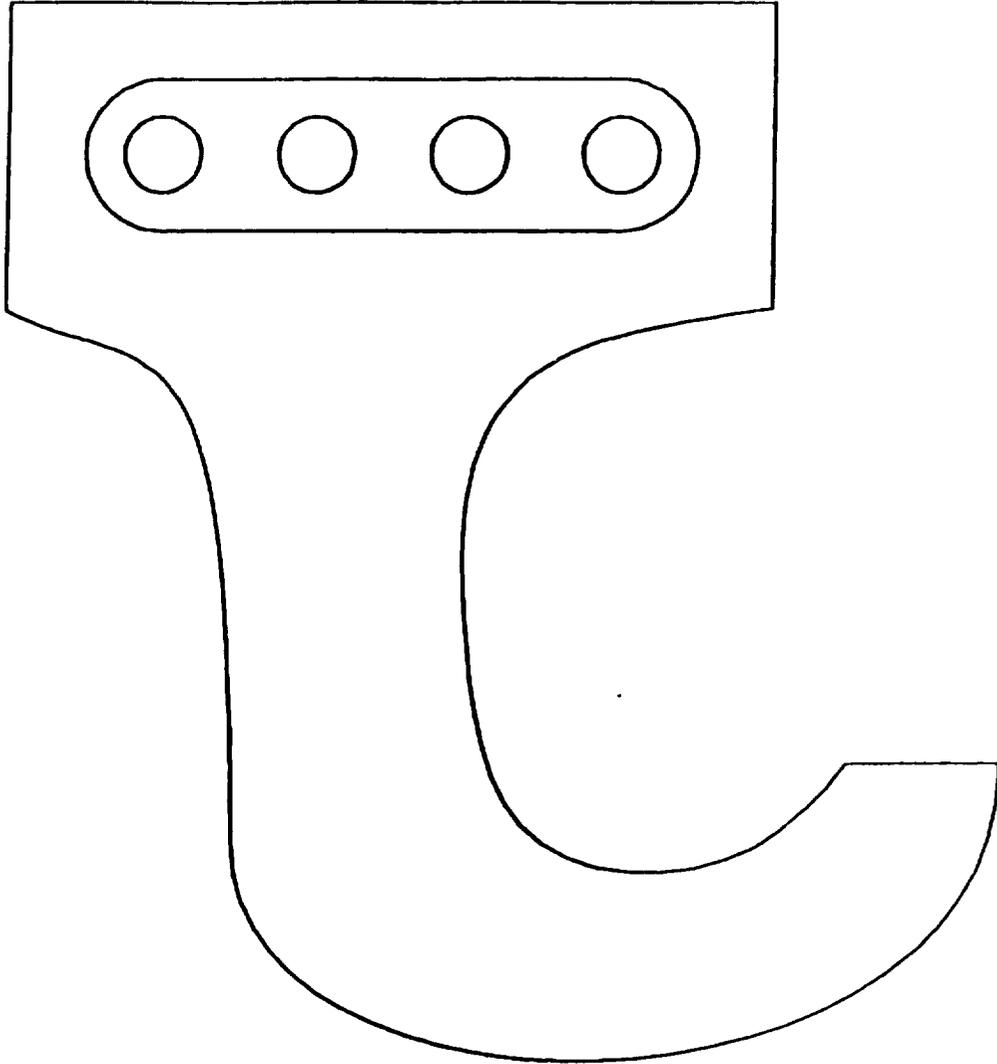


SPARTA, INC.

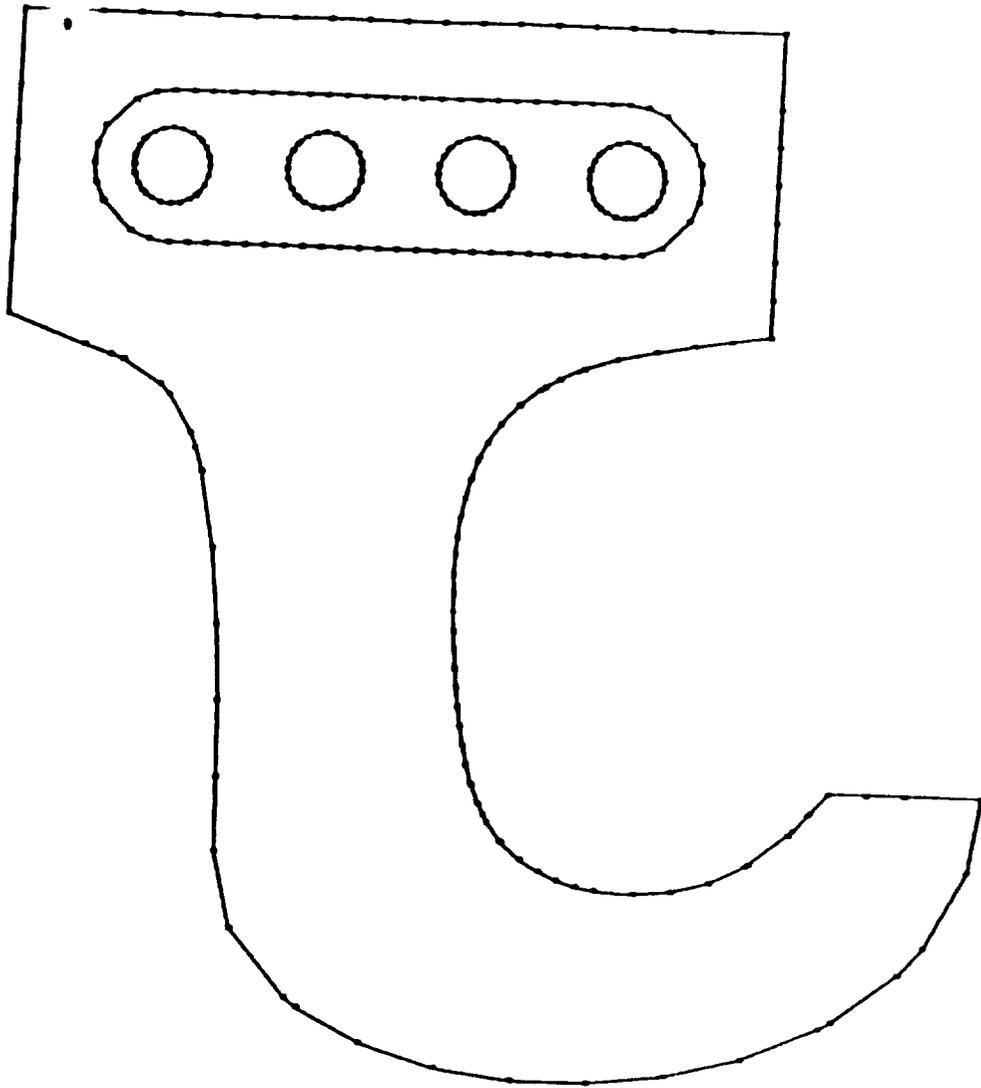
# QUADTREE THEORY



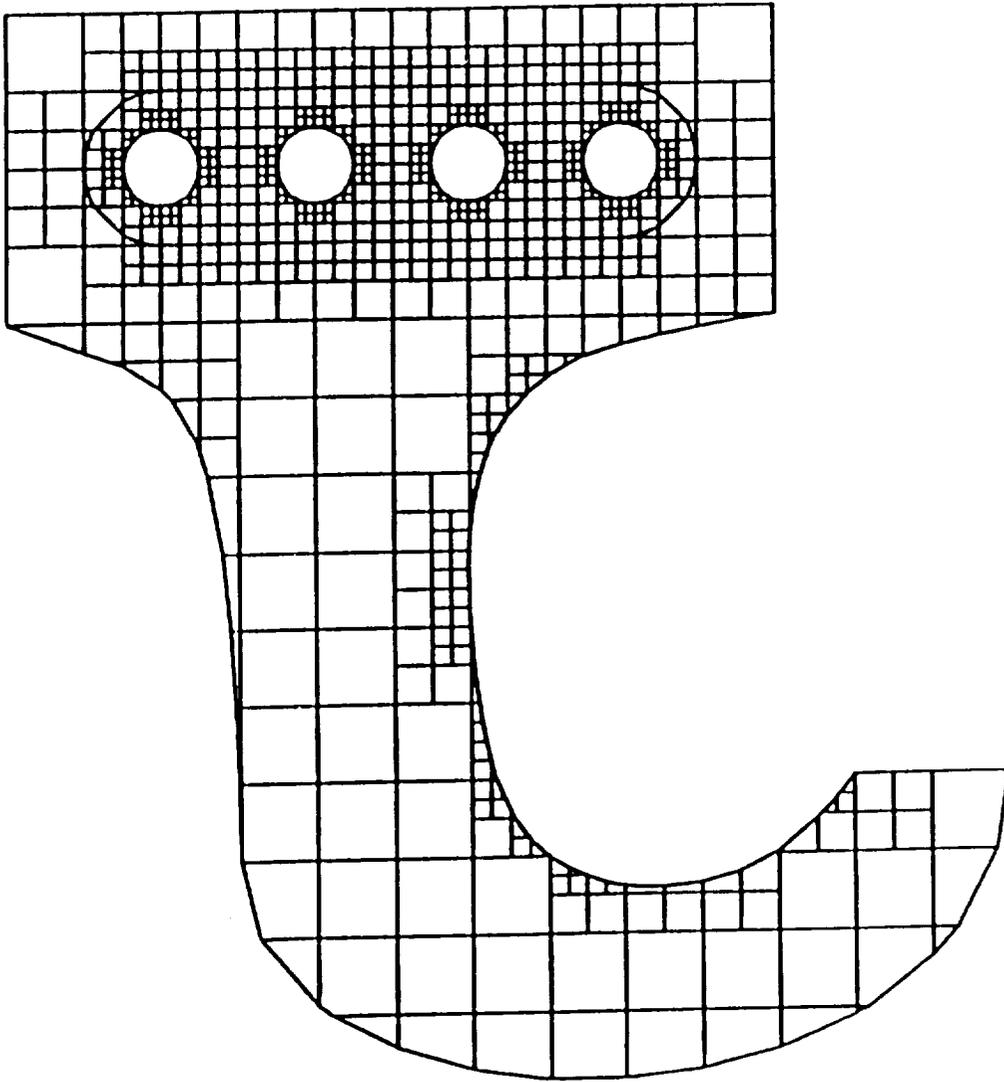
**FIGURE 11**  
**MODEL SECTIONED INTO QUADRANTS**



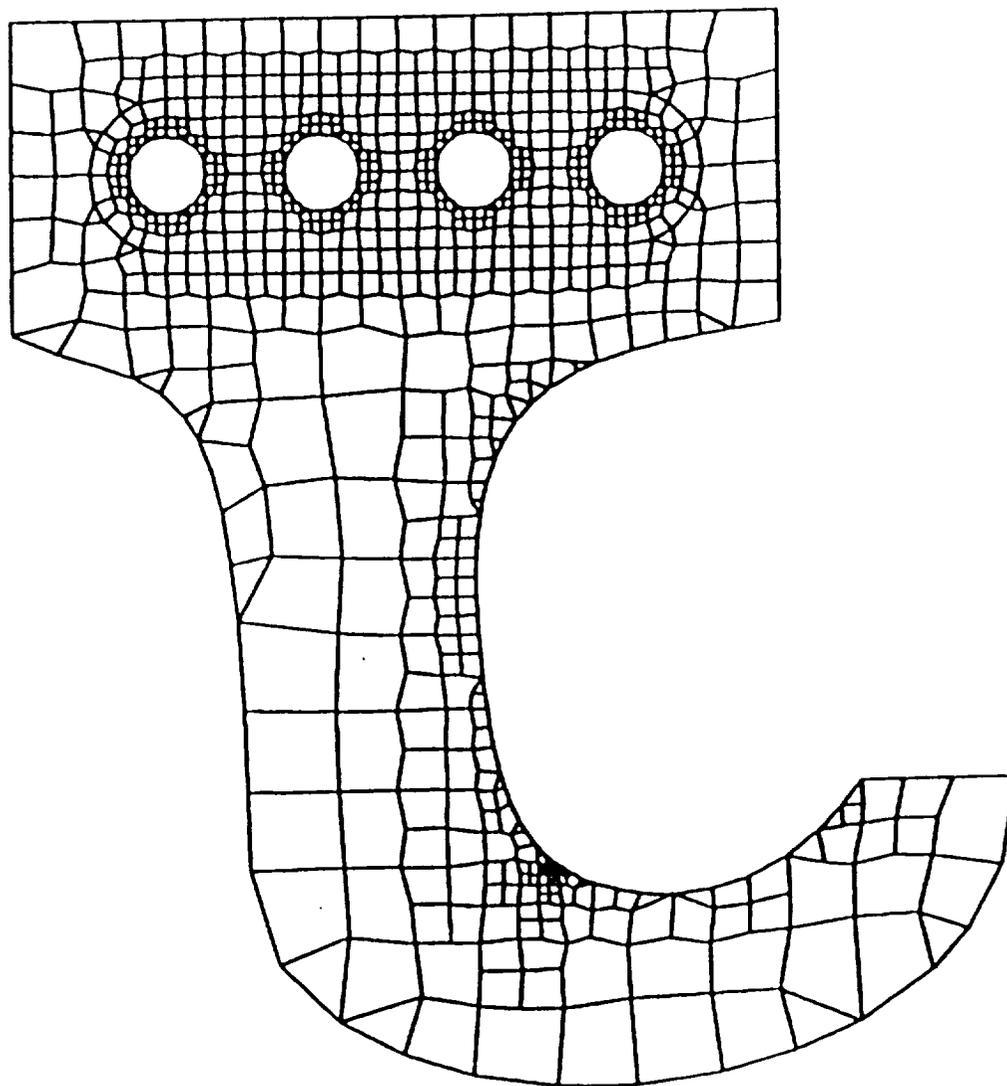
**FIGURE 12**  
**Geometric model created with the**  
**geometry module in FINITE QUADTREE**



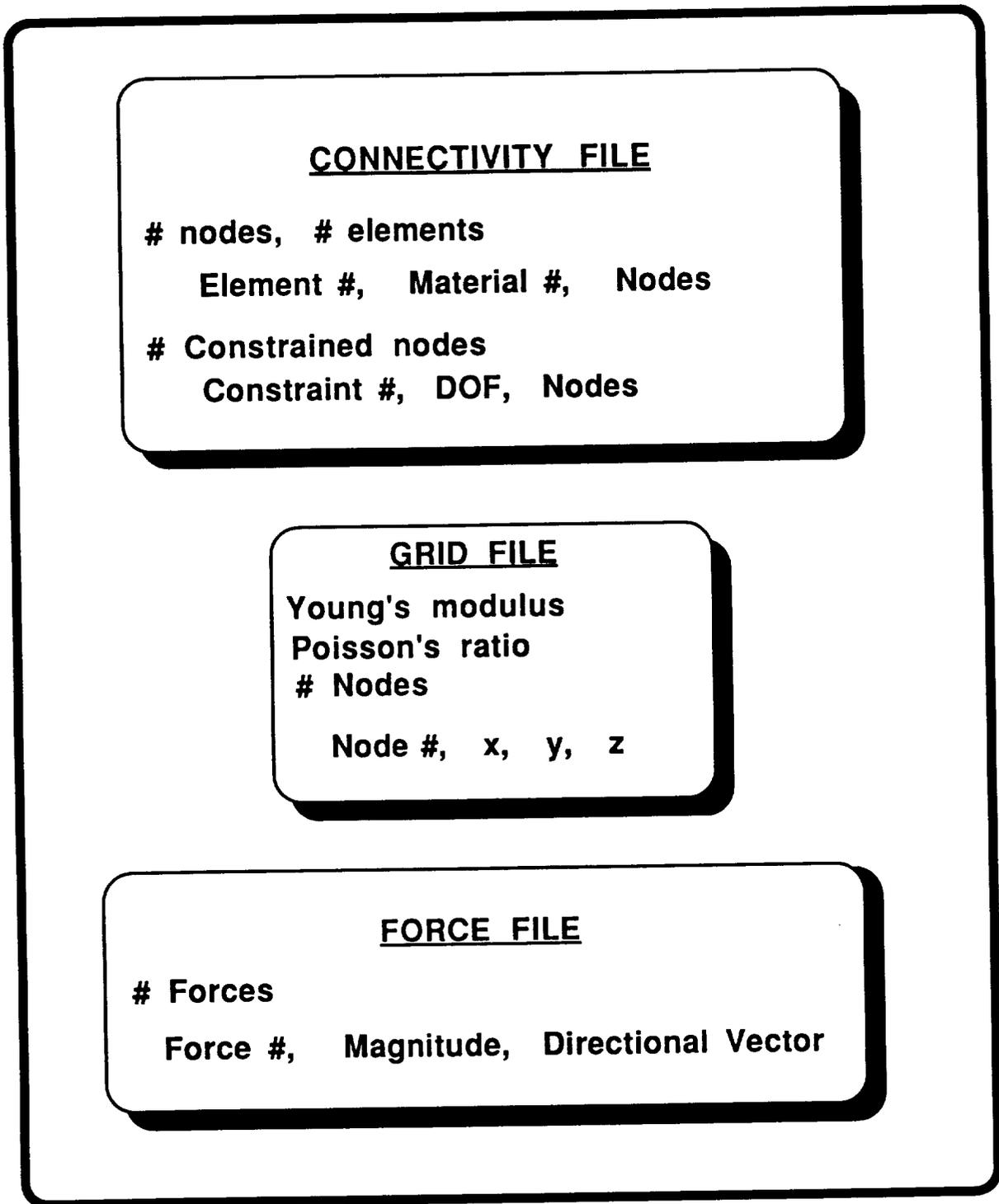
**FIGURE 13**  
**Boundary Discretization**



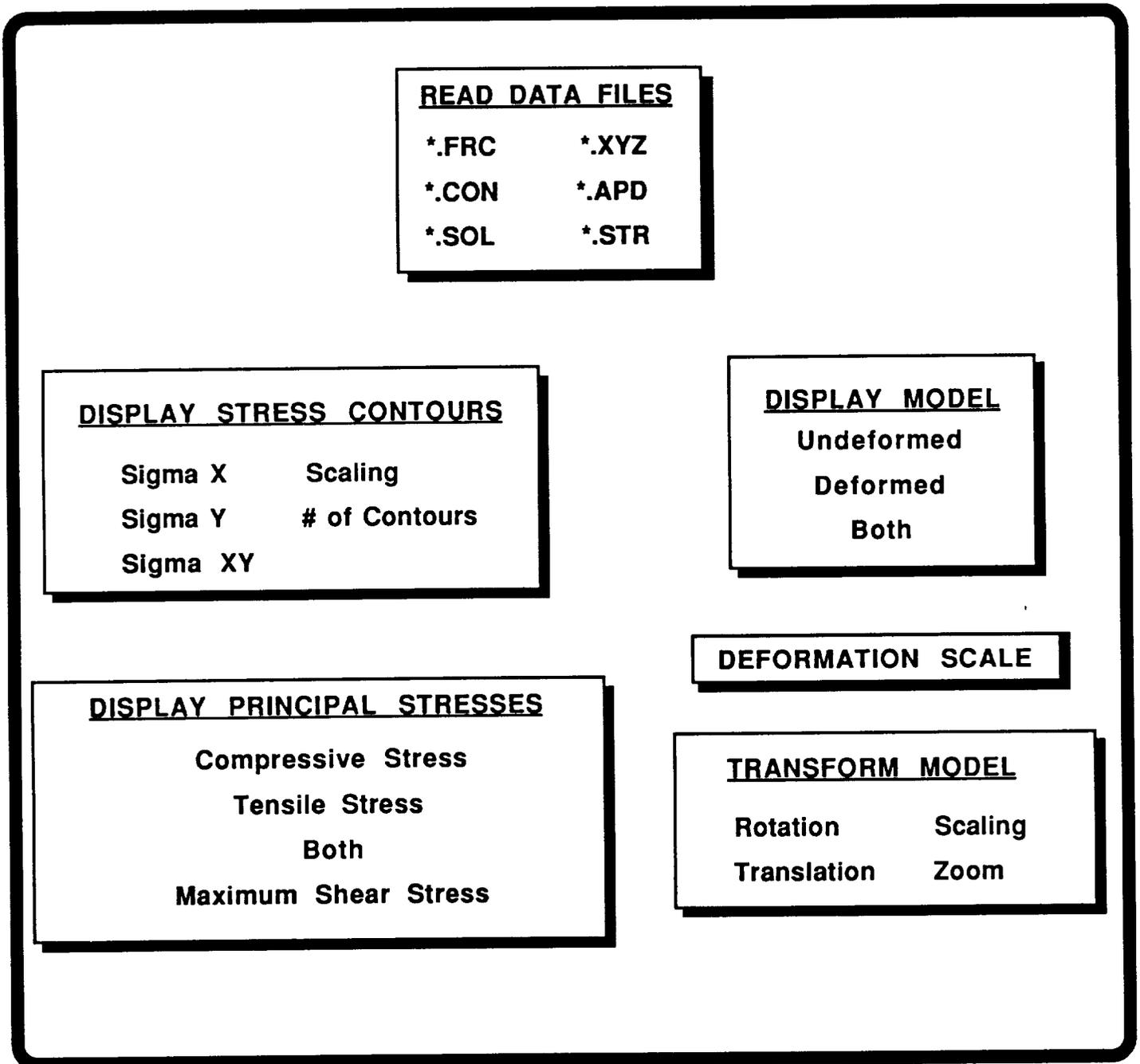
**FIGURE 14**  
**Discretization of the**  
**geometric model into quadrants**



