NONLINEAR STRUCTURAL RESPONSE USING ADAPTIVE DYNAMIC RELAXATION ON A MASSIVELY-PARALLEL-PROCESSING SYSTEM

David R. Oakley *
Clemson University
Clemson, SC 29634-0921

Norman F. Knight, Jr. **
Old Dominion University
Norfolk, VA 23529-0247

Abstract - A parallel adaptive dynamic relaxation (ADR) algorithm has been developed for nonlinear structural analysis. This algorithm has minimal memory requirements, is easily parallelizable and scalable to many processors, and is generally very reliable and efficient for highly nonlinear problems. Performance evaluations on single-processor computers have shown that the ADR algorithm is reliable and highly vectorizable, and that it is competitive with direct solution methods for the highly nonlinear problems considered. The present algorithm is implemented on the 512-processor Intel Touchstone DELTA system at Caltech, and it is designed to minimize the extent and frequency of interprocessor communication. The algorithm has been used to solve for the nonlinear static response of two- and three-dimensional hyperelastic systems involving contact. Impressive relative speedups have been achieved and demonstrate the high scalability of the ADR algorithm. For the class of problems addressed, the ADR algorithm represents a very promising approach for parallel-vector processing.

1. INTRODUCTION

Use of the finite element method to solve structural problems of increasing computational size and complexity continues to be the focus of intense research. Yet even on current high-speed vector computers, solution costs, especially for transient dynamic analyses, are often prohibitive. Emerging high-performance computers offer tremendous speedup potential for these types of applications, provided an optimal solution strategy is implemented. Existing sequential solution procedures may be adapted to operate on these computers. However, these procedures have been developed and customized for sequential operation and may not be the best approach for parallel processing. To exploit this potential fully, problem formulations and solution strategies need to be re-evaluated in light of their suitability for parallel and vector processing. As such, the overall goal of this research is to develop an adaptive algorithm for predicting static and dynamic response of nonlinear hyperelastic structures which exploits these emerging high-performance computing systems.

* Dean's Scholar, Department of Mechanical Engineering

** Associate Professor, Department of Aerospace Engineering

The basic formulation for the adaptive dynamic relaxation (ADR) algorithm for hyperelastic structures is given by Oakley and Knight [1]. Dynamic relaxation is a technique by which the static solution is obtained by determining the steady-state response to the transient dynamic analysis for an autonomous system. In this case, the transient part of the solution is not of interest, only the steady-state response is desired. Since the transient solution is not desired, fictitious mass and damping matrices which no longer represent the physical system are chosen to accelerate the determination of the steady-state response. These matrices are redefined (using existing equations) so as to produce the most rapid convergence. For highly nonlinear problems where stiffness changes significantly during the analysis, adaptive techniques exist which automatically update the integration parameters when necessary [6].

An ADR algorithm represents a unified approach for both static and transient dynamic analyses, and is known to be very competitive for certain problems with high nonlinearities and instabilities [6,7,8]. Reliability is ensured by integration parameters which are adaptively changed throughout an analysis to accommodate these nonlinear effects. It is based on an explicit direct-time integration method and a very small time step is generally required to ensure numerical stability; however, the computational cost per time step is very low and is mostly associated with evaluation of the internal force vector.

The present paper builds on a study which was begun to evaluate the potential of the ADR algorithm for solving complex structural problems on parallel-vector computers. The formulation of an ADR algorithm with application to nonlinear hyperelastic structures is presented in reference [1] including a complete derivation of the algorithm and the problem adaptive scheme used to ensure reliability and improve performance. Finite element equations are derived for the nonlinear analysis of elastic and hyperelastic solids subject to large deformations. A very simple and efficient algorithm based on solver constraints is developed to enforce frictionless contact conditions.

The performance of a sequential implementation of the ADR algorithm is evaluated in reference [2] using a Convex C240 minisupercomputer. A new organization of the finite element computations is implemented to exploit vector processing. Two- and three-dimensional test cases are used to assess the analysis and performance capabilities of the algorithm. Performance of the ADR algorithm for the nonlinear static analysis of each test case is compared with that of an existing finite element code which employs the Newton-Raphson procedure and a highly optimized Cholesky skyline solver. Relative speedups due to vectorization are also presented. This algorithm is found to be reliable and highly vectorizable, and it outperforms the direct solution method for the highly nonlinear problems considered.

The performance of a parallel implementation of the ADR algorithm is evaluated in reference [3] on a cluster of 16 Sun SPARC IPX workstations using PVM [10] and on a 128-processor Intel iPSC/860 hypercube.

The parallel implementation is designed such that each processor executes the complete sequential algorithm on a subset of elements. One-dimensional strip partitioning and two-dimensional block partitioning are used to divide the problem domain among the available processors. Load balancing is ensured by using structured meshes of one material. Efficient schemes are implemented to accomplish the required nearest-neighbor and global communication. Relative speedups are presented for the nonlinear static analysis of the 2-D and 3-D test cases developed and analyzed in reference [2]. Impressive relative speedups are achieved on the