**FIGURE 15**  
**Re-positioning quadrant**  
**vertices using smoothing**

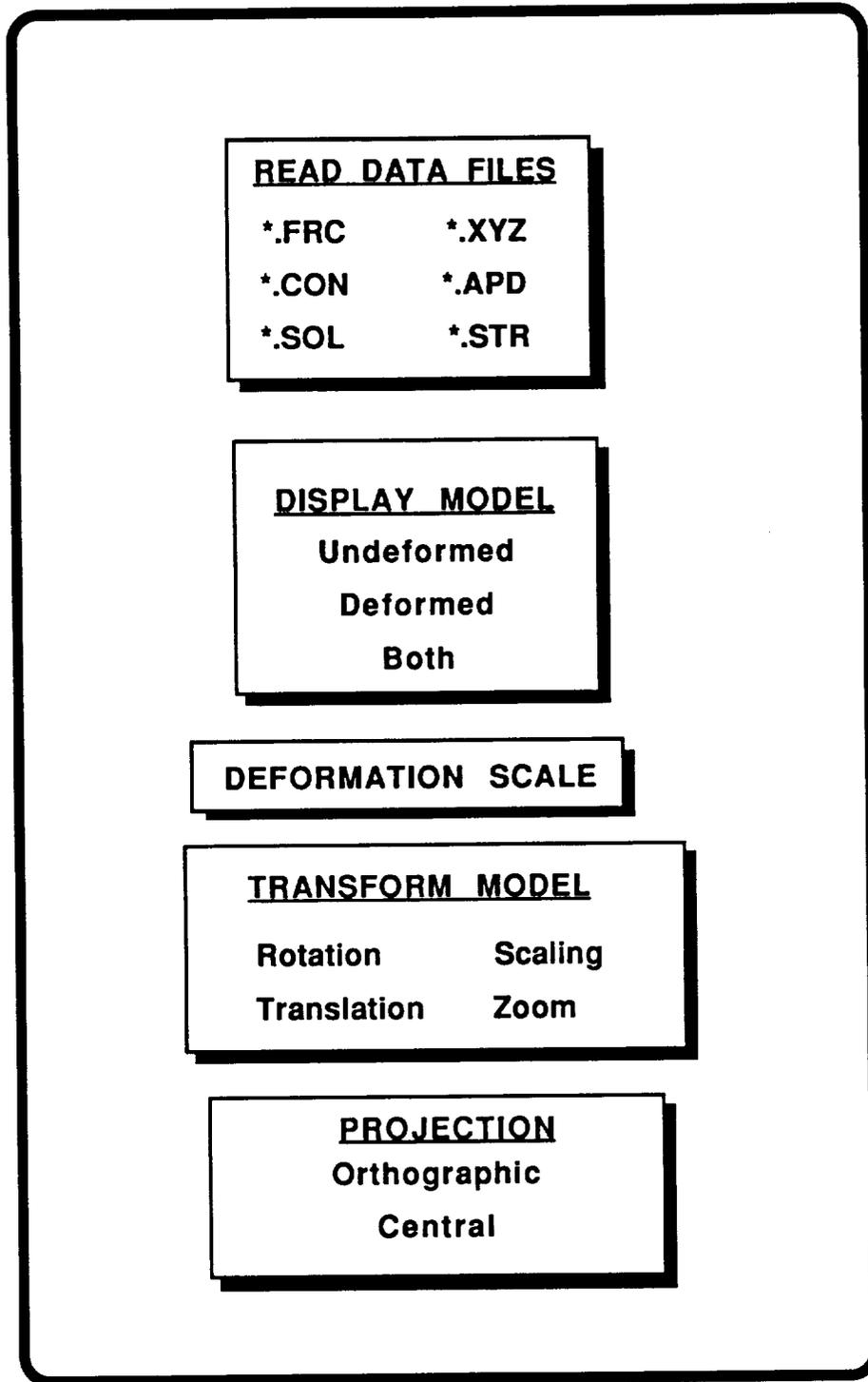
**FIGURE 16**

Files output from NASTRAN interface in XPFEM input format

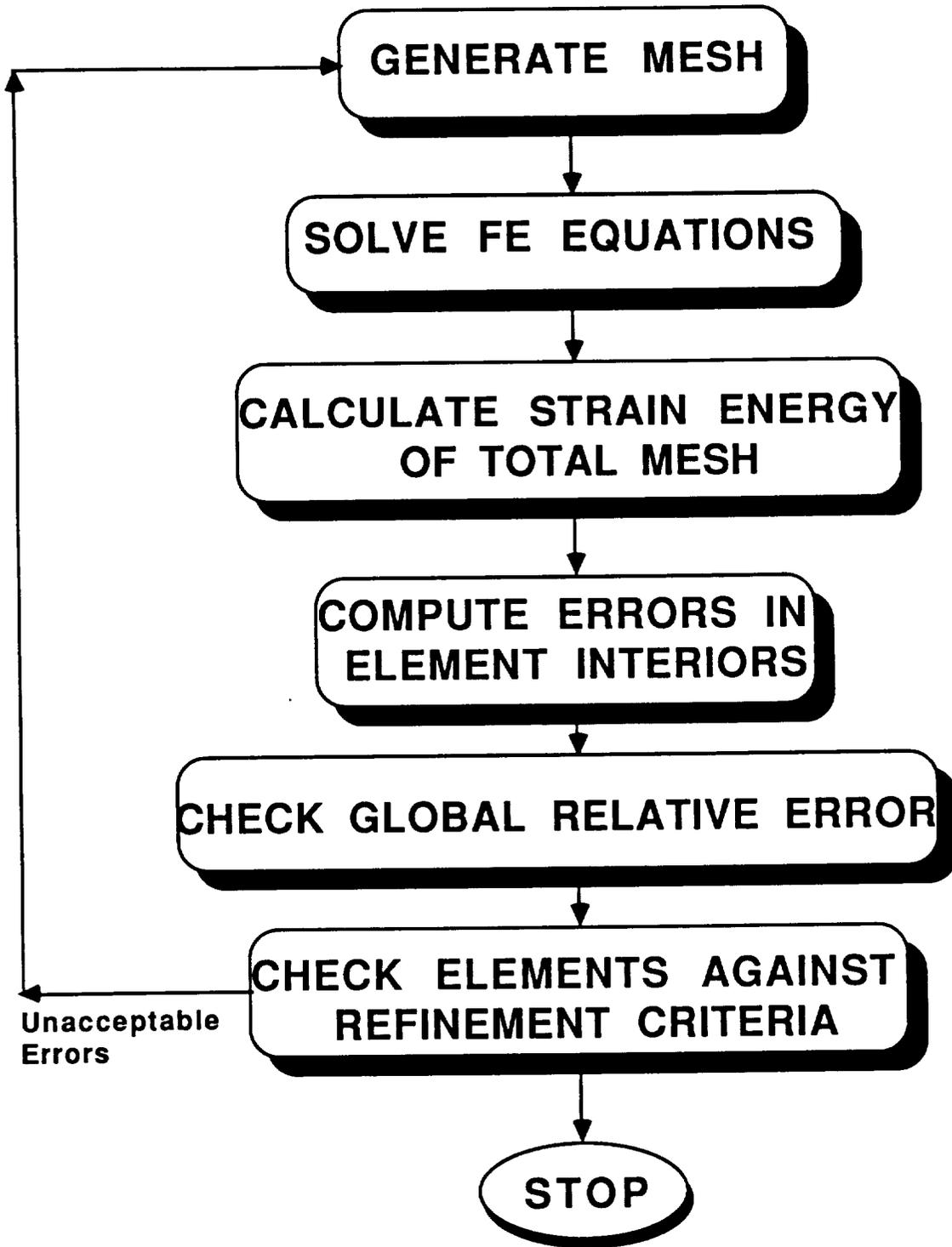


**FIGURE 17**

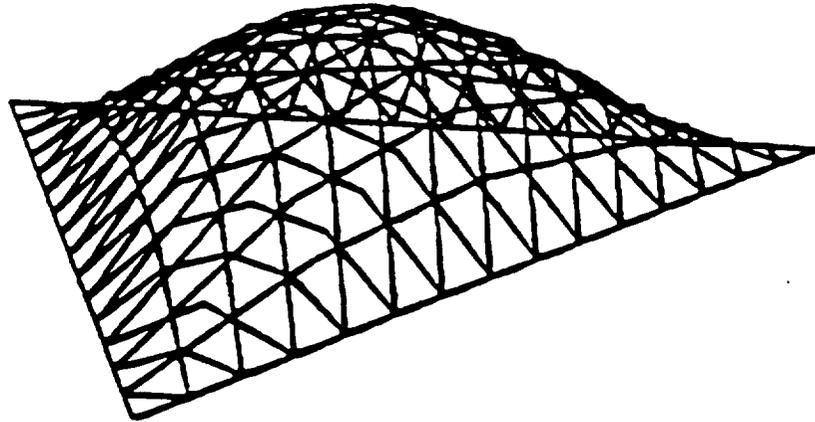
Capabilities of XPOST.2d,  
the two-dimensional post-processing module of XPFEM



**FIGURE 18**  
Capabilities of XPOST.3d,  
the three-dimensional post-processing module of XPFEM



**FIGURE 19**  
Adaptive Analysis algorithm implemented in XPFEM



$$\psi_1 = \xi_1 \xi_2 \xi_3 = \xi_1 \xi_2 (1 - \xi_1 - \xi_2)$$

$$\psi_2 = \xi_1 \psi_1 = \xi_1^2 \xi_2 (1 - \xi_1 - \xi_2)$$

$$\psi_3 = \xi_2 \psi_1 = \xi_1 \xi_2^2 (1 - \xi_1 - \xi_2)$$

$$E_j = C_1 \psi_1 + C_2 \psi_2 + C_3 \psi_3$$

**FIGURE 20**

**Bubble functions used for error interpolation in the adaptive analysis module**

**Element errors calculated in energy norm :**

$$\|E_i\| = \left( \int_{\Omega_i} (BE_i)^T D BE_i dx dy \right)^{1/2}$$

**Total error :**

$$\|E\|^2 = \sum_{i=1}^N \|E_i\|^2$$

**N = number of elements**

**Approximate relative percent error:**

$$\bar{\eta}_k = 100 \times \left\| \frac{(U + E) - U}{(U + E)} \right\|_k$$

**FIGURE 21**

**Error calculations for adaptive analysis**

## XPFEM: A TRANSPUTER BASED FINITE ELEMENT WORKSTATION

The primary objective of this effort, a turnkey transputer based finite element workstation, required integration of the previously described pre-processing, solution, and post-processing modules. Success towards achieving this objective may be measured by the capabilities of the resulting prototype system. The integrated prototype enables an analyst, using mouse and keyboard input, to create and mesh a geometric model, or interface and input a NASTRAN displacement model, compute and solve the resulting finite element equations and post-process resulting data to display deformed geometry, principal stresses, and stress contours or perform error calculations for adaptive analysis.

Integration of the various components of XPFEM on the transputer network was accomplished using menu driven procedures. A main menu (Figure 22) enables various analysis modules to be selected. Each module contains a system of checks to insure that the proper data files are available; if not, control is returned to the main menu and another selection is required. Detailed use of XPFEM is discussed in Appendix A, XPFEM User's Manual.

Screens from FINITE QUADTREE are shown in Figures 23 and 24. In Figure 23, the mesh generation menu with a geometric model is displayed. This figure not only displays the various utilities available for generating a mesh, but also displays the general working menu format for pre- and post-processing in XPFEM. Also seen in Figure 23 are the markers indicating the mesh control parameters for the vertices, edges, and face of the geometric model. The optional background grid can be refined and the workspace scale adjusted in module WORKSPACE GRID. Figure 24 displays a mesh and refinements produced using the mesh generation module. The initial mesh was created with default modeling parameters for the vertices, edges, and face of the geometric model. The refinements were achieved by varying the modeling parameters of specific vertices and edges. Relative and absolute mesh control points unrelated to the geometry are also available for mesh refinement. For a relative control point, the mesh refinement level at the point equals the sum of the point's modeling parameter and the modeling parameter of the surrounding face. For an absolute control point, the modeling parameter of the point is the level of mesh refinement.

Computational times for computing, assembling, and solving finite element equations on the 32 processor network are shown in Figures 25-28. Figure 25 is a comparison of the computational and assembly times of the global stiffness matrix of the NASA SSME turbine blade model. The blade model contains 1025 8-node brick elements and 4500 independent degrees-of-freedom. The results show that, for the scalar operation of computing and assembling the global stiffness matrix, the XPFEM system outperforms the NASA/Lewis Cray X-MP24 by a factor of two. A comparison of the solution times for the system of finite element equations of the blade model is shown in Figure 26. In this vectorizable operation, COSMIC NASTRAN on the Cray X-MP24 was six times faster than the XPFEM system. Overall, the total solution time (computation, assembly and solution of the finite element equations) of the turbine blade model on the XPFEM system was one-third the speed of the Cray X-MP24 and 70 times faster than a VAX 11/780 executing COSMIC NASTRAN (Figures 27 and 28).

Results from XPPOST, the post-processing module of XPFEM, are displayed in Figures 29-32. Figure 29 shows the undeformed turbine blade model displayed in wire-frame mode and Figure 30 shows both undeformed and deformed models concurrently. The display of the deformed model also depicts the applied constraints and forces. Figure 31 consists of stress contours in a two-dimensional C-clamp constrained at three points on the lower right and loaded with a point load at the upper right corner. The contours displayed in Figure 31 are those for shear stress. Normal stress contours in the x and y directions are also available in XPFEM. The number of contour levels in Figure 31 is 30; this parameter is set by the user and may range from 2 to 224.

Principal stresses for the two-dimensional C-clamp model are shown in Figure 32. This figure displays the calculated tensile and compressive principal stresses at the element centroid. The stress bars indicate the stress orientation with respect to the x-axis. Their lengths are normalized so that the greatest principal stress is equal to the averaged side length of the largest element. XPFEM options also permit tensile, compressive, or maximum shear stresses to be displayed separately with or without the finite element model.

Adaptive analysis results are presented in Figure 33. The sequence of meshes displayed in Figure 33 indicates the future capability of the XPFEM adaptive analysis module to significantly refine a mesh at a singularity and to distribute mesh error

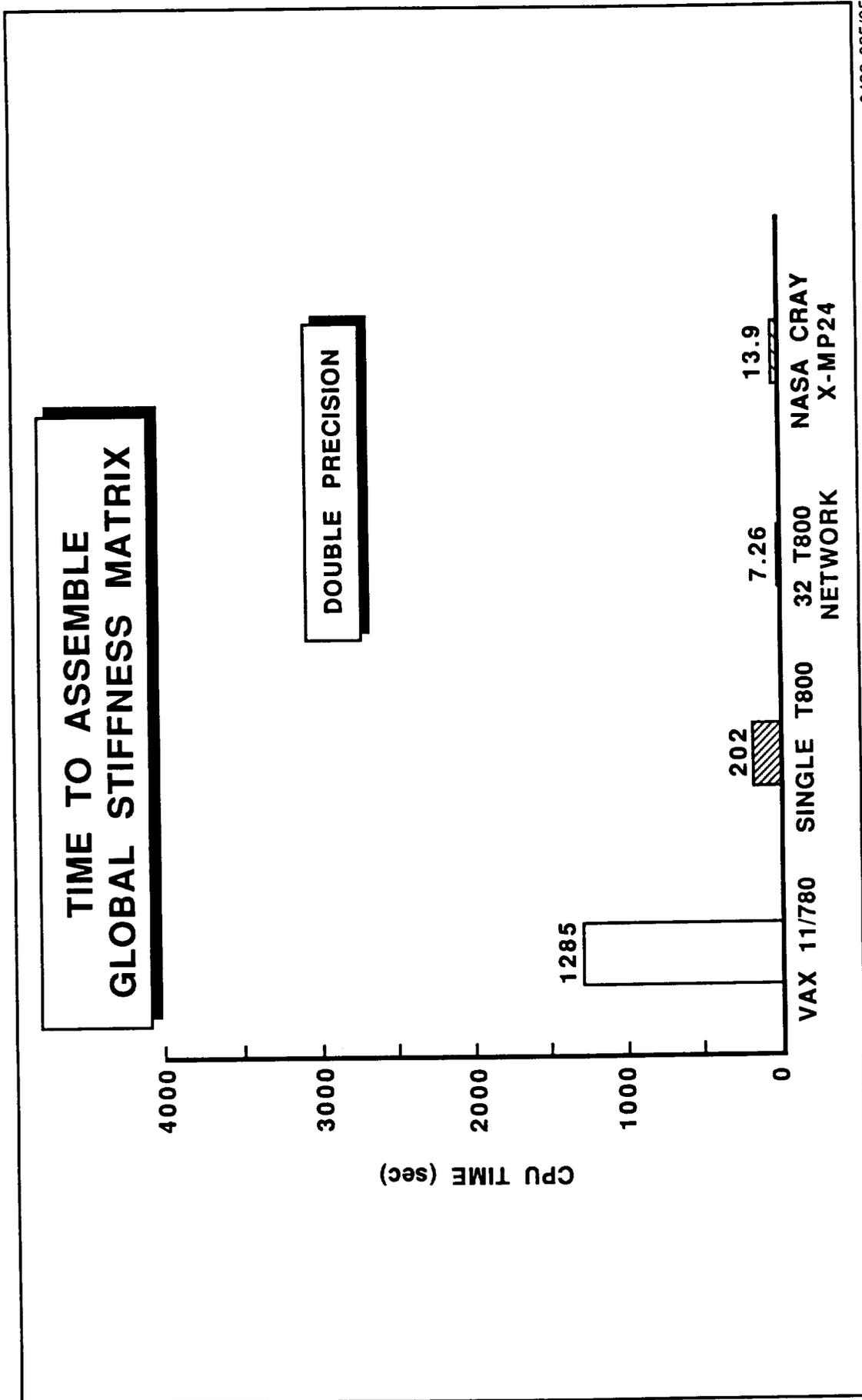
following this singularity refinement. Currently, this module, which has been tested and implemented on a MicroVAX, is implemented in XPFEM but a mesh generation error prevents it from being fully robust.

<b>XPFEM</b> <b>version 89.1</b>	
<b>PROMPT/ERROR:</b>  	
<b>CURRENT PROBLEM:</b> SPARTA	<b>SPECIFIER:</b> EXAMPLE1
NEW PROBLEM	OLD PROBLEM
PROBLEM TYPE	WORKSPACE GRID
FINITE QUADTREE MODEL GEOMETRY	
MATERIAL PROPERTIES	ANALYSIS ATTRIBUTES
FINITE QUADTREE MESH GENERATOR	
ADAPTIVE ANALYSIS	
OUTPUT FOR ANALYSIS	READ NASTRAN FILE
2-D SOLVER	3-D SOLVER
2-D POST PROCESSOR	3-D POST PROCESSOR
EXIT	

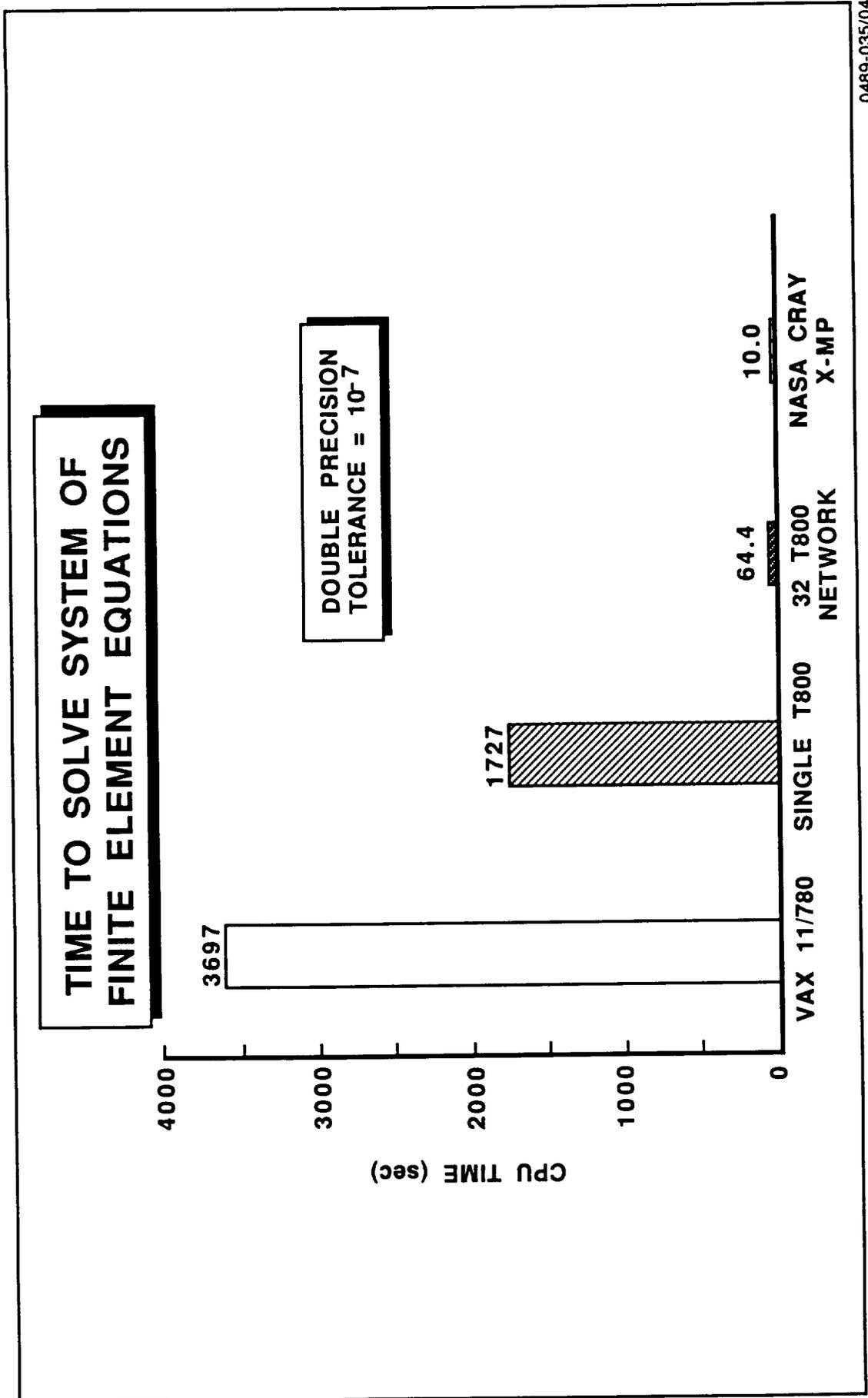
**FIGURE 22**  
**Main menu for XPFEM**

Figure 24: Mesh with Refinements, Page 55

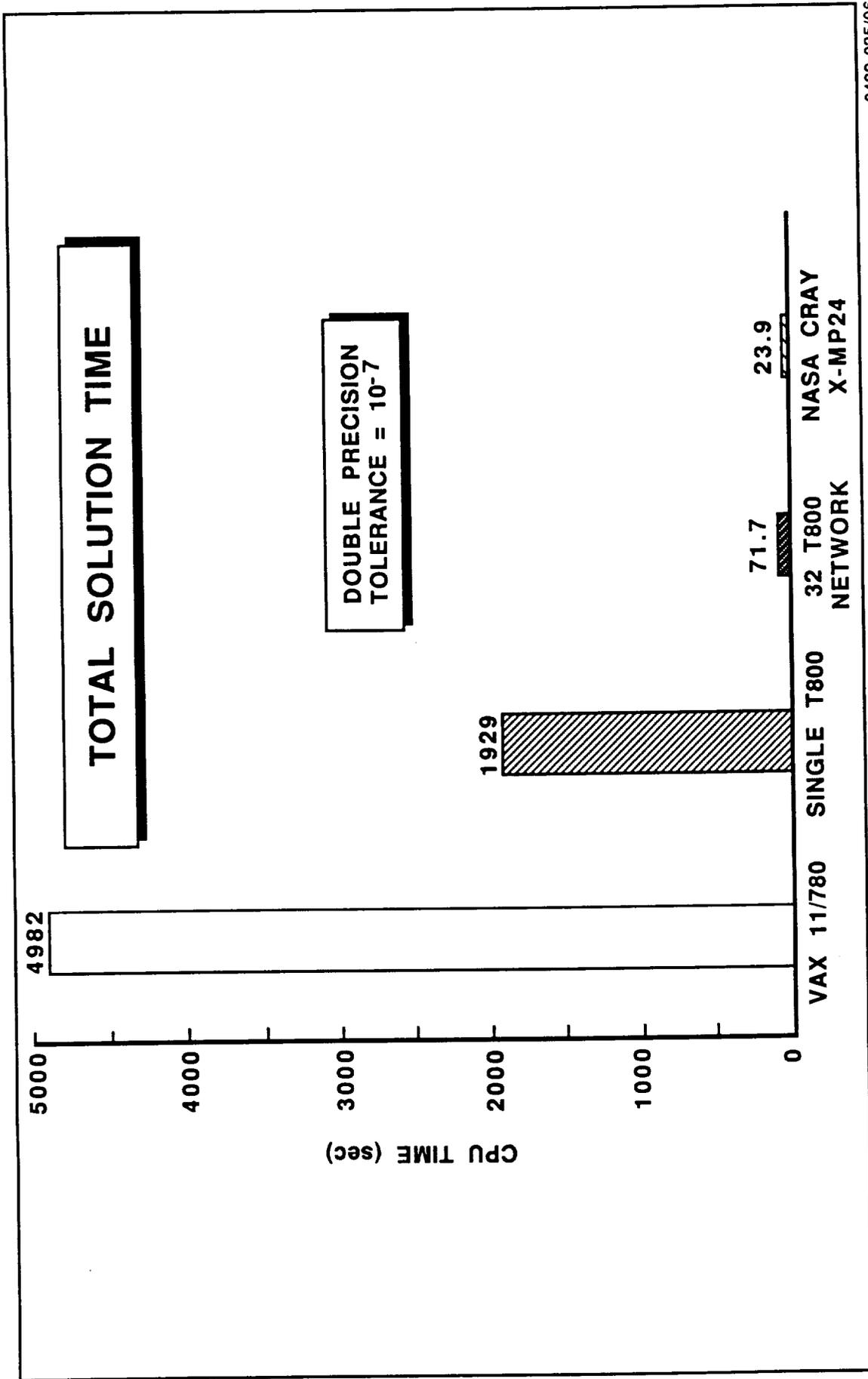
Figure 23: Geometric Model with Mesh Parameters, Page 54



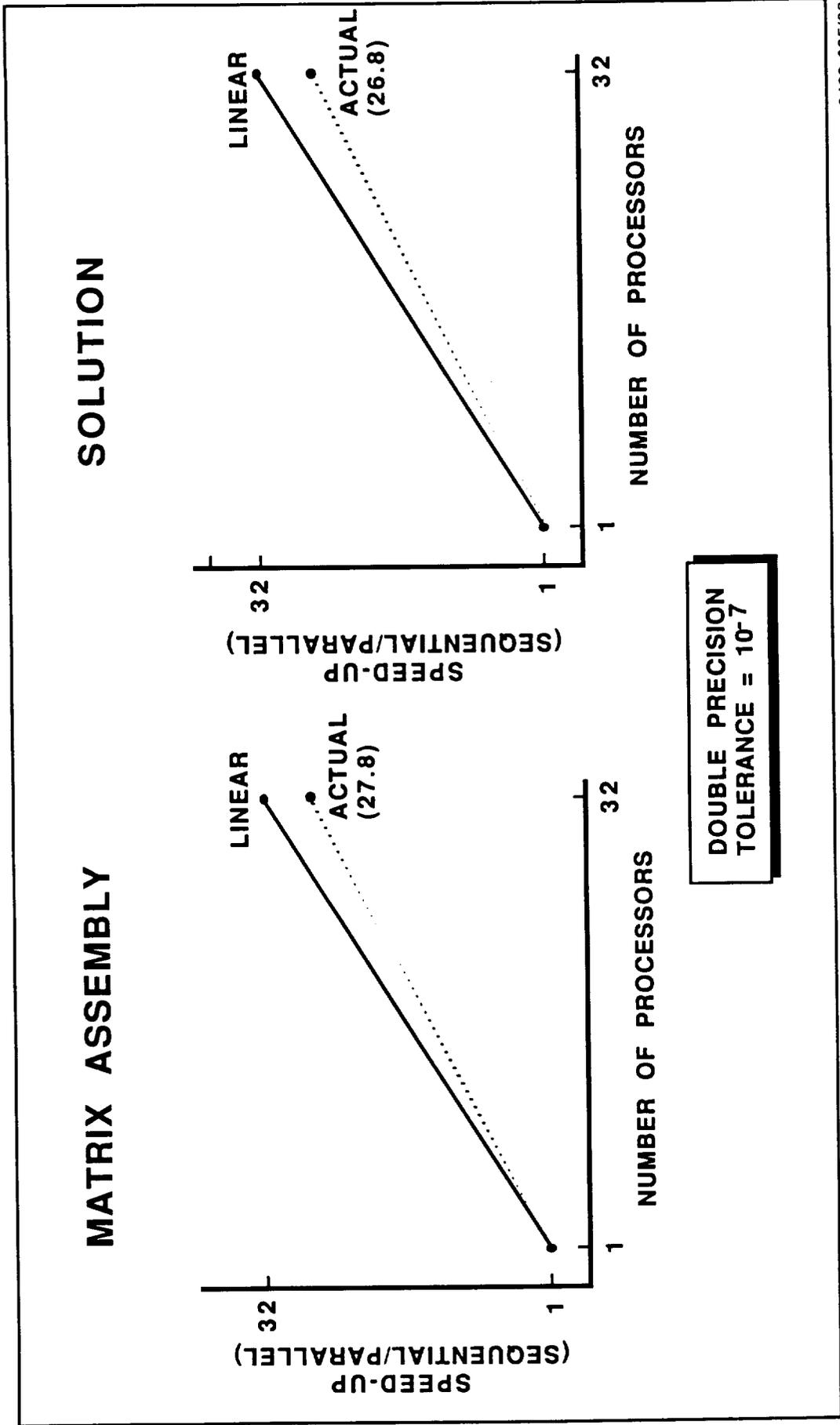
**FIGURE 25**  
**Assembly Time for SSME Turbine Blade**



**FIGURE 26**  
**Solution Time of Finite Element Equations for SSME Turbine Blade**



**FIGURE 27**  
**Total Solution Time for SSME Turbine Blade**



0489-035/08

FIGURE 28  
Speed-up for the Turbine Blade

01 01 01 01  
01 01 01 01  
01 01 01 01

01 01 01 01  
01 01 01 01  
01 01 01 01

01 01 01 01  
01 01 01 01  
01 01 01 01

01 01 01 01  
01 01 01 01  
01 01 01 01

01 01 01 01  
01 01 01 01  
01 01 01 01

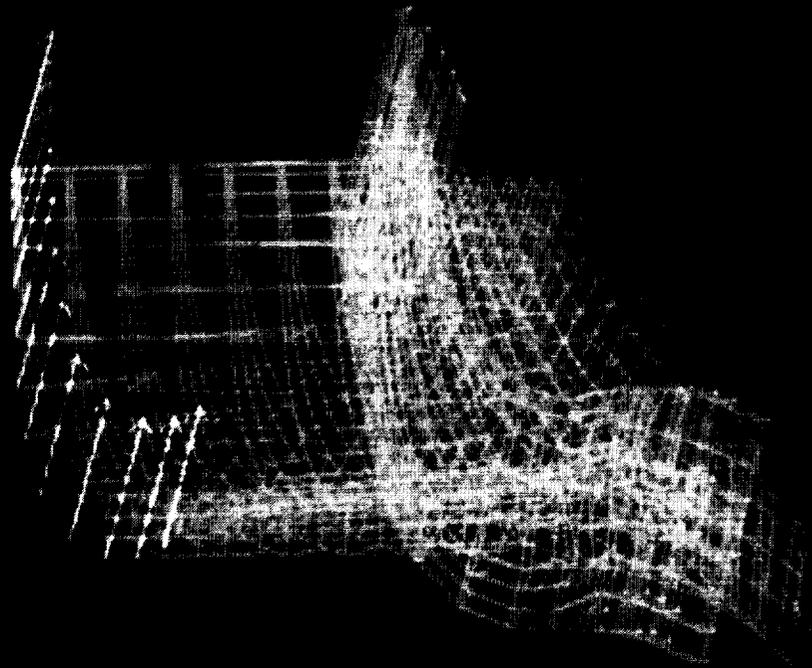


Figure 29: Space Shuttle Main Engine Turbine Blade, Page 60

001

002

003

004

TURBINE

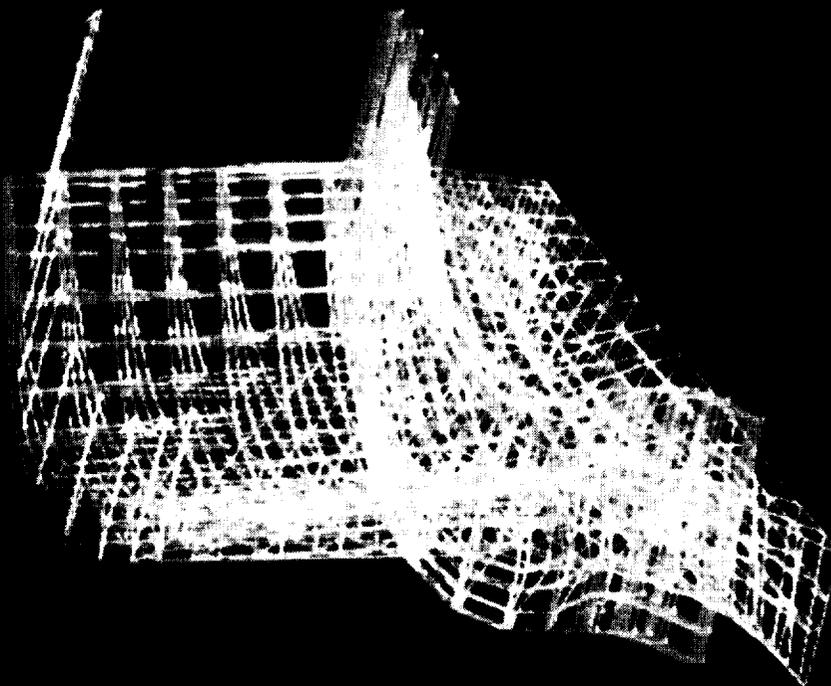


Figure 30: Deformed and Undeformed SSME Turbine Blade Page 61

FILED 6011

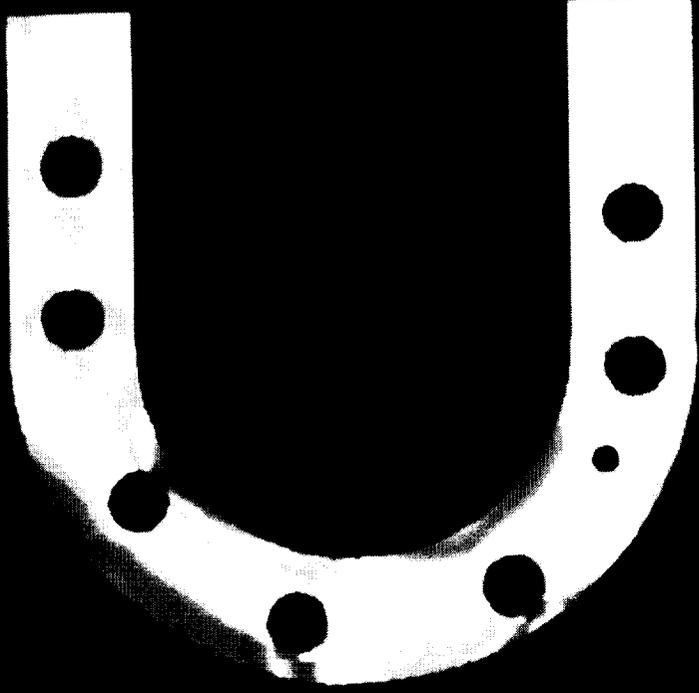
DATE

01/11/00

01/11/00

01/11/00

01/11/00



TYPE OF STRESS = SIGMA\_XX

MIN.STRESS = -1.4E+03      MAX.STRESS = 1.4E+04

Figure 31: Stress Contours in a C-clamp, Page 62



1000  
1000

1000  
1000

1000  
1000

1000  
1000

1000  
1000

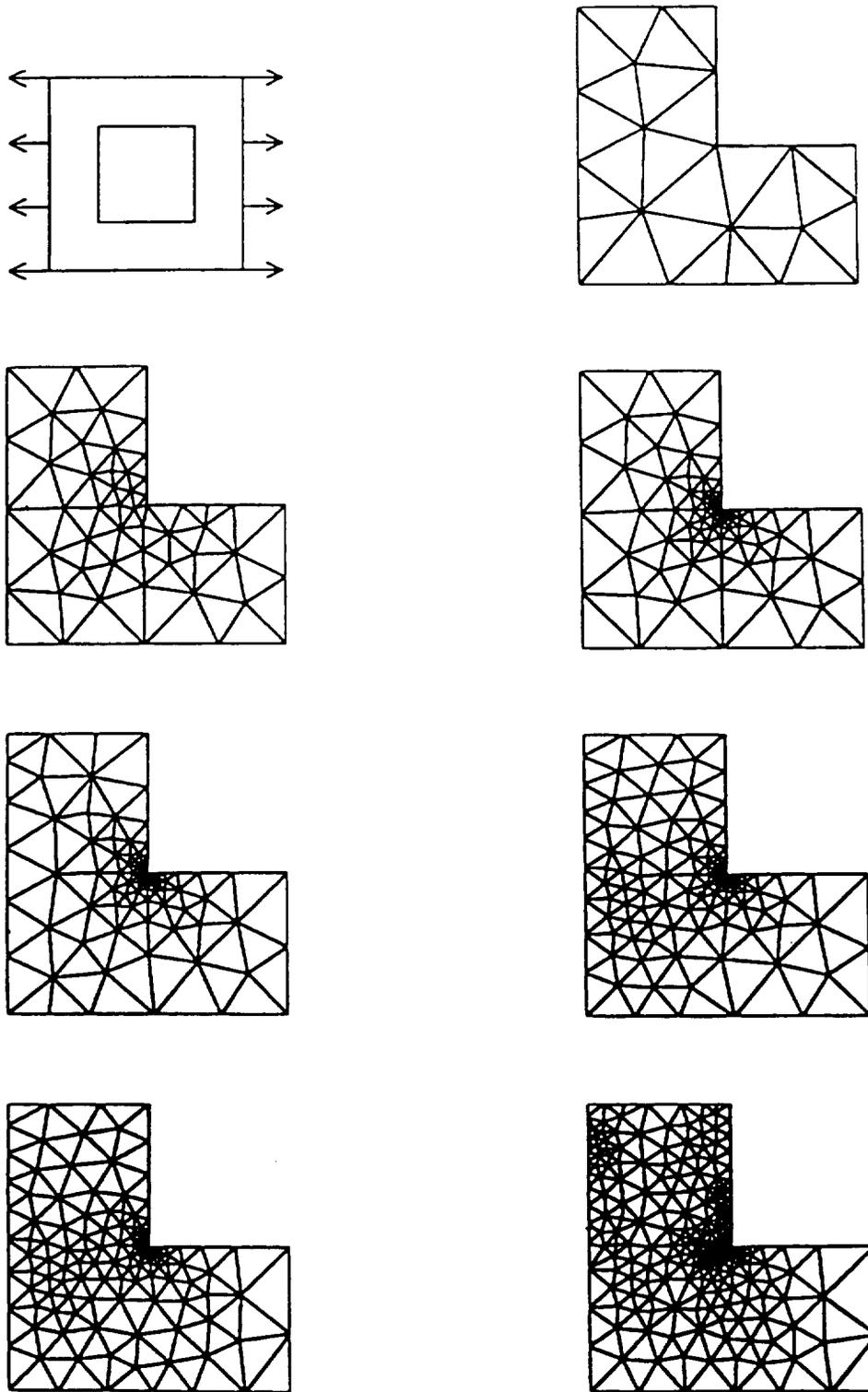
1000  
1000

1000  
1000

1000  
1000

TENSILE STRAIN

Figure 32: Principal Stresses in a C-clamp, Page 63



**FIGURE 33**

**Adaptive analysis example  
of a plate in uniform tension**

## **VI. CONCLUSIONS AND RECOMMENDATIONS**

The structural analysis system developed in this Phase II SBIR effort demonstrates the vast potential of a transputer based workstation. The essential foundation has been laid for a class of application-specific coprocessing hardware and software which can provide supercomputing analysis capability at dramatically lower cost (Figure 34). The major objective of demonstrating highly efficient parallel computation for problems common to demanding industrial-level applications was clearly achieved. This confirms the forecasted potential and strongly supports a recommendation for commercialization.

The delivered prototype version of XPFEM is limited to solving static, linear finite element problems with 20,000 or fewer degrees of freedom. Although the major foundation of an end-to-end parallel finite element workstation has been achieved, additional work is needed to expand XPFEM's capabilities to the full level expected by industrial users. It is recommended that additional key capabilities be incorporated that would require about twelve man-months of effort. These include non-linear, dynamic and thermal analyses, and the use of alternate out-of-core solvers for problems with hundreds of thousands of degrees of freedom.

During the development of XPFEM, a number of additional features were identified which could further expand or improve the capabilities of the workstation. It is further recommended that these enhancements and features be implemented to make XPFEM a fully capable, dramatically more cost-effective alternative to the finite element analysis systems currently running on existing supercomputers. Additional conclusions and a summary description of recommended enhancements and improvement features are provided below.

### **Enhancements to Pre-Processing**

#### **Improved Boundary Conditions**

The current user interface demonstrates transputer based implementation concepts but permits only the constraint and loading of vertices. Edge and face constraints are necessary additions to XPFEM and can be implemented easily. In

— addition, the menu for boundary and initial conditions is being evaluated to determine if inputs can be made with fewer menu selections.

### — **Three-Dimensional Mesh Generation**

— As defined in the SBIR Phase II Statement of Work, the current mesh generator runs only on two dimensional models. SPARTA recently received from RPI the FINITE OCTREE code (in FORTRAN) for 3D mesh generation, and is evaluating portions of it to determine how best to implement the code in occam. A three dimensional capability is important to many applications and either direct integration or bundling of a third-party 3D mesh generator is recommended.

### — **Parallel Mesh Generation**

— A peculiar aspect of XPFEM is that it takes longer to mesh large problems than it does to solve them. This discrepancy can be corrected with the use of parallel mesh generation, a new concept being investigated by the RPI researchers (Appendix H). Three dimensional models in particular will benefit from distributed mesh generation since 3D models require much more time to mesh.

### — **Node Renumbering**

— Large bandwidths have an adverse effect on nearly all linear equation solvers. Although not affected by bandwidth in the typical manner of increased storage and computational requirements, the JCG solver demonstrated that large bandwidths sharply increase communication, and therefore, solution times.

— Renumbering routines can reduce bandwidth to near optimum levels and are essential in finite element programs. XPFEM has an excellent renumbering algorithm but it is integrated into the mesh generator and therefore only works on models generated with FINITE QUADTREE. Externally created models are solved with their original numbering schemes. In order to efficiently solve models built on other systems, a stand-alone renumbering routine must be added to XPFEM. Renumbering codes are common so this addition would not require intensive effort.

## Enhancements to the Solver

The sparse Jacobi-conditioned Conjugate Gradient (JCG) equation solver developed in the Phase II research is innovative, fast and efficient but is limited to solving symmetrical systems of equations that reside entirely in core memory. Different solution techniques must be added to handle more complex boundary conditions and marginally stable problems. Also, despite the sparse storage implementation, FE models can always be made to exceed in-core memory capability, so the out-of-core solver described in Section III must be added. Some alternative large-capacity solution methods are described in *Increasing Solver Capacity*. Finally, a number of minor modifications to the sparse JCG solver were identified during the development of XPFEM which would markedly increase its performance. These modifications are described under *Fine-Tuning the JCG Solver*.

### **Increasing Solver Capacity**

Methods are available to raise the current in-core FE problem size limit from 20,000 degrees of freedom to well over 100,000 degrees of freedom. Several of these methods need some form of external storage to save intermediate results and can be considered hybrid in-core/out-of-core methods. Some methods involve advanced concepts in FE analysis and could take considerable coding effort while other, more pedestrian approaches could be easily implemented but would not be efficient.

#### Recomputing the GSM

The simplest method of expanding capacity is to compute, rather than store, terms of the global stiffness matrix as they are needed in the computations. This approach is highly inefficient--large problems will require tens of thousands of redundant GSM computations--but is also easy to implement in XPFEM and provides for a quick link to extremely large FE problems.

#### Multigriding

Multigriding, originally proposed to accelerate finite difference solutions, has tremendous potential for speeding up FE analysis. In this method, a densely meshed model is solved by iteratively solving increasingly coarser meshes on the same model, and carrying solution approximations back and forth (via interpolation and

extrapolation) between the different mesh granularities. Order of magnitude improvements in solution times can be realized with multigridging. Disadvantages are that memory requirements are doubled, and that software development is difficult.

In the context of increasing solver capacity, a multigrid-like approach can be used to solve very large problems by remeshing the model coarsely so that it can be solved entirely in-core, then extrapolating the coarse solution to the full mesh. The extrapolated solution will then be an excellent guess to the actual solution, so time consuming out-of-core iterations will be minimized.

### Block Iterative Methods

The usual *calculate versus communicate* dilemma in parallel programming becomes even more complicated with the addition of external storage devices. Mass data storage devices such as tapes and disk drives are, by their physical (rather than electronic) natures, extremely slow on a microprocessor's time scale and therefore introduce stringent design parameters to parallel equation solvers that use them. The programming question expands to *calculate v. interprocessor communicate v. external storage communicate*. The method described in *Recomputing the GSM* abandons all form of communication and calculates everything repeatedly. The multigrid method uses external storage only as an intermediate step between relatively long in-core compute cycles. Fully out-of-core methods use external storage as an integral part of the solution process.

Alternative fully out-of-core methods to the one presented in Section III are the parallel block Conjugate Gradient methods described in [12,13]. These methods preserve the iterative nature of the current XPFEM solver and would therefore integrate perfectly with adaptive analysis and multigridging. However, several load balancing issues remain to be resolved.

### Fine-Tuning the JCG Solver

The current implementation of the JCG solver is fast and efficient but a few modifications can improve it even further. Perhaps the most significant modification would be an improved preconditioner. More powerful preconditioners generate larger preconditioning matrices which result in more floating point operations per iteration, yet maintain the identical number of communication steps per iteration. As a

consequence, the calculate-to-communicate ratio and, therefore, efficiency, increases. In addition, improved preconditioning reduces the number of iterations. These features combine to make a vastly more efficient parallel solver. Other refinements include adding an efficient renumbering scheme, investigating more uniform load balancing methods, and rewriting the pointer tables to use less memory.

### Preconditioning by Blocks

The JCG solver operates on a term-by-term basis in which degrees of freedom are operated on one at a time. More importantly, the Jacobi preconditioning matrix has only one term on each row. If the solver were to operate on small blocks of terms, and the preconditioning matrix consisted of blocks of terms rather than just one term on the diagonal, a much stronger preconditioning would occur.

It is suggested that the blocks be multiples of the dimension of the problem (a multiple of a patch) since the code already stores data in this manner. The small block method would incur a minimal penalty in storing the larger preconditioning matrix. Blocks on the order of 4 times the dimension will need an additional 20 Kbytes per processor to store the preconditioning matrix.

### Polynomial Preconditioning

Most Conjugate Gradient preconditioners rely on some form of Gaussian elimination to create a preconditioning matrix. Unfortunately, these schemes are not efficient on banded or sparse matrices when distributed over many processors. The polynomial preconditioning method as described in [14], however, is based on a sequence of matrix-vector multiplies (as the JCG method itself is) which are highly efficient on multiprocessor systems. This preconditioning method appears ideally suited for XPFEM.

### Improved Load Balancing

Despite the term-wise load distribution, some processors do as much as 5% more work than other processors. This slight discrepancy reduces speed-ups, since the unloaded processors become completely idle while their more loaded counterparts finish their tasks. The nature of the load imbalance can be revealed by placing timing calls after each math routine.

## Rewriting Pointer Tables

The pointers required for the sparse matrix assembly and matrix multiplication routines take nearly 20% of each processor's memory. This value can be reduced sharply if some of the tables were written as one dimensional vectors with entry pointers rather than the current two dimensional arrays. This rewrite, however, could slightly increase overhead and would take several weeks of coding.

## Enhancements to Post-Processing

The interpretation of finite element solutions depends on a large extent to the form in which the solutions are presented. Proposed enhancements include illustrating structurally critical stress contours, and speeding up portions of post-processing with parallel computation.

### **Plasticity and Zero Stress Contours**

To assist the structural designer in determining whether or not the model in question can withstand design loads, specially colored contour bands can be added which indicate regions of plasticity. Other special contour bands can indicate regions in the model which experience negligible stresses (within a user-defined tolerance). The addition of these features would involve a minimal implementation effort and would be immensely practical.

### **Parallel Contour Computation**

The calculation of stress contour levels can take a considerable amount of time (20 seconds or more) for large models. These computations are currently done on the host transputer, but can easily be distributed since the computations are element-based and therefore completely parallel. The addition of parallel contour computation would make this portion of post-processing appear virtually instantaneous.

### **Three Dimensional Stress Contours**

Current post processing can only display stress contours on two dimensional models. Stress contours are such a vital part of the FE analysis, however, that one of the first additions recommended for XPFEM is a three dimensional stress contour capability.

## **Additional Features**

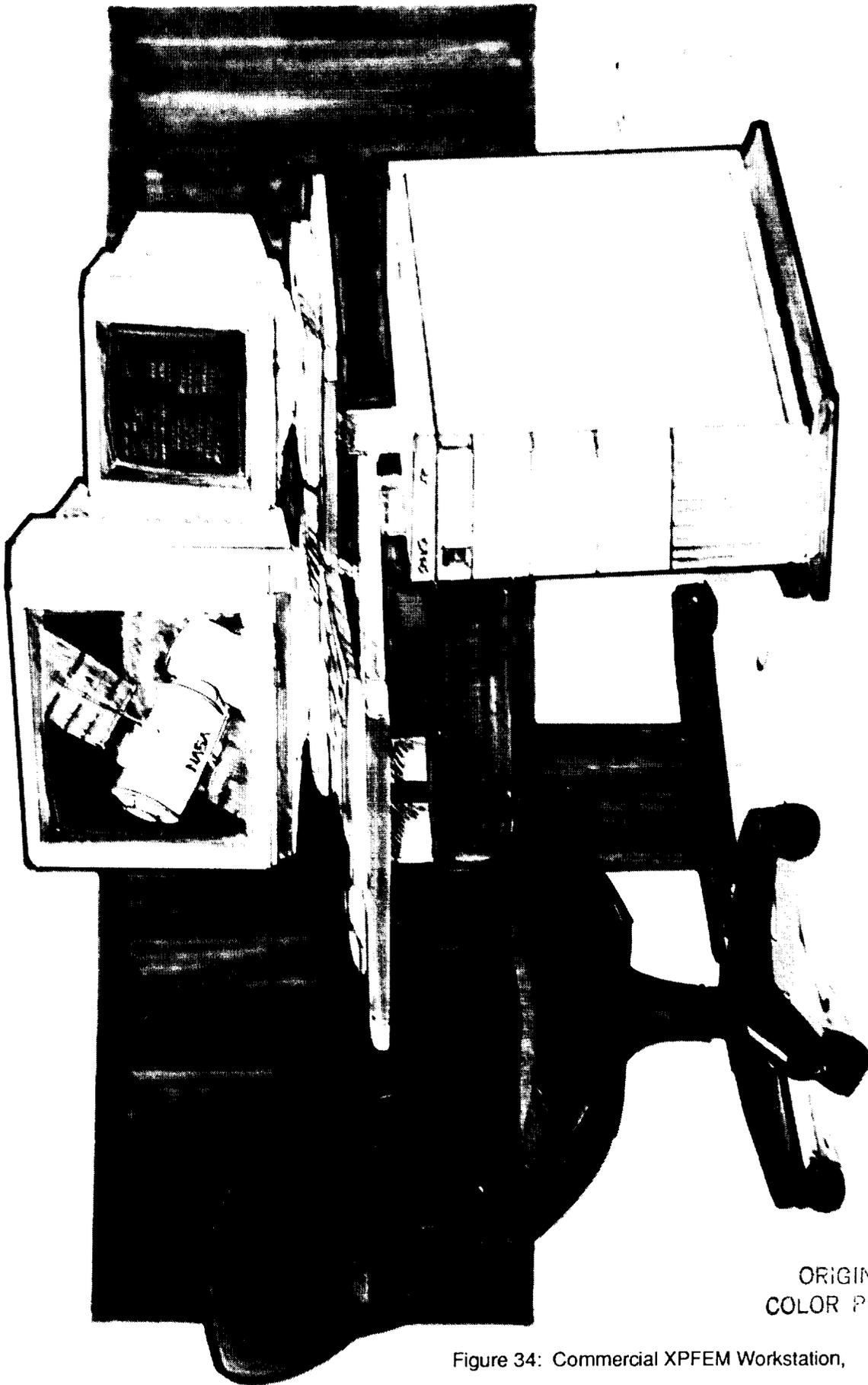
Although XPFEM currently solves only static, linear FE problems, the software foundation for a comprehensive structural analysis workstation is firmly in place. Additional capabilities such as thermal, non-linear and time dependent solutions can be added directly since the techniques used to solve these problems can be written as straight-forward extensions to the existing static, linear solver. Significant additions to input and output routines will have to be made to accommodate new material properties and initial conditions, but the core structure (and performance) of the solver will remain unchanged.

Additional features have been divided in two classes, based on how quickly they can be implemented on XPFEM. The first class of features includes transient, modal and thermal analyses. It is recommended that these be incorporated into XPFEM as an immediate follow on effort to produce a commercial product.

The second class of features includes non-linear analysis, frequency response and computational fluid dynamics. These features involve substantial software modifications. It is recommended that these be added after XPFEM is introduced to the market.

In summary, a clearly significant capability for parallel acceleration of linear, static structural engineering analysis is in hand. A critical evaluation of the implemented system relative to industrial application requirements has been made and a number of enhancements identified to meet the needs of over 95% of the structural analysis user base. A phased implementation approach has been outlined based on a summary description of each additional feature.

If accomplished in a timely manner, the resulting system can provide affordable, desktop supercomputing for finite element analysis to a wide segment of the ten billion dollar structural analysis market place. The impact of dramatically improved turn-around and wider availability on engineering productivity is potentially revolutionary.



ORIGINAL PAGE  
COLOR PHOTOGRAPH

Figure 34: Commercial XPFEM Workstation, Page 72

## VII. REFERENCES

- [1] Favnesi, J., Danial, A., and Bower, M., "Transputer Based Finite Element Solver Phase I SBIR," Final report for NASA contract NAS3-25126, NASA/Lewis Research Center, Cleveland, OH, 1987.
- [2] Rehak, D. R., Keirouz, W. T., Hendrickson, C. T., Cendes, Z. J., "Trends in Engineering Software and Hardware: Evaluation of Finite Element System Architectures," *Computers & Structures*, Vol. 20, No. 1-3, pp. 17-29, 1985.
- [3] Law, K. H., "A Parallel Finite Element Solution Method," *Computers & Structures*, Vol 23, No. 6, pp. 845-858, 1986.
- [4] King, R. B., and Sonnad, V., "Implementation of an Element-by-Element Solution Algorithm for the Finite Element Method on a Coarse-Grained Parallel Computer," *Computer Meth. Appl. Mechanics and Engineering* 65, pp 47-59, 1987.
- [5] Gustafson, J. L., Montry, G. R., and Benner, R. E., "Development of Parallel Methods for a 1024-Processor Hypercube," *SIAM Journal on Scientific and Statistical Computing*, Vol. 9, No. 4, July 1988.
- [6] Rodgers, Jack, Department of Mathematics, Auburn University, informal communication, July, 1988.
- [7] Group discussion, 1989 North American Transputer Users Group Spring Conference, University of Utah, April, 1989.
- [8] Bjørstad, Petter E., "A Large Scale, Sparse, Secondary Storage, Direct Linear Equation Solver for Structural Analysis and its Implementation on Vector and Parallel Architectures," *Parallel Computing* 5, pp. 3-12, 1987.
- [9] Zienkiewicz, O. C., *The Finite Element Method*, McGraw-Hill Book Company (UK) Limited, 1986, pp. 162-164.

- [10] Stasa, F. L., *Applied Finite Element Analysis for Engineers*, Holt, Rinehart and Winston, 1985, pp. 337-339.
- [11] Baehmann, P. L., "Automated Finite Element Modeling and Simulation," Ph.D. dissertation, Rensselaer Polytechnic Institute, Troy, NY, 1989.
- [12] Kowalik, J. S., and Kumar, S. P., "An Efficient Parallel Block Conjugate Gradient Method for Linear Equations," *Proc. 1982 Int. Conf. Parallel Processing*, pp. 47-52.
- [13] O'Leary, D. P., "Parallel Implementation of the Block Conjugate Gradient Algorithm," *Parallel Computing*, 1987, pp. 127-139.
- [14] Johnson, O. G., Micchelli, C. A., and Paul, G., "Polynomial Preconditioners for Conjugate Gradient Calculations," *SIAM Journal of Numerical Analysis*, Vol. 20, No. 2, April 1983, pp. 362-375.

**APPENDIX A****XPFEM USER'S MANUAL**

Execution of XPFEM is initiated from the IBM PC-AT compatible host computer by performing the following steps:

1. At the MS-DOS prompt `C:\>`, move to the XPFEM directory:

```
C:\> cd \fes12\disk
C:\FES12\DISK>
```

2. Invoke the Transputer Development System (TDS):

```
C:\FES12\DISK> tds2
... XPFEM.TOP
```

3. Enter the XPFEM top level fold:

<7 on the numeric keypad>

4. Move the cursor to the XPFEM executable code fold:

<down arrow several times, until the cursor is  
on ...EXE XPqtrees>

5. Load XPFEM into the host transputer's memory:

**F 5**

6. Run XPFEM:

**F 6**

XPFEM has two monitors, both of which are active during operation. The graphics screen is the primary display device (displays menus, finite element models and all graphics information), and the PC's monitor is the secondary display (for showing status information, descriptions of file transactions, and a run-time display of the error norm during solution).

At the start of XPFEM, the user is prompted with a title page on the primary screen. To proceed, the user moves the mouse. This action causes the main menu of XPFEM to be displayed (Figure 22). The main menu contains fifteen options. Menu options are selected using the left or middle buttons on the mouse. The right mouse button is reserved for controlling mouse speed.

The main menu also contains title, specifier, and prompt/error display boxes. XPFEM automatically loads the latest model into memory (assuming there is a TITLES.DAT file containing the latest model). The title and specifier of the current problem appear in the appropriate boxes. Possible errors include NO GEOMETRY DEFINED and NO FACES ASSIGNED.

Clicking the right mouse button from within a program module will make visible a bar chart of the mouse rate. Moving the mouse up and down increases and decreases the rate, respectively. Note that a high mouse rate causes the mouse to move several screen pixels for each input and therefore prohibits the user from refined point selection required for vertices or edges of a model.

Erasing the workspace within a program module is accomplished using the CLEAR WRKSPCE option. Since segmentation is not available on the INMOS B409 board, erasing the workspace is completed by blanking the viewport coinciding with the workspace; therefore, to re-display selected model features, each feature must be re-toggled.

The fifteen options available from the main menu are described below.

## **XPFEM OPTIONS**

### **NEW PROBLEM**

Executing module NEW PROBLEM initializes XPFEM for a new problem. The user is prompted for the problem title and specifier; both identifiers must be legal file names without .\* extensions, and must contain fewer than ten characters. Data arrays for scaling, geometric modeling, and mesh generation are set to default values.

### **RECOVER OLD PROBLEM**

Module RECOVER OLD PROBLEM permits retrieval of a problem stored on disk. The user is prompted for the problem specifier. Existence of the requested problem is checked. If the problem exists, it is placed in core as the current problem; otherwise, a warning bell is sounded and the current problem is unaltered.

### **PROBLEM TYPE**

Module PROBLEM TYPE not implemented in XPFEM, version 89.1.

### **WORKSPACE GRID**

Options for defining the user workspace are provided in module WORKSPACE GRID. Options include grid size, grid scale, origin location, and grid line additions. Details for defining the workspace parameters are discussed on pages 4.2 - 4.4 of the *FINITE QUADTREE Mesh Generator User's Manual*.

### **FINITE QUADTREE GEOMETRIC MODELER**

Executing module FINITE QUADTREE GEOMETRIC MODELER provides a menu interface for geometric modeling using vertices, edges, and faces. Details for creating a geometric model are discussed in Chapters 3 and 4

(pages 3.1 - 4.12) of the *FINITE QUADTREE Mesh Generator User's Manual*. The user is required to declare model faces in this module (using ASSIGN FACE) prior to material assignment or mesh creation. Module options currently unavailable include SPLINE, ELLIPSE, AIRFOIL, CONCATENATE EDGE, CLOSED EDGE, WITH VERTEX, and RESET EDGE.

## **MATERIAL PROPERTIES**

Module MATERIAL PROPERTIES provides a menu interface for creating, editing, and assigning material properties. Details for using this feature of XPFEM are discussed on pages 4.12 - 4.15 of the *FINITE QUADTREE Mesh Generator User's Manual*. Currently options ORTHOTROPIC MATERIAL, ANISOTROPIC MATERIAL, SCROLL UP, SCROLL DOWN, LINEAR THICKNESS, SORT ON INDUSTRY, and SORT ON INTERNAL are not implemented.

## **ANALYSIS ATTRIBUTES**

Executing module ANALYSIS ATTRIBUTES provides the user with a menu interface for assigning boundary conditions. Permissible attributes include those associated with two-dimensional static analysis; therefore, menu options currently unavailable include INITIAL CONDITION, U' (velocity), and FACE.

The user initializes the attribute data arrays by selecting DEL ALL ATTRIBS. To assign a boundary condition, the user is required to select BOUNDARY CONDITION, specify the type (U for displacement and F for force), identify the geometric entity to which the condition is applied (VERTEX or EDGE), and input the boundary condition (1ST COMP ONLY, 2ND COMP ONLY, SCALAR, or VECTOR, and UNIFORM, LINEAR, or QUADRATIC). Two user prompts appear during the definition process - one for identifying the location of the geometric entity and one for inputting the magnitude of the boundary condition. To define a boundary condition with two components, the options 1ST COMP and 2ND COMP remain deselected. Following definition, the boundary condition is stored using STORE ATTRIB. Prior to selecting STORE ATTRIB, selected options may be altered without affecting the database.

Using CLEAR ATTRIB, all active options may be deselected. Boundary conditions can also be labeled using LABEL.

In summary, a boundary condition is assigned by activating one option in each of the listed groups:

1. BOUNDARY CONDITION

2. U (displacement)

F (force)

3. VERTEX (prompted to identify geometric entity)

FORCE

4. 1ST COMP ONLY (no selection indicates both components)

2ND COMP ONLY

5. SCALAR

VECTOR

6. X/Y

TANGENT/NORMAL

7. UNIFORM (prompted for coefficients)

LINEAR

QUADRATIC

8. STORE ATTRIB

As an example of applying attributes, a vertex may be constrained using the following sequence of selections:

```
DEL ALL ATTRIB      (for the first attribute only)
BOUNDARY CONDITION
U
VERTEX              (identify vertex when prompted)
VECTOR
XY
UNIFORM             (enter 0.0, 0.0 as two separate entries when
                    prompted)
STORE ATTRIB
```

## **FINITE QUADTREE MESH GENERATION**

A menu interface for meshing geometric models is contained in module **FINITE QUADTREE MESH GENERATION**. Options enable the user to specify mesh parameter sizes, element types, and refinement points. Detailed use of this module is discussed in Chapters 3 and 4 (pages 3.7 - 3.8, 4.16 - 4.25) of the *FINITE QUADTREE Mesh Generator User's Manual*.

## **OUTPUT FOR ANALYSIS**

Module **OUTPUT FOR ANALYSIS** is a menu interface for outputting data files. Options include viewing the model, mesh, attributes, and various parameters. Options for creating higher order elements, specifying faces of plane stress or plane strain, and displaying the renumbered nodes and elements also exist. Currently, the only output format available is XPFEM format. Selecting the option for XPFEM output causes four standard XPFEM input files for connectivity and constraints (\*.con), nodes (\*.xyz), forces (\*.frc.) and applied displacements (\*.apd) to be written to disk. These four files are used as input to the solver modules.

## READ NASTRAN FILE

Module READ NASTRAN FILE is the NASTRAN interface for XPFEM. Executing this module results in a prompt for the name of a NASTRAN file. The designated NASTRAN file is read by XPFEM, analyzed, and translated to XPFEM input format. Three standard XPFEM input files for connectivity and constraints, nodes, and applied forces are written to disk and used as inputs to the solver modules.

## 2D/3D SOLVER

Modules SOLVER 2D and SOLVER 3D compute a displacement solution for an XPFEM finite element problem. The solvers are based on the conjugate gradient method, an iterative solution for solving systems of linear equations with positive definitive coefficient matrices.

Following a selection of one of the solver modules, the user is prompted for a tolerance level. Iterations cease when all terms of the residual error vector have magnitudes lower than this tolerance. Typical values for the tolerance range from 1.0E-6 to 1.0E-9. As the solver runs, the secondary display shows the error norm at every tenth iteration.

The completed solution (nodal displacements) is written to a file on XPDOS in the directory of the user with filename `<problem name>.sol`. This file is used for post-processing in XPPOST.

The delivered prototype version of XPFEM can solve problems with 10,000 nodes in two dimensions (for 20,000 degrees of freedom) and 6,000 nodes in three dimensions (for 18,000 degrees of freedom). Other limitations govern the number of elements used, and the complexity of the elements. See Appendix B, Solver Limitations, for further details about solver capacity. This appendix also shows how the solver can be modified to handle over 35,000 degrees of freedom if simple elements are used.

## 2D/3D POST PROCESSOR

Executing modules 2D POST PROCESSOR/3D POST PROCESSOR permits finite element displacement solutions to be interactively post-processed using a menu interface. Separate menu interfaces are available for two-dimensional and three-dimensional models.

### 2D POST PROCESSOR

1. **DISPLAY MODEL:** This option allows the user to view the model. The user can choose to display the undeformed model, deformed model, or both by selecting UNDFRM, DFRM, or BOTH, respectively.

2. **DEFORM SCALE:** This option allows the user to input the deformation scaling factor. The default value is unity. The user should type in a real number and the model will be updated according to the new scale. Either DFRM or BOTH must be selected in the DISPLAY MODEL in order to observe the scaled deformed model.

3. **KEY IN:** This option allows the user to enter the transformation parameters from the keyboard. The following items are available:

**SCALE:** When this option is selected, the current x scale value will appear in the message box (top center of the screen), and the user will be prompted to input the new x scale (REAL). The program repeats the same procedure for y scale.

**TRNSLT:** When this option is selected, the current x translation value will appear in the message box and the user will be prompted to input the new x translation value (REAL). The program repeats the same procedure for y translation.

**ROTATE:** The current rotation angle value appears in the message box when this option is selected. The user is prompted to input the new rotation angle (REAL).

EXIT: Exit the KEY IN option.

4. **CHANGE DISPLAY:** This option allows the user to change the transformation parameters by 1 unit using mouse input. The user can select the following items:

**SCALE:** When this option is selected, a pop up window will appear with options for direction (X or Y), increment (+ or -), and EXIT.

**TRNSLT:** When this option is selected a pop up window will appear with options for direction (X or Y), increment (+ or -), and EXIT. The user can continue in this option until EXIT is hit.

**ROTATE:** When this option is selected a pop up window will appear with options for rotation angle (THETA), increment (+ or -), and EXIT.

**ZOOM IN:** This option allows the user to zoom in the view of the model.

**ZOOM OUT:** This option allows the user to zoom out the view of the model.

EXIT: Exit the CHANGE DISPLAY option.

5. **CONTOUR:** This option allows the user to view stress contours. The user may choose to display normal stresses in the x and y direction, and shear stresses by selecting SIG X, SIG Y, or SIG XY, respectively. The user also has option to input the number of contours by selecting NUM CONTOUR. Acceptable integer values range from 2 to 224 with 224 being default. Another option, SCALE, allows the user to enter the scale of the model from 0.5 to 0.75 with 0.5 being default. At the bottom of the work space, the color bar will be displayed along with the type of the contour, maximum stress value, and minimum stress value.

6. **PRINCIPAL STRESS:** This option allows the user to view the principal stresses. The user can choose to display compressive stresses, tensile stresses, both, or maximum shear stress using COMPRSS, TENSILE, BOTH,

and SHEAR MAX, respectively. Displays with or without the model are obtained selecting DISPLAY MODEL. At the bottom of the work space, the green line represents the length of the maximum principal stress, the magenta line represents the length of the maximum shear stress, the orange line indicates compressive stress, and the cyan line indicates tensile stress.

### 3D POST PROCESSING

1. DISPLAY MODEL: This option allows the user to view the model. The user can choose to display the undeformed model, deformed model, or both by selecting UNDFRM, DFRM or BOTH, respectively.

2. DEFORM SCALE: This option allows the user to input the deformation scale. The default value is unity. The user should type in a real number and the model will be updated according to the new scale. Either DFRM or BOTH must be selected in order to observe the scaled deformed model.

3. KEY IN: This option allows the user to enter the transformation parameters from the keyboard. The following items are available:

**SCALE:** When this option is selected, the current x scale value will appear in the message box, and the user will be prompted to input the new x scale (REAL). The program repeats the same procedures for y scale and z scale.

**TRNSLT:** When this option is selected, the current x translation value will appear in the message box and the user will be prompted to input the new x translation value (REAL). The program repeats the same procedures for y translation and z translation.

**ROTATE:** The current rotation angle value about the x-axis will appear in the message box when this option is selected. The user is prompted to input the new x rotation angle (REAL). The program repeats the same procedures for rotation angles about the y and z axes.

ZOOM: When this option is selected, the current viewpoint distance will appear in the message box. The user is requested to input the new viewpoint distance (REAL).

EXIT: Exit the KEY IN option.

4. CHANGE DISPLAY: This option allows the user to change the transformation parameters by 1 unit using mouse input. The user can select the following items:

SCALE: When this option is selected, a pop up window will appear with options for direction (X, Y, or Z) increment (+ or -), and EXIT. The user can continue in this option until EXIT is hit.

TRNSLT: When this option is selected, a pop up window will appear with options for direction (X, Y, or Z) increment (+ or -), and EXIT. The user can continue in this option until EXIT is hit.

ROTATE: When this option is selected, a pop up window will appear with options for rotation angles (THETA-X, THETA-Y, THETA-Z), increment (+ or -), and EXIT. The user can continue in this option until EXIT is hit.

ZOOM IN: This option allows the user to zoom in the view of the model. The user can continue in this option until EXIT is hit.

ZOOM OUT: This option allows the user to zoom out from the model. The user can continue in this option until EXIT is hit.

EXIT: Exit the change display option.

5. ORTHO PROJ: This option allows the user to toggle between an orthographic or central projection.

## EXIT

Executing module EXIT returns the user to the PC operating system.

## CONVERTING BINARY FILES TO TEXT FILES AND VICE VERSA

All disk files used by XPFEM are in binary format for fast input and output. Although most of these binary files are of no interest to the user, six of them contain data which the user may wish to examine or modify. These six files consist of four finite element input files (connectivity [`*.con`], node coordinates [`*.xyz`], applied forces [`*.frc`] and applied displacements [`*.apd`]) and two output files (node displacements [`*.sol`] and solution residual [`*.res`]). `EXE convert.tsr`, a binary/ASCII conversion program, enables the user to examine, in standart text format, the data contained in binary files, and conversely, to convert text input files into binary files cabable of being read by XPFEM.

### **Running `convert.tsr`**

The file conversion program is given as a standard occam EXE. It is accessible through the XPFEM.TOP. Load the program by hitting F5 while the cursor is on the ... `EXE convert.tsr` fold, and run it with F6.

### **Options**

`Convert.tsr` has three options. The first two perform bidirectional conversions on the four input files (with suffixes of `.con`, `.xyz`, `.frc` and `.apd`), and the third does a one-way conversion of the solution files (with suffixes `.sol` and `.res`) from binary to text. Note that solution files may not be converted from text format to binary format. The options appear as:

1. Convert input files from Binary to ASCII
2. Convert input files from ASCII to Binary
3. Convert solution files from Binary to ASCII
4. <quit>

After each option the program prompts for a user name, the source file name, the destination file name and the dimensionality of the problem.

## Binary Input Files to Text Input Files

For example, if the user wishes to convert binary input files for the 2D beam example to text, he would select option 1 and reply with the following prompts (bold-face letters represent user input):

```

Enter user name           DEMO
Enter binary filename     BEAM
Enter text filename       BEAM_TXT
Enter the Dimension (2/3, quit = 1) 2

```

The user name DEMO was selected because the example beam problem (and several others) was created with that user name. Note that only the root filename is given for both binary and text files; i.e., the suffixes .con, et cetera, are not added. After providing the four inputs above, the program will automatically convert all four (.con, .xyz, .frc and .apd) files using the filenames given:

binary file (old files)	text file (new files)
-----	-----
beam.con	beam_txt.con
beam.xyz	beam_txt.xyz
beam.sol	beam_txt.frc
beam.apd	beam_txt.apd

It is important to give a root destination filename that indicates what the file format is. For Option 1, the destination file will be in text format. For Option 2, the destination file will be in binary format. Typical conventions are to add \_TXT or \_TX for new text files and \_BIN or \_BN for new binary files.

The newly created text files may be examined with the 'type' command by entering XPDOS (Appendix D) and moving to the DEMO subdirectory. The text files can also be transferred to the host PC disk via the 'dosput' command where the files can be viewed or edited with any ASCII based wordprocessing program.

## Text Input Files to Binary Input Files

If the user manually creates input files for XPFEM, or edits an input file which was converted from binary to text, it is necessary to convert the text files to binary before XPFEM can use them. Any kind of user-created or modified input file must reside on the host PC disk (since XPDOS does not provide a means for users to edit files) and must therefore be moved to XPDOS so that the conversion program can reach them. See Appendix D, "Transferring Files between XPDOS and MS-DOS," on the 'dosget' command.

Once the text files are in the proper user subdirectory on XPDOS, the second option of the conversion program can be used to create binary input files. The inputs are similar to those for Option 1. The example below continues from the example in Section 1.3: assume the user first created text input files from the 2D beam problem, then moved the files to the PC hard disk where the beam\_txt.frc file was edited. After the text editing session, the user moved the four input files back to the DEMO subdirectory of XPDOS. The user then left XPDOS and ran the conversion program. The following inputs to the conversion program would produce valid XPFEM input files with rootname BEAM\_BIN:

```

Enter user name           DEMO
Enter text filename      BEAM_TXT
Enter binary filename    BEAM_BIN
Enter the Dimension (2/3, quit = 1) 2

```

This produces the following new files from the four input files:

text file (old files)	binary file (new files)
-----	-----
beam_txt.con	beam_bin.con
beam_txt.xyz	beam_bin.xyz
beam_txt.frc	beam_bin.sol
beam_txt.apd	beam_bin.apd

Note there are now two binary input files for the 2D beam example: the input files with rootname BEAM and the modified input files with rootname BEAM\_BIN. When running XPFEM, the user has the choice of running either the original beam problem by specifying BEAM as the problem name, or running the modified problem by specifying BEAM\_BIN as the problem name.

### Binary Solution Files to Text Solution Files

The third conversion option, converting binary solution files text files, permits inspection of the numerical finite element displacement solution and the error residual produced by that solution. In addition to the inputs for user name, problem name and dimensionality, this option also requires input for the number of nodes in the problem. If the user does not know how many nodes the problem has, and if the connectivity file (\*.CON) for that problem exists in the same subdirectory as the solution files, the user can let the conversion program find the number of nodes from the .CON file. If user opts for the program to search for the connectivity and the file does not exist, the program will crash.

The conversion program reads the solution (.sol) and residual files (.res) and writes them in text format to the file name specified by the user. The following sample session continues from the previous examples:

```
Enter user name          DEMO
Enter text filename      BEAM_BIN
Enter binary filename    BEAM_TXT
Enter the Dimension (2/3, quit = 1) 2
```

1. Enter number.of.nodes
2. Get number.of.nodes from .CON file

There are 72 nodes in BEAM\_BIN.CON.

This produces the following new files from the two solution files:

binary file (old files)	text file (new files)
-----	-----
beam_bin.sol	beam_txt.sol
beam_bin.res	beam_txt.res

The text solution files can be examined via the 'type' command of XPDOS, or can be transferred to the host PC disk with 'dosput.'

### **Problems with EXE convert.tsr**

The most frequently encountered problem with the conversion program is the 'File Not Found' error which, at this stage of development, causes the program to crash. Whenever the conversion program fails to work properly, enter XPDOS and verify that the expected input files are in the appropriate subdirectory.

If input files are modified in a manner that violates the file format, the text to binary conversion will crash. Known errors include:

1. Rather than exiting cleanly, a null string for a user or filename will cause the program to crash. Hitting 'Enter' to quit at a name prompt will exit as the program says, but will also cause a crash.
2. The code cannot recognize the difference between 2D and 3D problems, so if the dimensionality is entered incorrectly, the program will crash.

## APPENDIX B

### SOLVER LIMITATIONS

Due to the many variables which govern memory usage in the parallel finite element equation solver, defining a capacity limit is difficult. In general, the solver is limited to solving problems with up to 20,000 degrees of freedom (d.o.f.), both in two and three dimensions. It is possible, however, that one FE problem with 20,000 d.o.f. will run, but another equally large problem will not. It is also possible to make minor modifications to the solver libraries and raise the limit to 38,000 d.o.f. but this will incur several other limitations. This section describes the primary memory parameters, how they influence memory usage, and how the system can be reconfigured to solve the absolute largest problems possible.

In addition to restrictions on the maximum problem size, there is also a limit on the *minimum* problem size. This problem and a simple solution are explained at the end of this appendix.

#### Primary Memory Parameters

Eight parameters determine how much memory a given solver configuration requires. They are, in order of significance:

1. Problem Dimension (2 or 3)
2. Number of nodes
3. Number of elements
4. Maximum nodes per element
5. Maximum elements using any one node
6. Average nodes of influence
7. Average segments per row
8. Maximum fully constrained nodes

The number of degrees of freedom in a finite element model equals the product of the first two parameters, the *problem dimension* and the *number of nodes*. Since memory usage is a function of six other parameters, the dimension and number of nodes define only a subset of the critical variables.

The third most significant parameter, the *number of elements*, is closely coupled to the number of nodes and the model type. If only 3 noded triangular elements are expected, the number of elements must be at least twice as large as the number of nodes. If, on the otherhand, the FE model consists of 20 noded brick elements, the number of elements need only be one third of the number of nodes. The fourth parameter, the *maximum number of nodes per element*, follows directly from the same argument: lower order, two dimensional elements will have fewer nodes per element than higher order, three dimensional elements.

Unless the user manually creates the FE model, he or she may have little control over the fifth parameter, the *maximum number of elements sharing a common node*. A value large enough to include the worst case expected must be assigned. Typically, triangular elements rarely result in more than 10 elements sharing any one node. Three dimensional tetrahedral elements may have as many as 20 elements clustered about a single node.

*Nodes of influence* for a particular node are those remaining nodes contained in the elements which share that particular node. For example, imagine that six equilateral triangular elements fit together to form a hexagonal structure. The nodes of influence for the central node are all the remaining nodes in the structure, since all these nodes share an element with the central node. The central node therefore has six nodes of influence. The average number of nodes of influence for all nodes will be a function of the number of nodes in each element, and how the elements are joined. Long slender structures will result in a lower value for the average nodes of influence while a densely packed structure will have a higher value.

The last two parameters, *average segments per row* and *maximum fully constrained nodes* have minor effects on memory usage. Segments on a row refer to the number of consecutive non-zero terms on a given row of the global

stiffness matrix. Tightly banded problems typically have fewer than eight or ten segments per row, while poorly banded problems can have up to thirty. *Fully constrained nodes* are those which do not permit motion in any degree of freedom. Although most FE models have only a small portion of their nodes constrained in this manner, a relatively large value (e.g., equal to 50% of the total number of nodes) may be assigned to Maximum fully constrained nodes without excessive memory penalty.

### **Memory Usage as a Function of the Primary Memory Parameters**

The expression which defines the amount of memory used by the solver as a function of the eight memory parameters is complex. To assist the user in finding values yielding increased solver capacity, a Pascal program `memopt.pas` is given (in both source and executable form) which takes as input trial values for the eight variables and computes the number of bytes required by the remote transputers. Each of the 32 remote processors has 1 Mbyte of memory, so the number returned by `memopt.pas` should be less than 980,000 bytes. The remaining 24,000 bytes are required by the solver code itself and other minor working variables.

### **Reconfiguring the Solver for Larger Problems**

The current version of the solver is configured for 10,000 nodes, 10,000 elements, and a maximum of 8 nodes per element for 2 dimensions and 6,000 nodes, 5,000 elements and a maximum of 20 nodes per element for 3 dimensional problems. The remaining parameters are set for structures of average density and robustness of element connectivity. If the user expects to solve models with simple elements, the parameters may be altered to nearly double the solver's capacity.

Note that only an experienced occam programmer should attempt to reconfigure the solver. The following steps briefly describe the modification process. Full detail is intentionally omitted to encourage interested users to contact SPARTA before considering a reconfiguration.

1. The user must obtain a set of memory parameters which will bound the largest problems expected and still fit in memory. The `memopt.pas` program can be used to assist in the search.
2. The appropriate VAL's in the libraries `2Dvals.tsr` and `3Dvals.tsr` are changed to new values.
3. A global recompile is performed. "2Dvals.tsr" and "3Dvals.tsr" affect virtually every library and SC so that even code unrelated to the solver must be recompiled.
4. Check memory usage by remote processors to ensure that the memory required by the code fits on each transputer's 1 Mbyte of RAM.
5. Check memory usage by the host transputer to ensure that enough workspace is being passed to the solver host routine.

### **Minimum Problem Size**

A minor drawback to the prototype version of XPFEM is that it cannot solve problems with fewer than about 40 finite element nodes. Small problems can leave one or more processors without work to do, resulting in a communications hang. Simple models can always be solved by meshing with smaller elements to introduce additional nodes.

## APPENDIX C

### JACOBI-CONDITIONED CONJUGATE GRADIENT ALGORITHM

#### Notation:

{a}	vector
{a'}	partial vector
[a]	matrix
[a']	partial matrix

#### The Problem:

$$\text{solve } [A]\{x\} = \{b\}$$

#### Variables:

[A']	the horizontal slice of the stiffness matrix
{x'}	the displacement vector (divided up among processors)
{b}	the force vector (appears only in initialization and is not used explicitly)
[M.inv']	the preconditioning matrix (the inverse of the diagonal of [A'])
{p}	the projection vector (each processor gets a full copy)
{Ap'},	
{r'},	
{s'}	working vectors

#### Memory usage on each processor:

[max.degrees.of.freedom] REAL64 p: -- {p}  
 [max.degrees.of.freedom.per.processor] REAL32 x,r,s,Ap,M.inv:  
 {x'}, {r'}, {s'}, {Ap'}, [M.inv']  
 REAL64 pAp, rs, rs.new, a, c:

## Initialization:

1. Extract  $[M.inv']$  from  $[A']$ . Note that this is not straight forward, since  $[A']$  is stored in sparse form.
2. Assume initial guess of  $\{x\} = \{0\}$ , making  $\{r\} = \{b\}$ . Load the force vector directly into  $\{r\}$
3. Matrix-vector product:  $\{s'\} = [M.inv']\{r\}$  (actually a vector outer product since  $[M.inv']$  is stored as a vector)
4. Dot product:  $rs' = \{r'\}\{s'\}$
5. Assignment:  $\{p'\} = \{s'\}$
6. Vector global sum:  $\{p'\}$  becomes  $\{p\}$
7. Scalar global sum:  $rs'$  becomes  $rs$

## Iteration Loop:

8. Matrix-vector product:  $\{Ap'\} = [A']\{p\}$
9. Dot product:  $pAp' = \{p\}\{Ap'\}$  (use only  $\{p'\}$ )
10. Scalar global sum:  $pAp'$  becomes  $pAp$
11. Scalar division:  $a = rs/pAp$
12. Vector scale and add:  $\{x'\} = \{x'\} + a\{p\}$  (use only  $\{p'\}$ )
13. Vector scale and add:  $\{r'\} = \{r'\} - a\{Ap'\}$
14. Matrix-vector product:  $\{s'\} = [M.inv']\{r'\}$  (a vector outer product for Jacobi conditioning)
15. Dot product:  $rs.new' = \{r'\}\{s'\}$
16. Convergence test: is  $|a\{Ap'\}|$  small?
17. Scalar global sum:  $rs.new'$  becomes  $rs.new$ , also exchange results of convergence test
18. Scalar division:  $c = rs.new/rs$
19. Scalar assignment:  $rs = rs.new$
20. Vector scale and add:  $\{p'\} = \{s'\} + c\{p\}$  (use only  $\{p'\}$ )
21. Vector concatenation:  $\{p'\}$  becomes  $\{p\}$
22. Terminate on convergence test of Step 16, otherwise go to Step 8.

Communication occurs at steps 10, 17 and 21. The scalar sums at steps 10 and 17 are performed using the tetrodal configuration. The vector sum at

step 21 is a nearest-neighbor pipeline exchange of only the portions of  $\{p\}$  that processors need for the  $[A]\{p\}$  multiplication--this shadow of affectivity of  $\{p\}$  between processors equals the bandwidth of  $[A]$ .

## APPENDIX D

### XPDOS USER'S MANUAL

#### XPDOS COMMAND SHELL

The XPDOS shell allows the user to directly manipulate the directory and file structure on the transputer disks using commands similar to UNIX<sup>1</sup> and MS-DOS. To execute the XPDOS Shell from the transputer development system (TDS), first enter the XPFEM top (see Appendix A, XPFEM User's Manual) then use the arrow keys to position the cursor over the line which reads "...F comshell -- XPDOS Command Shell," press the **F5** key (Get Code), then the **F6** key (Run Code). The screen will clear and the XPDOS header will be displayed.

The XPDOS shell recognizes two disk drives, noted "a:" and "b:". The primary, or "root" directory of each disk is denoted by a lone slash ("/"). This directory can contain up to 255 files and subdirectories. Each subdirectory can also contain 255 files and subdirectories, and so forth ad in finitum. Files and subdirectories are assigned names up to 16 characters in length, which may consist of numbers, letters, dashes, underbars, and other miscellaneous symbols. The "." character may be included in a filename to provide compatibility with MS-DOS file extensions.

DRIVE	ROOT DIRECTORY	SUBDIRECTORY	FILES (+ SUBDIRECTORIES)
		materials ("a:/materials")	example.lib ("a:/materials/examples.lib") problem.lib ("a:/materials/problem.lib")
a:	("a:/")	John ("a:/john")	example.con ("a:/john/example.con") example.xyz ("a:/john/example.xyz")
		Rick	problem.con ("a:/rick/problem.con")

---

<sup>1</sup>UNIX is a trademark of AT&T



The `dir` (or `ls` for those who prefer Unix) will display a list of all files in a single directory. A directory pathname may be specified, or if it is omitted, the current directory will be displayed. The filesize in bytes is displayed next to each filename. A "\*" to the left indicates that this is a subdirectory rather than a file. Example:

<code>a:/&gt;dir rick</code>		<code>a:/&gt;dir</code>	
<u>Directory:</u> <code>a:/rick</code>		<u>Directory:</u> <code>a:/</code>	
<code>problem.con</code>	<code>32768</code>	<code>* materials</code>	<code>4096</code>
<code>problem.xyz</code>	<code>15000</code>	<code>* john</code>	<code>4096</code>
<code>* old problem</code>	<code>4096</code>	<code>* rick</code>	<code>4096</code>

Total of 51864 blocks in 3 files.

Total of 12288 blocks in 3 files.

## DELETING FILES

The `del` (or `rm`) command may be used to delete files. The file or files specified are permanently removed from the disk (there is no way to "undelete" a file). Examples:

```
a:/john> del example.*
a:/john> del b:/joe/*.*
```

## COPYING FILES

The `copy` (or `cp`) command will duplicate a file or files. The copy command looks like this:

"copy source files destination files"

Where the "source files" are the original files and "destination files" are the names of the new files.

Wildcards and full or partial path names may be used with the copy command.

Examples:

```
a:/> copy x.dat y.txt      (copies the file "x.dat" to "y.txt" in the current directory)
a:/> x:/joe/*.*          (copies all files in "b:/joe" to the current directory)
```

a:/> copy \*.txt \*.doc (copies all files in the current directory with the extension ".txt"  
into files w/extension ".doc")

## RENAMING AND MOVING FILES AND SUBDIRECTORIES

The `rename` (or `mv`) command will rename a file or subdirectory, and may even move it to a different directory. It is specified as "rename old name new name", and wildcards may be used for either old name or new name. Note that if a directory path is specified in new name which is different from that of old name, the file will be moved to the new directory as part of the renaming process. Directories themselves may also be renamed, and if they are moved, then all of the files inside of them move too. For example, if the command `mv a:/rick/old problem a:/` is executed, then the subdirectory `old problem` will be entered in the root directory alongside `rick`; thus, its full path name changes from `a:/rick/old problem` to `a:/old problem`, and its file `old.con` is now specified `a:/old problem/old.con`. Other examples:

a:/john> rename example.\* useful.\* (renames all "example: file  
to the name "useful",  
preserving the file extensions)

a:/john> rename \*/ (moves all files from the current  
to the root directory)

Note that the `rename` command may never be used to create circular path names; for instance, if `old problem` is a subdirectory of `a:/rick`, then the command `mv a:/rick a:/rick/old problem` will be rejected.

## CREATING DIRECTORIES

The `md` (or `mkdir`) command will create a new subdirectory, which is then capable of holding other subdirectories or files. If a full pathname is specified, then XPDOS will create a directory which has that full pathname; otherwise it will create a subdirectory of the current directory. Examples:

```

a:/john> md stuff          (creates "a:/john/stuff")
a:/john> md b:/joe/other  (creates "b:/joe/other")

```

## REMOVING DIRECTORIES

The `rd` (or `rmdir`) command will remove (delete) an existing subdirectory. Note that a subdirectory must be completely empty in order to be removed; if it has any files or subdirectories of its own, XPDOS will reject the `rd` command. The `del`, `rd`, and `rename` commands may be used to eliminate the files from a subdirectory prior to deletion. Examples:

```

a:/rick> del old problem*
a:/rick> rd old problem      Removes all files in "rick" and "rick/old problem," and then
a:/rick> del*                removes the directories themselves.
a:/rick> cd/
a:/> rd rick

```

## CHANGING DEFAULT DIRECTORY

The `cd` (or `chdir`) command will change to a different default directory and/or disk. the XPDOS prompt will change to match. If the full pathname is specified, that path will be used exactly; otherwise it will be computed based on the current default directory. Note also that the dot.dot ("`..`") specifier can be used to specify the current directory's "parent". Examples:

```

a:/> cd rick
a:/rick> cd old problem
a:/rick/old problem> cd ..
a:/rick> cd ..
a:/> cd b:/joe/stuff
b:/joe/stuff> cd a:/
a:/>

```

## CHANGING DEFAULT DISK

The default disk may be changed by simply typing the name of the disk "a:" or "b:") that you wish to change to. Examples:

```
a:/rick> b:
b:/> a:
a:/rick >
```

## DISPLAYING DISK FREE SPACE

The `free` command will display information about current disk space usage.

## DISPLAYING A FILE

The `type` command will display a file on the screen. During the display, the SPACE bar can be used to pause and restart the display. To abort the display, press any other key. Example:

```
a:/> type rick/problem.com
```

## OBTAINING HELP

The `help` command will list the XPDOS commands available.

## TRANSFERRING FILES BETWEEN XPDOS AND MS-DOS

The `dosget`, `dosput`, `dosgetbinary` and `dosputbinary` commands exist to transfer files between the transputer disks and the local PC/AT hard drive. The standard command (`dosget/dosput`) will convert between XPDOS and MS-DOS ASCII formats for text files; the binary versions (`dosget binary/dosput binary`) will duplicate binary and executable files precisely. All four commands work like the `copy` command, except that the source and destination files reside on different machines.

dosget:

The `dosget` commands load a file from MSDOS to XPDOS; the "binary" extension specifies exact duplication rather than ASCII conversion. The command is "`dosget <MS-DOS file> <XPDOS file>`."

`a:/> dosget c:/textfile.txt mytext.txt` (the MSDOS file "c:/textfile.txt" is loaded, converted, and placed on the transputer disk in the current directory as "mytext.txt")

`a:/> dosgetbinary file.bin a:/rick/problem.xxx` (the MS-DOS file "file.bin" from the current MS-DOS default drive and directory is loaded into a:/rick/problem.xxx with no ASCII conversion)

dosput:

The `dosput` commands write a file from XPDOS to MS-DOS; the "binary" extension specifies exact duplication rather than ASCII conversion. The command is `dosput <XP file> <MS file>`.

`a:/> dosput data.txt c:/data.txt`  
`a:/> dosputbinary problem.xxx file.bin`

**IMPORTANT:** No wildcards may be used for `dosget` and `dosput` commands. Also, the source and destination filenames must be completely specified (there is no "default filename"), although the default disk and directory are both recognized. The following commands are incorrect:

`a:/> dosput data.dat a:` (correct: `dosput data.dat a:data.dat`)  
`a:/> dosget textfile.txt` (correct: `dosget textfile.txt textfile.txt`)

EXECUTING MS-DOS COMMANDS

The `dos` command will allow the execution of MS-DOS commands from within the XPDOS shell. This is sometimes useful during file transfers (to obtain

directories and prepare files for transfer). When the `dos` command is entered, you will immediately be returned to MS-DOS. Type your MS-DOS commands, and when you are finished, type `exit` to return to XPDOS. Note that some larger programs will cause XPDOS to be overwritten in memory, in which case the `exit` command will cause the TDS to reboot w/the message "Transputer Error Flag Set" rather than returning to XPDOS. In this case, simply reload XPDOS as if starting from power up; no harm is caused by this procedure.

### EXITING FROM XPDOS

To exit the XPDOS shell, simply type the command `exit`; you will be returned to the TDS.

## APPENDIX E

### XPDOS DISK LIBRARY ROUTINES AND ERROR CODES

The following text describes the procedures available in XPDOS, the transputer disk operating system developed for the network disks in XPFEM. This disk operating system is a stand-alone, separately compiled occam library module which can be linked to by user-developed occam programs. Programs which call XPDOS must include the statement `#USE "disk.tsr"`, and be written in the `c:\fes12\disk` directory so that the linker finds the XPDOS library.

#### SYSTEM ROUTINES

##### Variables

`bioslink` host transputer link (0-3) which is connected to file server; this is declared as a VAL in `disks.tsr` and can thus be passed as a constant when using this library

`disknum` the disk to be initialized (0 or 1)

`status` variable to receive return status from routine.  
(1 = everything ok, negative = an error occurred)

##### Procedures

PROC BoostDisks (VAL INT bioslink)  
Boots the file server if necessary (has no effect if already booted)

PROC Initialize (VAL INT bioslink, disknum, INT status)  
Initializes the file structure on a new disk drive.

*WARNING!!! This routine will erase ALL data on the disk, and thus should not be called except the very first time a disk is used. If it is desired to actually initialize the disk drive, the recommended procedure*

*is to use the "initialize" command from the XPDOS Shell, rather than invoking this procedure from a program.*

## DIRECTORY ACCESS ROUTINES

The following routines are used to manipulate the file and directory structure of an XPDOS dis. Each routine corresponds to an XPDOS Shell command of the same name, and is in fact called by the Shell in order to perform its function. These routines may be included in user programs in order to perform directory and file operations.

```
PROC MakeDirectory (VAL INT bioslink, VAL []BYTE pathname,
                   INT status)
```

Attempts to create a new directory (called by `md/mkdir` command).

pathname = the full or partial path of the directory to be created.

No wildcards are permitted for this command.

```
PROC RemoveDirectory (VAL INT bioslink, VAL [] BYTE pathname,
                     INT status)
```

Attempts to remove a directory (Called by `rd/rmdir` command).

pathname = full or partial path of the directory to be removed.

No wildcards are permitted for this command.

```
PROC ListDirectory (VAL INT bioslink, VAL []BYTE pathname,
                   []BYTE dirpath, INT dirsize,
                   [dirmax*dlistlen]BYTE dir.buffer, INT status)
```

Lists the directory specified by pathname into a buffer which can then be examined by the user routine. Variables are as follows:

pathname = full or partial path of directory or files to be listed.

Wildcards of any kind are allowed here (\* and ?).

dirpath = BYTE buffer to receive the full pathname of the directory being listed. The maximum length is 80 characters.

dirsize = INT variable to receive the number of entries in this directory. The maximum number of entries is 255.

dir.buffer = BYTE buffer to receive the directory entries. This buffer must be of exactly length `dirmax*dlistlen`, each of which is

declared as a constant in this library. The entries are packed into the buffer and can be read as follows:

```
[dirmax*dlistlen]BYTE dir.buffer      :
[dlistlen]BYTE list.entry             :
[filenamemax]BYTE list.filename RETYPES [
    list.entry FROM 7 FOR      filenamemax] :
INT list.size RETYPES [list.entry FROM 0 FOR 4] :
INT list.dirflag RETYPES [list.entry FROM 4 FOR 4] :
SEQ index = 0 FOR dirsize
SEQ
    list.entry := [dir.buffer FROM (index*dlistlen) FOR
                  dlistlen]:
    ... Examine directory entry (list.filename, list.size,
                                list.dirflag)
```

```
list.filename = the name of the file
list.size     = the size in bytes of the file
list.dirflag  = flag: 1=subdirectory, 0=file
```

```
PROC ChangeDirectory (VAL INT bioslink, VAL [] BYTE
    pathname, INT disknum, [] BYTE returnpath, INT status)
Changes the current working directory (called by cd/chdir). After the
working directory is changed, the new disk and path is returned.
pathname = BYTE Array containing path to be change to. If pathname
is empty, ChangeDirectory simply returns the current path.
disknum  = INT variable to receive the disk number (0,1) of the new
default path.
returnpath = BYTE array to receive the new default path.
```

```
PROC MoveFile (VAL INT bioslink, VAL [] BYTE sourcepath,
    destpath, INT status)
Renames files & subdirectories and/or moves them from one directory to
another. (Called by mv/rename)
sourcepath = pathname existing files to be moved or renamed.
Wildcards of all kinds are permitted.
```

destpath = new pathname for files. Wildcards of all kinds are permitted.

PROC CopyFile (VAL INT bioslink, VAL []BYTE sourcepath,  
destpath, INT status)

Copies files from one place to another. (called by cp/copy)

sourcepath = pathname of existing file(s). Wildcards are permitted.

destpath = pathname to copy to. Wildcards are permitted.

PROC DeleteFile (VAL INT bioslink, VAL []BYPE pathname, INT  
status)

Deletes the file or files specified (Called by rm/del)

pathname = the name of the file(s) to be deleted. Wildcards are permitted here.

PROC FreeSpace (VAL INT bioslink, disknum, INT freetotal,  
free.avail, free.reserved, free.used, status)

Returns free space information for the disk specified. All information is provided in terms of 4096 byte blocks.

disknum = the disk # (0 or 1) of the disk to be examined.

free.total = the total number of blocks existing on the disk

free.avail = the number of blocks not currently in use (i.e. free blocks)

free.reserved = the number of blocks reserved for use by the file system

free.used = the number of blocks already allocated to files

PROC ErrorMessage (VAL INT status)

Displays the error message that corresponds to a particular status code.

If a file routine returns a negative status, that status can be passed to this routine to display the appropriate error message on the screen. Refer to the end of this document for a list of error messages.

## FILE OPENING

### Variables

bioslink	host transputer link (0-3) connected to fileserv
unit	variable to receive the unit number assigned to the file. This variable must then be passed as unit to all subsequent calls involving the file. XPDOS currently allows up to 4 files to be open at once.
filename	text (BYTE) string containing the name of the file to be opened, terminated either by a null (zero) byte or the physical end of the byte array.
status	variable to receive the return status of the open routine. (1 = everything went OK, negative = error code)
reclen	(direct access only). INT value specifying the number of bytes in a fixed length record.

### Procedures

```
PROC OpenFileBinaryRead (VAL INT bioslink, INT unit,
                        VAL []BYTE filename, INT status)
  Opens a file for readonly access as a stream of bytes (binary/ASCII)
```

```
PROC OpenFileBinaryWrite (VAL INT bioslin, INT unit,
                          VAL [] BYTE filename, INT status)
  Creates a new file (overwrites any previously existing version) for write
  only access as a stream of bytes (binary/ASCII)
```

```
PROC OpenFileBinaryAppend (VAL INT bioslink, INT unit,
                           VAL []BYTE filename, INT status)
  Like OpenFileBinaryAppend, except that if the file already exists, it will
  be appended to rather than overwritten.
```

PROC OpenFileDirectReadWrite (VAL INT bioslink, INT unit,  
 VAL []BYTE filename, VAL INT reclen, INT status)  
 Opens a file (creates it if it does not exist) for random read/write access  
 with a fixed record length of reclen

PROC OpenFileDirectReadOnly (VAL INT bioslink, INT unit,  
 VAL []BYTE filename, VAL INT reclen, INT status)  
 Like OpenFileDirectReadWrite, except that records may only be read;  
 never written.

## GENERAL ROUTINES

The following routines can be used for all types of files -- both binary and direct access.

PROC FlushBuffer (VAL INT bioslink, unit, INT status)  
 Flushes the file's i/o buffer to disk; any data currently being held in memory is now written out. This routine can be used to make sure that information from previous Put/FileWrite calls has actually been output to the disk drive. This routine is called automatically by CloseFile.

PROC EndOfFile (VAL INT bioslink, unit, BOOL eof)  
 Checks to see if the end of file marker has been reached for this file. If the end of the file has been reached, eof will return TRUE; otherwise, it will return FALSE.

PROC CurrentPosition (VAL INT bioslink, unit, INT curpos)  
 Returns the current position of a readonly or direct access file. For direct access files, curpos returns the next record number; for binary files, curpos returns the byte offset from the beginning of the file. The first byte/record in a file is #0.

PROC CloseFile (VAL INT bioslink, unit, INT status)  
 Closes a file, flushes its i/o buffer, and frees its unit number for use by other files. Once CloseFile is called, no further i/o calls should be

addressed to that unit # unless a new `OpenFile` call is issued.  
`CloseFile` terminates all file activity for that file.

## DIRECT ACCESS

### Variables Common to, and Visible from, all Direct Access Routines

`record`            array of length `reclen`, containing the fixed-length record to be written or to receive the one about to be read.

`recnum`            the logical record number to be read or written. The first record in a file is record 0. If a record is written beyond the current end of file marker, the file is extended so as to contain that record. If a record is read beyond the current end of file marker, an error is returned.

### Procedures

```
PROC GetRec (VAL INT bioslink, unit, [] BYTE record,
            VAL INT recnum, INT status)
    Reads one random access record
```

```
PROC PutRec (VAL INT bioslink, unit, VAL []BYTE record,
            VAL INT recnum, INT status)
    Writes one random access record
```

## BINARY ACCESS

```
PROC Get Byte (VAL INT bioslink, unit, BYTE item, status)
PROC PutByte (VAL INT bioslink, unit, VAL BYTE item, status)
    Routines to read or write a single byte to the binary file.item - the byte to be read or written.
```

```
PROC Rewind (VAL INT bioslink, unit INT status)
    Rewinds a read-only file. The next byte read will be the very first byte in the file. This has the same effect as closing the file and reopening it.
```

PROC Position (VAL INT bioslink, unit, filepos, INT status)

Positions a read-only file to the desired location

filepos = the byte # in the file to position to. The very first byte in a file is #0, so a position to byte 0 produces the same effect as the Rewind routine.

PROC BackSpace (VAL INT bioslink, unit, INT status)

Positions a readonly file to one less than the current position. This essentially has the effect of "unreading" the last byte read; thus, the next byte read will be the same as the last type read. Multiple BackSpaces can be used to produce multiple "unreads" ad infinitum. BackSpacing past the beginning of a file produces an error code.

PROC FileWrite (VAL INT bioslink, unit, VAL [] BYTE line,  
INT status)

PROC FileWriteln (VAL INT bioslink, unit VAL [] BYTE line,  
INT status)

Procedures to write a string of bytes; has the same effect as a series of calls to PutByte. FileWrite simply writes the string passed to it, while FileWriteln will append a carriage return (ASCII 13).

line = BYTE array to be written. Every byte in line will be stored to the file; the number of bytes written is thus equal to SIZE line.

PROC FileWriteINT (VAL INT bioslink, unit, number, fieldsize,  
INT status)

Writes an integer value in ASCII format into the file.

number = INT number to be written

fieldsize = # of ASCII bytes to output. If fieldsize is greater than or equal to the number of bytes necessary for numeric output, spaces will be added to pad out the field. If fieldsize is less than or equal to the number of bytes needed, or if fieldsize is 0, then the number will be output followed by a space.

```

PROC FileWriteREAL32 (VAL INT bioslink, unit,
                    VAL REAL32 number, VAL INT leftfield,
                    rightfield, INT status)

```

```

PROC FileWriteREAL 64 (VAL INT bioslink, unit,
                    VAL REAL64 number, VAL INT leftfield, rightfield,
                    INT status)

```

Procedures to write a single or double precision real in ASCII format to the disk. The string written is guaranteed to begin and end with a blank space character and to have no internal blank spaces.

number = the REAL32 or REAL64 number to be written

leftfield = field size to the left of the decimal point

rightfield = field size to the right of the decimal point (as noted above, the string will begin and end with a space)

```

PROC FileRead (VAL INT bioslink, unit, []BYTE line,
              VAL INT lentogot, INT lenwegot, status)

```

```

PROC FileReadBlock (VAL INT bioslink, unit, []BYTE line, VAL
                  INT lentogot, INT lenwegot, status)

```

Procedures to read an array of bytes from the disk. Much like a series of calls to GetByte. FileRead stops short if an end-of-line character (carriage return) is encountered; FileReadBlock reads carriage returns as if they were regular bytes.

line = BYTE array to receive the bytes read. Must be at least as long as lentogot.

lentogot = number of bytes to read

lenwegot = INT variable to receive the number of bytes actually read. Normally will be equal to lenwegot, but may be less if an end of line or end of file is encountered. In any event, this contains the number of bytes of line which are significant.

```

PROC FileReadNewLine (VAL INT bioslink, unit INT status)

```

Can be used to position the file to the beginning of the next line of text (i.e. after the next end-of-line marker)

```
PROC FileReadIn (VAL INT bioslink, unit, []BYTE line, INT
                status)
```

Much more useful than FileRead, this routine reads a series of bytes which is terminated when an end of line marker (carriage return) is encountered. On return, line will contain the bytes read followed by null bytes. The INT function StrLength has been provided in this library to determine the significant length of the returned string. Example:

```
FileReadIn (bioslin, unit, readme, status)
len := StrLength (readme)
write("String: ")
writeln ([readme FROM 0 FOR len])
```

```
PROC FileReadINT (VAL INT bioslink, unit, INT number, status)
```

Attempts to read an integer number (stored in ASCII format) from the file. Any spaces and end-of-lines between the current position and the number will be ignored, but any non-numeric or REAL data will produce an error status. This routine will leave the file positioned at the first byte following the integer data read.

```
PROC FileReadREAL32 (VAL INT bioslink, unit, REAL32 number,
                    INT status)
```

```
PROC FileReadREAL64 (VAL INT bioslink, unit, REAL64 number,
                    INT status)
```

Attempts to read a single or double precision real (stored in ASCII format). Any spaces and end-of-lines between the current position and the number will be ignored, but any non-numeric or INT data will produce an error status. These routines will leave the file positioned at the first byte following the real data read.

### **Packed Formats**

The following routines (also compatible with binary files) implement \*packed\* numeric output formats, as opposed to the ASCII formats of the previous routines. Whereas ASCII formats simply write out numbers in text format (e.g. "1.08E+07"), and then convert them back character by character during reads, packed formats write the information exactly as the program

stores it in memory. These formats are much faster and much more compact than ASCII formats, but are not compatible with ASCII read and write routines (they also produce files that are unreadable by human eyes, unlike their ASCII counterparts). Because packed formats ignore end-of-line markers, such formats should not be used in conjunction with any of the `FileRead` or `FileWrite` routines, except for `FileReadBlock` and `FileWrite`, which can be safely used to read and write fixed length strings.

```
PROC PutINT      (VAL INT bioslink, unit, number, INT status)
```

```
PROC PutREAL32  (VAL INT bioslink, unit, VAL INT REAL32
                 number, INT status)
```

```
PROC PutREAL64  (VAL INT bioslink, unit, VAL INT REAL64
                 number, INT status)
```

Writes the specified value in packed format to the file. INT and REAL32 values require 4 bytes, while REAL64 values require 8 bytes.

```
PROC GetINT      (VAL INT bioslink, unit, INT number, status)
```

```
PROC GetREAL32  (VAL INT bioslink, unit, REAL32 number,
                 INT status)
```

```
PROC GetREAL64  (VAL INT bioslink, unit, REAL64 number,
                 INT status)
```

Reads the specified type number in packed format from the file. Note that the file must be positioned to the exact spot at which a number in the specified format was previously written, or an incorrect value will be returned. Note also that though INT and REAL32's each require 4 bytes, the formats are NOT compatible, and cannot be used interchangeably. A number written as REAL32 must be read as REAL32.

## STATUS CODES

### Positive Values

1 = Operation successful.

### Negative Values

-1 = Resource not available.

The resource which you have requested (probably disk space) has been exhausted. Check disk free space and delete some files. If disk already shows free space, then the inode table has been exhausted; again, the solution is to delete some files.

-2 = File not found.

The file which you have attempted to access does not exist, and the routine which you have called cannot proceed without an already existing file. Check your filename and path to make sure they are correct. You may be trying to access a file which resides in another subdirectory, in which case you must either change directory to that subdirectory, or use the full pathname (from root) of the file.

-3 = Invalid path.

You have specified an invalid path. Note that in a long path such as `/usr/brian/stuff`, every entry except the last one **MUST** exist already ("`MakeDirectory`" cannot be used to create a whole tree of subdirectories with a single command). Only the last entry (i.e. `stuff`) can be non-existent, depending on the nature of the command. Check to make sure that your pathname has been spelled correctly, and that each entry corresponds to an existing subdirectory.

-4 = Directory full.

You have attempted to create a file or subdirectory in a directory which is already full. Each directory can contain no more than 255 entries, so the only solution at this point is to delete some files or move them to another directory. The best advice is to divide the current directory into a series of subdirectories anyway.

-5 = Not a directory.

You have attempted to perform a directory operation (make, remove, change) on a file. These functions can ONLY be used to modify directory structures.

-6 = Pathname overflow.

XPDOS has attempted to return a pathname which is longer than the maximum pathname allowed (80 chars). The solution is to keep subdirectory trees to a reasonable depth and not go so hogwild that you end up with 85 character pathnames.

-7 = File allocation overflow.

A file has attempted to expand beyond the maximum number of blocks which XPDOS is able to allocate to a single file (slightly more than 8 Megabytes in the current implementation). No more information can be added to this file, although a second file could be opened and information added to it where the first left off.

-8 = Cannot create: already exists.

You have attempted to create a directory with a pathname that is already in use (i.e. the directory already exists or a file exists with the same name you had in mind for the directory). Rename the existing occupant of the name, or create your directory using a different name.

-9 = No such file(s).

A wildcard file operation has failed to match any files. In other words, no files matching your wildcard were found, and no operations were performed.

-10 = Cannot remove: directory not empty.

You have attempted to remove a subdirectory which still contains entries (subdirectories or files of any kind, empty or not). Only directories which are completely empty (i.e. no entries except for "." and "..") can be removed.

-11 = Cannot remove current directory.

You have attempted to remove the directory which you are logged to (i.e. your current default directory). Use `ChangeDirectory` to move to the parent directory and try the operation again.

-12 = Cannot remove root directory.

You have attempted to remove the root ("/") directory. This directory is a permanent part of the file structure and cannot be removed at any time for any reason (after all, why would you want to?).

-13 = Cannot OPEN a directory file.

You have attempted to open a subdirectory for file access. The only way to modify a directory "file" is through the directory manipulation commands (`MakeDirectory`, `RemoveDirectory`, `MoveFile`, `ListDirectory`). File operations like `OpenFile` and `CloseFile` are never allowed.

-14 = File is already open.

You have attempted to open a file which is already open. To reopen a file using a different access method you must close it first.

-15 = Record length does not match record size.

You have attempted to perform a fixed length record operation (`GetRec` or `PutRec`) using a record whose `SIZE` does not match the record length (`reclen`) specified at the time you executed the `OpenFile` command. Alter your array dimensions so that they conform to the record size, or pass an appropriately sized substring of a larger array.

-16 = Read past end of file.

Your read operation has requested more information than remains in the file, and is thus unable to proceed. No more information can be read from this unit # unless the file marker is repositioned using `Rewind`, `Position`, `Backspace`, or `GetRec/PutRec`, or until the file is `Closed` and the `reOpened`.

-17 = Block not in file.

Your `GetRec` command has attempted to access a record which has a higher id number than any which has previously been written to the file (in other words, you have attempted to read a record which does not yet exist). A record should not be read from until it has been written to at least once.

-18 = Cannot read from writeonly file.

You have attempted to perform a read operation on a file which has been opened for write-only access. To read from this file, you must close it and then reopen it for read access.

-19 = Cannot write to readonly file.

You have attempted to write information to a file which has been opened for readonly access. To write to the file, you must close it and then reopen it in an access mode which permits writing.

-20 = File is not open.

You have attempted to perform a file operation (Read, Write, Get, Put) on a unit # which has not yet been opened. You must use an `OpenFile` command to open a file and obtain a unit # before you can perform i/o operations on it.

-21 - Not a binary file.

You have attempted to perform a binary i/o operation (`FileRead`, `FileWrite`, `GetByte`, `PutByte`, etc.) on a file which is opened for direct access.

-22 = Not a direct access file.

You have attempted to perform a direct i/o operation (`GetRec`, `Put Rec`) on a file which is opened for binary access.

-23 = Buffer too small.

A `FileRead` operation has retrieved more information than will fit into the buffer which you have provided. You should alter your array dimensions to create a larger buffer, or check the file to be sure that it was really intended to be read in this manner (for example, a `FileReadIn` command performed on a file which contains no carriage returns, will simply try to return the whole files as one line--probably not what you had intended). Possibly a different form of access (such as `GetByte`) is called for.

-24 = Not a file.

You have attempted to perform a file operation on something which is not a file.

-25 = Target file already exists.

A `MoveFile` operation has attempted to change a file's name such that it would be identical with the name of another already existing file. Since XPDOS does not allow identical filenames in the same directory, the operation has failed. Alter your `MoveFile` operation so that it will give each file a unique name.

-26 = Cannot move root directory.

You have attempted to move or rename the root directory. The root directory cannot be moved, removed, renamed, replaced, deleted, folded, spindled, mangled, or mutilated in any way. It is one of those constants in life which stands apart from time and space as we know it.

-27 = Missing parameter.

You have attempted to execute an XPDOS shell command without supplying one or more required parameters (`mkdir`, for instance, requires that you supply the name of the directory to be created). Check your command's syntax and supply the missing parameter.

-28 = Cannot copy/move file to itself.

You have attempted to copy or move a file to the exact same location and filename which it already has. Such an operation is not possible, and would do nothing useful anyway. Change your copy or move command so that it puts the file somewhere else or with a different name.

-29 = Cannot move parent to child.

You have attempted to move a directory into its own subdirectory (or some subdirectory of which it is the parent, grandparent, etc.). The directory structure must always be a TREE; never a circle, tube, cylinder, or other aberration. No directory operation which would result in the creation of a circular subdirectory structure, or in the severance of part of the structure from the rest of the tree is allowed.

-30 = Cannot move logged directory structure.

You have requested a `MoveFile` operation which would result in your current default directory being moved. As this is not allowed, you must first use `ChangeDirectory` to move to a different directory (such as the root) and then try the operation again.

-31 = Cannot move between disks.

You have attempted to move or rename a file so that it would end up on a different disk from where it started. Only the `CopyFile` command can transfer information between disks, so it is suggested that you use that.

-32 = Illegal unit number.

You have attempted to perform an operation using an illegal unit number. You should use an `OpenFile` command to open a file and get a legal unit number before you attempt to perform any i/o operations.

-33 = No unit #'s available.

You have attempted to open a file when no unit #'s are available on which to open it. This means that 4 files are already opened, and you must close one of them before you can open any more.

-34 = Device does not exist.

You have specified a disk number which does not correspond to a disk existing in your system. Check to see how many disks you have and try again.

-35 = Backspace past top of file.

You have called the `BackSpace` procedure when no information has yet been read from the file (or all information has already been "unread"). Backspace cannot be used to move beyond the top of a file.

-36 = Bad data on integer read.

A `FileReadINT` operation has encountered non-blank, non-numeric data before reaching any useful integer information. Check your data file to be sure that it has been written in the proper format.

-37 = Integer conversion failure.

A `FileReadINT` operation has encountered numeric data which it is unable to convert, probably because the number itself is too large to fit into a 32 bit integer variable. Check your file to make sure that the data is not all crammed together with no spaces in the middle or something like that.

-38 = Conversion buffer overflow.

A `FileReadINT` or `FileReadREAL` operation has encountered numeric data so lengthy that it will not fit into the internal conversion buffer. This probably means either that your data has been run together without spaces, is so large that it wouldn't fit into a variable anyway, or has more significant figures than even a `REAL64` could represent.

-39 = Found REAL data on an INT read.

A `FileReadINT` operation has encountered numeric data with a decimal point (".") in the middle of it. This probably means that you are reading your file in the wrong order, or wrote `REAL` data where `INT` data belonged. The number was read anyway (& rounded off to `INT`), but any further results from this file are dubious.

-40 = Found INT data on a REAL read.

A `FileReadREAL` operation has encountered numeric data with no decimal point ("."). This probably means that you are reading your file in the wrong order, or wrote INT data where REAL data belonged. The number was read anyway (& converted to REAL), but any further results from this file are dubious.

-41 = Error opening host file.

Your `dosget` or `dosput` MS-DOS transfer operation has failed to open the file on the host system. Either the file does not exist in the case of a `dosget`, or there is something wrong with the pathname. Examine your host directory structure to see what the problem is.

-42 = Failed to execute host command.

XPDOS has attempted to execute a command on the host system and has for some reason failed.

-43 = Packed read failed.

Your packed read operation (`GetINT`, `GetREAL32`, `GetREAL64`) has failed because there are not enough bytes left in the file to hold the specified variable. INT & REAL32 each occupy 4 bytes, while REAL64 occupies 8. This error probably means that something is horribly wrong with your file format, or that you are reading it back in the wrong order.

-44 = No wildcards allowed for this command.

You have specified wildcards ("\*" or "?") for a command which does not allow them. Commands which allow wildcards are `ListDirectory`, `MoveFile`, `CopyFile`, and `DeleteFile`. Commands which never allow wildcards are `MakeDirectory`, `RemoveDirectory`, `ChangeDirectory`, and `OpenFile`.

## APPENDIX F

### XPGRAPHICS USER'S MANUAL

The following text describes XPGRAPHICS, a library of graphics primitives. XPGRAPHICS was developed as software support for the INMOS B408/B409 graphics hardware system included in XPFEM. The library is a stand-alone module which can be linked to applications by including #USE "graphics.tsr" in the occam code.

One variable, *link*, occurs in each XPGRAPHICS routine. Its definition for all routines is given below:

*link* = integer variable which defines the graphics channel number between the TDS and the graphics disk server (B408).  
 $0 \leq \textit{link} \leq 3$

#### SYSTEM ROUTINES

PROC InitServer.XL (VAL INT link)  
Initializes the graphics server.

PROC Terminate.XL (VAL INT link)  
Terminates the graphics server.

#### COLOR ROUTINES

PROC SetBkColor.XL (VAL INT link, color)  
Sets the background color.  
color = background color value. Default is 0 (BLACK).  
 $0 \leq \textit{color} \leq 255$

PROC Set FgColor.XL (VAL INT link, color)

Sets the foreground color.

Input: color = foreground color value. Default is 15 (WHITE).

0 <= color <= 255

PROC SetColor.XL (VAL INT link, color.value, red, green,  
blue)

Defines a new color value in the color look up table.

Input: color.value = new color value identification number.

0 <= color.value <= 255

red = value of red color.

0 <= red <= 63

green = value of green color.

0 <= green <= 63

blue = value of blue color.

0 <= blue <= 63

PROC SelectColorTable.XL (VAL INT link, table.number)

Selects a color table for drawing.

Input: table.number = number of desired color table. Default = 1.

0 <= table.number <= 2

PROC GetPixelColor.XL (VAL INT link, x, y, INT color, error)

Returns the color value of an assigned pixel.

Input: x, y = coordinates of desired pixel.

(Monitor and frequency dependent.)

Output: color = color value assigned to pixel at (x,y)

error = error code.

PROC GetBkColor.XL (VAL INT link, INT color, error)

Returns the background color number.

Output: color = current background color value.

error = error code.

PROC GetFgColor.XL (VAL INT link, INT color, error)

Returns the foreground color number.

Output: color = current foreground color value.  
error = error code.

PROC SetMaskReg.XL (VAL INT link, mask)

Sets the mask register on the IMS G170.

Input: mask = pixel mask register. The pixel mask register in the IMS G170 is bitwise ANDed with the address register to give a masked address. This masking can be used to alter displayed colors without altering the video memory or the look-up table contents.

### Point, Line, Polygon and Curve Routines

PROC SetDash.XL (VAL INT link, VAL INT16 dash.pattern)

Sets dash pattern.

Input: dash.pattern = desired dash pattern number.  
Default is 0 (\_\_\_).

1 = .....

2 = \_.\_.\_

3 = ----

4 = \_\_\_\_

5 = ..\_.\_

6 = \_\_\_\_\_

7 = \_.\_.\_.\_

8 = \_\_\_\_\_

9 = . . . . .

PROC Line.XL (VAL INT link, x0, y0, x1, y1)

Draws a line given two endpoints.

Input: x0, y0 = coordinates of the beginning point.  
x1, y1 = coordinates of the endpoint.

PROC LineRel.XL (VAL INT link, del.x, del.y)

**Draws a line relative to the current cursor position.**

**Input:** del.x, del.y = drawing increments in the x and y directions.

PROC LineTo.XL (VAL INT link, x2, y2)

**Draws a line from the current cursor position to an specified endpoint.**

**Input:** x2, y2 = coordinates of the endpoint.

PROC LineFrom.XL (VAL INT lin, x2, y2)

**Draws a line from the current cursor position to a specified endpoint without changing the current cursor position.**

**Input:** x2, y2 = coordinates of the endpoint.

PROC Point.XL (VAL INT link, x, y)

**Plots a point at a specific position.**

**Input:** x, y = coordinates of the desired position.

PROC Circle.XL (VAL INT link, xc, yc, rad)

**Draws a circle determined by the center and radius.**

**Inputs:** xc, yc = coordinates of the center of the circle.  
rad = radius of the circle.

PROC Arc.XL (VAL INT link, x1, y1, x2, y2, x3, y3)

**Draws an arc of a circle between the three points specified.**

**Input:** x1, y1 = coordinates of the beginning point of the arc.

x2, y2 = coordinates of a point which lies on the arc.

x3, y3 = coordinates of the endpoint of the arc.

PROC Rect.XL (VAL INT link, x1, y1, x2, y2)

**Draws a rectangle using the top left corner and bottom right corner.**

**Input:** x1, y1 = coordinates of the top left corner of the rectangle.

x2, y2 = coordinates of the bottom right corner of the rectangle.

```
PROC Polygon.XL (VAL INT link, number.sides,
                VAL [] [] INT polygon)
```

**Draws a polygon given a specified number of sides and the vertices.**

**Input:** number.sides = number of sides of the polygon.

          [] [] polygon = coordinates of the vertices of the polygon.

```
PROC PolygonFill.XL (VAL INT link, number.sides,
                   VAL [] [] INT polygon, VAL INT color )
```

**Fills a polygon with a specified color. The polygon is specified by number of sides and vertices.**

**Input:** number.sides = number of sides of polygon.

          [] [] polygon = coordinates of the vertices of the polygon.

          color = define desired fill color.

```
PROC FillPolygon.XL (VAL INT link, x, y, color)
```

**Fills a polygon with a specified color. The polygon is specified by an interior point.**

**Input:** x,y = coordinates of a point inside the polygon.

          color = fill color.

```
PROC QuickFill.XL (VAL INT link, x, y, color)
```

**Fills a polygon that has simple convex shape. The polygon is specified by an interior point.**

**Input:** x,y = coordinates of a point inside the polygon.

          color = fill color.

```
PROC MoveDraw.XL (VAL INT link, x, y)
```

**Moves the cursor to a specified position.**

**Input:** x,y = the coordinates of the desired cursor position.

```
PROC MoveDrawRel.XL (VAL INT link, del.x, del.y)
```

**Moves the cursor position relative to its current position.**

**Input:** del.x, del.y = incremental distances from current position.

PROC SetDrawMode.XL (VAL INT link, mode)

**Sets drawing mode.**

Input: mode = define desired drawing mode. Default = 0 (normal ).

1 = and mode

2 = or mode

3 = xor mode

4 = foreground mode

PROC GetDrawingCursor.XL (VAL INT link, INT x, y)

**Returns the cursor position.**

Output: x, y = the coordinates of the current cursor position.

## **SCREEN and WINDOW ROUTINES**

PROC ClearSelectedScreen.XL (VAL INT link, color, INT error)

**Clears the currently selected screen with a specified color.**

Inputs: color = screen color. Default = 0 (BLACK).

0 <= color <= 255

Output: error = error code.

PROC ClearScreen.XL (VAL INT link, screen.number, color,  
INT error)

**Clears a screen using a specified color.**

Input: screen.number = screen number to be cleared. Default = 0.

0 <= screen.number <= 1

color = screen color,

default = 0 (BLACK).

0 <= color <= 255

Output: error = error code.

PROC SelectScreen.XL (VAL INT link, screen.number, INT error)

**Selects a screen for drawing.**

Input: screen.number = screen number for drawing. Default = 0.

0 <= screen.number <= 1

Output: error = error code.

PROC DisplayScreen.XL (VAL INT link, screen.number,  
INT error)

Selects a screen to be displayed.

Input: screen.number = screen number for display. Default =0.  
0 <= screen.number <= 1

Output: error = error code.

PROC FlipScreen.XL (VAL INT link, new screen, INT error)

Deselects displayed screen and activates designated screen for drawing.

Deselects any displayed windows.

Input: new.screen = screen number to be selected.  
0 <= new.screen <= 1

Output: error = error code.

PROC CopyScreen.XL (VAL INT link, from.screen, to.screen,  
INT error)

Copies the contents of one screen onto another.

Input: from.screen = the screen to be copied from.  
0 <= from.screen <= 1

to.screen = the screen to be copied to.  
0 <= to.screen <= 1

Output: error = error code.

PROC CopyScrnSegToScrn.XL (VAL INT link, from.screen,  
to.screen, from.x, from.y,  
width, length, to.x, to.y, mask,  
INT error)

Copies a screen segment onto another screen.

Input: from.screen = the screen to be copied from.  
0 <= from.screen <= 1

to.screen = the screen to be copied to.  
0 <= to.screen <= 1

from.x, from.y = coordinates of the upper left corner of the  
segment to be copied.

width, length = width and length of the segment to be copied.

to.x, to.y = coordinates of the beginning point on the  
 screen to which the segment is to be copied.  
 mask = copy mask mode.  
         0 = normal copy  
         1 = only copy non-zero bytes  
         2 = only copy zero bytes  
 Output: error = error code.

```

PROC CopyWindSegToWind.XL (VAL INT link, from.window,
                           to.window, from.x, from.y, width, length,
                           to.x, to.y, mask, INT error)
  
```

Copies a window segment onto another window.

Input: from.window = the window to be copied from.  
         0 <= from.window <= 14  
 to.window = the window to be copied to.  
         0 <= to.window <= 14  
 from.x, from.y = the coordinates of the upper left corner of the  
 segment to be copied.  
 width, length = width and length of the segment to be copied.  
 to.x, to.y = coordinates of the beginning point on the  
 window to which the segment is to be copied.  
 mask = copy mask mode.  
         0 = normal copy  
         1 = only copy non-zero bytes  
         2 = only copy zero bytes  
 Output: error = error code.

```

PROC CopyScrnSegToWind.XL (VAL INT link, from.screen,
                           to.window, from.x, from.y, width, length,
                           to.x, to.y, mask, INT error)
  
```

Copies a screen segment onto a window

Input: from.screen = the screen to be copied from.  
         0 <= from.screen <= 1  
 to.window = the window to be copied to.  
         0 <= to.window <= 14  
 from.x, from.y = coordinates of the upper left corner of the

width.length = segment to be copied.  
 = width and length of the segment to be copied.  
 to.x, to.y = coordinates of the beginning point on the window to which the segment is to be copied.  
 mask = define desired copy mask mode.  
 0 = normal copy  
 1 = only copy non=zero bytes  
 2 = only copy zero bytes  
 Output: error = error code.

```
PROC SetWindow.XL (VAL INT link, wondiw.number, x, y,
                  INT error)
```

Sets up a window with given width and length.

Input: window.number = window number.

0 <= window.number <= 14

x,y = width and length of the desired window.

Output: error = error code.

```
PROC ClearSelectedWindow.XL (VAL INT link, color, INT error)
```

Clears the currently selected window using a specified color.

Input: color = window color.

0 <= color <= 255

Output: error = error code.

```
PROC ClearWindow.XL (VAL INT link, window.number, color,
                    INT error)
```

Clears a set window to a specified color

Input: window.number = id number of window to be cleared.

0 <= window.number <= 14

color = window color.

0 <= color <= 255

Output: error = error code.

PROC SelectWindow.XL (VAL INT link, window.number, INT error)

Selects a set window for drawing.

Input: window.number = window number for drawing.

0 <= window.number <= 14

Output: error = error code.

PROC DisplayWindow.XL (VAL INT link, window.number, x, y,  
INT error)

Input: window.number = window number for display

0 <= window.number <= 14

x, y = display position of the window.

Output: error = error code.

PROC ClearSegment.XL (VAL INT link, x1, y1, x2, y2, color,  
INT error)

Clears a segment on a currently active screen/window.

Input: x1, y1 = the coordinates of the top left corner of the segment  
that is to be cleared.

x2, y2 = the coordinated of the bottom right corner of the  
segment that is to be cleared.

color = define desired segment color.

0 <= color <= 255

Output: error = error code.

PROC ClearScreenSeg.XL (VAL INT link, screen.number, x1, y1,  
x2, y2, color, INT error)

Clears a segment segment.

Input: screen.number = screen which contains the segment.

x1, y1 = coordinates of top left corner of segment.

x2, y2 = coordinates of bottom right corner of  
segment

color = segment color.

0 <= color <= 255

Output: error = error code.

```
PROC ClearWindowSeg.XL (VAL INT link, window.number, x1, y1,
                       x2, y2, color, INT error)
```

**Clears a window segment.**

**Input:** window.number = window which contains the segment to be cleared.

          x1, y1       = coordinates of top left corner of segment to be cleared.

          x2, y2       = coordinates of bottom right corner of segment to be cleared.

          color        = segment color.  
                        0 <= color <= 255

**Output:** error       = error code.

```
PROC GetCurrentScrnNum.XL (VAL INT link, INT screen.number)
```

**Returns the current screen number.**

**Output:** screen.number = current screen number

```
PROC GetDisplayedScrnNum.XL (VAL INT link, INT screen.number)
```

**Returns the current displayed screen number.**

**Output:** screen.number = current displayed screen number.

## **TEXT ROUTINES**

```
PROC DrawChar.XL (VAL INT link, char.number, INT error)
```

**Draws a character provided by the character number.**

**Input:** char.number = character number

          0 <= char.number <= 255

```
PROC WriteStr.XL (VAL INT link, str.size, VAL [] BYTE text,
                 INT error)
```

**Writes a string of text horizontally without justification.**

**Input:** str.size = string size. IF "0" is input, the program will calculate the size.

          text = character string to be drawn.

**Output:** error = error code.

PROC WritesStrJustify.XL (VAL INT link, str.size,  
                           VAL [] BYTE text, INT error)

Writes a string of text according to the justification.

Input: str.size = string size. IF ")" is input, the program will  
           calculate the size.

          text = character string to be drawn.

Output: error = error code.

PROC WriteNum.XL (VAL INT link, number, INT error)

Writes a given integer number on screen.

Input: number = integer number to be drawn on screen.

Output: error = error code.

PROC WriteRealNum.XL (VAL INT link, VAL REAL32 realnumber,  
                           VAL INT Ip, Dp, INT error)

Writes a given real number on screen.

Input: realnumber = real number to be drawn on screen.

          Ip = number of digits to the left of the decimal point.

          Dp = number of digits to the right of the decimal  
           point.

Output: error = error code.

PROC Scroll.XL (VAL INT link, INT error)

Scroll currently selected window/screen by one line

Output: error = error code.

PROC JumpScroll.XL (VAL INT link, INT error)

Scroll currently selected window/screen. The number of lines is  
           determined by the y height multiplied by the character height.

Output: error = error code.

PROC CarriageReturn.XL (VAL INT link, INT error)

Returns the cursor to the left edge of the currently selected  
           screen/window.

Output: error = error code.

PROC LineFeed.XL (VAL INT link, INT error)

Returns the cursor to the left edge of the currently selected screen/window.

Output: error = error code.

PROC LineFeed.XL (VAL INT link, INT error)

Increases the cursor position by the height of the characters being drawn. Scrolling is automatic if issued at the bottom of the screen/window.

Output: error = error code.

PROC SetTextStyle.XL (VAL INT link, font, text,dirct,  
width, height)

Sets desired test style.

Input: font = font id. Default= 0.

Currently, only the default is available.

text.dirct = text orientation. Default = 0.

1 = vertical

width = character width. Default = 1.

height = character height. Default =1.

PROC SetTextJustify.XL (VAL INT link, horiz, vert)

Sets desired text justify.

Input: horiz = horizontal text justification. Default = 0 (left justify).

1 = center justify

2 = right justify

vert = vertical text justification. Default = 0 (top justify).

1 = center justify

2 = bottom justify

PROC MoveTextTo.XL (VAL INT link, x, y, INT error)

Moves the text cursor position to specific position.

Input: x, y = the coordinates of the desired text cursor position.

Output: error = error code.

PROC MoveTextRel.XL (VAL INT link, delta.x, delta.y,  
INT error)

Moves the text cursor position relative to its current position.

Input: delta.x, delta.y = the distance from text current position.

Output: error = error code.

PROC Rotate.XL (VAL INT link, turn, INT error)

Rotates all subsequent characters through 0, 1, 2, or 3 quarter turns in a clockwise direction.

Input: turn = number of the turn.

Output: error = error code.

PROC ReflectX.XL (VAL INT link, INT error)

Reflects the textual output in the x plane. Can be cancelled by issuing a second reflect command.

Output: error = error code.

PROC ReflectY.XL (VAL INT link, INT error)

Reflects the textual output in the y plane can be cancelled by issuing a second reflect command.

Output: error = error code.

PROC GetTextSize.XL (VAL INT link, INT x.size, y.size)

Returns the text size. X.size is text width, y.size is text height.

Output: x.size = current text width.

y.size = current text height.

PROC GetTextCursor.XL (VAL INT link, INT x.axis, y.axis)

Returns the text cursor position.

Output: x.axis, y.axis = current text cursor position.

## CRT ROUTINES

PROC Xwidth.XL (VAL INT link, width)

Sets the pixel width.

Input: width = define desired pixel width. Default = 1.

PROC Yheight.XL (VAL INT link, height)

**Sets the pixel height.**

**Input:** height = define desired pixel height. Default = 1.

PROC SetLineFrequency.XL (VAL INT link, line.frequency)

**Sets the scan line frequency for CRT.**

**Input:** line.frequency = desired line frequency.

PROC SetFrameRate.XL (VAL INT link, frame.rate)

**Sets the frame for CRT.**

**Input:** frame.rate = desired frame rate.

PROC Interlace.XL (VAL INT link)

**Toggles interlace ON/OFF.**

PROC SetPixelClock.XL (VAL INT link, pixel.clock)

**Sets pixel clock frequency.**

**Input:** pixel.clock = desired pixel clock.

PROC Int.crt.XL (VAL INT link)

**Resets on receipt CRT controller.**

PROC BlankCrt.XL (VAL INT link)

**Blanks the monitor.**

PROC UnblankCrt.XL (VAL INT link)

**Unblanks the monitor.**

## Error Codes

Below is a list of possible errors returned by XPGRAPHICS routines. If the requested action was completed successfully, an error status of 0 is returned; otherwise, one of the listed errors is specified.

- 0 = successful completion
- 1 = coordinates out of screen/window range
- 2 = screen other than 0 or 1 selected
- 3 = window selected has not been set
- 4 = insufficient storage left for window definition
- 5 = the maximum number of windows has been exceeded
- 6 = drawing mode selected not in the range
- 7 = rotations allowed only in quarter turns, i.e. (0-3)
- 8 = color selection out of range of color look up table (0-255)
- 9 = character requested was not in the range of character font table (0-255)
- 10 = only 255 characters allowed in strings
- 11 = color table requested does not exist (0-2)
- 12 = command set to graphics package does not exist
- 13 = invalid window dimensions
- 14 = invalid window position
- 15 = invalid screen/window dimensions
- 16 = window has not been selected

**APPENDIX G**

**KEYWORDS RECOGNIZED BY THE NASTRAN INTERFACE**

**MODULE**

The following list contains the NASTRAN keywords recognized by the NASTRAN interface module in XPFEM. Currently the interface module recognizes only keywords associated with the displacement solution method in MSC and COSMIC NASTRAN.

**MSC KEYWORDS**

CHEXA  
CQUAD8  
CQUAD4  
CQDMEM1  
CTRIA6  
CTRIA3

**COSMIC KEYWORDS**

CIHEX1  
CIHEX2  
CQDEM1  
CTRIM6  
CTRMEM

**GENERAL KEYWORDS**

\$  
CORD1R  
CORD2R  
ENDDATA  
FORCE  
GRID  
GRDSET  
MAT1

PARAM  
PBAR  
PIHEX  
PQDMEM1  
PSHELL  
PSOLID  
SPC  
SPC1

## APPENDIX H

### PARALLEL MESH GENERATION with FINITE QUADTREE

The FINITE QUADTREE mesh generator uses tree data structures which are not readily adaptable for parallel calculations due to the extensive amount of interprocessor communication needed to maintain trees which are evenly balanced across processors. To adapt FINITE QUADTREE for parallel mesh generation, groups of independent elements, i.e., elements not associated through common degrees of freedom, must be identified. Such groups of independent elements can be identified using a coloring scheme. A coloring scheme selectively assigns a color set to a group of dependent elements. The color set is repeated for each group of dependent elements; thus, all elements of a specific color are independent of each other and available for parallel processing.

Two requirements of a coloring scheme are computational efficiency and minimization of the color set. Computational efficiency is a requirement because coloring schemes require large amounts of data and are not easily parallelized; therefore, one or a small number of processors will be employed for the coloring. Minimizing the size of the color set is a requirement to maintain coarse granularity and hence reduce the frequency of synchronization and communication.

Algorithms to color arbitrary finite element meshes with a minimum number of colors are data- and CPU-intensive. Simple and efficient coloring algorithms, however, can be developed when elements have a regular pattern. In FINITE QUADTREE, the quadrants used to generate the finite elements have such a regular pattern and thus will be the entities to be colored for parallel mesh generation.

The simplest coloring procedure would color quadrants as sets of independent quadrants based on the level of the terminal quadrants in the tree. This form of coloring is quite efficient since it can be accomplished by

performing one complete tree traversal in which each terminal quadrant is assigned the appropriate color based on level in tree and location of quadrants with respect to its parents. Each level requiring a separate set of colors uses four new colors.

In the current implementation of FINITE QUADTREE, neighboring quadrants that share an edge are allowed a one level difference. This restriction allows quadrants that share a corner to have a two level difference. Therefore, it is possible for three different levels to be dependent on each other since they share common degrees of freedom. Thus a total of twelve colors are required to identify independent quadrants (Figure H.1).

Clearly twelve colors is more than the minimum needed to ensure all quadrants neighboring a given quadrant have a different color. For example, it is possible to manually color the set of quadrants in Figure H.1 with six colors such that neighboring quadrants have different colors. If the one level difference were enforced across the nodes as well as the edges, the mesh could be colored with the above procedure using only eight colors. If the two level difference at corners were allowed in Figure H.1, dependent quadrants at such corners would color the same, which is not correct. However, at such a corner, if the quadrant at the lower tree level were given the color that would have been assigned to its parent quadrant, the coloring problem across the corner would be eliminated. Figure H.2 displays this concept applied to the set of quadrants in Figure H.1. As seen in Figure H.2, the resulting set of eight colors provides an acceptable set of colors.

The application of the procedure outlined above is more complex than assigning colors based solely on level during a tree traversal. It specifically requires the identification of quadrant corners where a two level difference exists. However, this identification process can be done with reasonable efficiency during an overall tree traversal process. Currently, stored information for each quadrant indicates if neighboring quadrants are one level smaller. A two level corner difference could be detected by determining quadrants with neighbors of two different tree levels. When these cases are detected, a localized tree traversal can be applied to find the level of the lower corner quadrant.

Further effort is required to ensure the functionality of the eight color scheme. Additionally, it will be necessary to develop procedures to undo color assignments during an adaptive refinement process which eliminates a two level difference at a corner.

1	4		5	8	4	
			6	7		
2	5	8	5	9   12	5	8
	6	7	6	10   11	6	7
1	4		5	9   12	5	8
			6	10   11	6	7
2	3		5	8	3	
			6	7		

1	4
2	3

5	8
6	7

9	12
10	11

**FIGURE H.1**  
**Use of twelve colors to**  
**identify independent quadrants**

1	4		5	8	4		
			6	7			
2	5	8	5	1	8	5	8
	6	7	6	2	3		
1	4		5	1	4	5	8
			6	2	3		
2	3		5	8	3		
			6	7			

1	4
2	3

5	8
6	7

**FIGURE H.2**  
**Use of eight colors to**  
**identify independent quadrants**

1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18  
19  
20  
21  
22  
23  
24  
25  
26  
27  
28  
29  
30  
31  
32  
33  
34  
35  
36  
37  
38  
39  
40  
41  
42  
43  
44  
45  
46  
47  
48  
49  
50  
51  
52  
53  
54  
55  
56  
57  
58  
59  
60  
61  
62  
63  
64  
65  
66  
67  
68  
69  
70  
71  
72  
73  
74  
75  
76  
77  
78  
79  
80  
81  
82  
83  
84  
85  
86  
87  
88  
89  
90  
91  
92  
93  
94  
95  
96  
97  
98  
99  
100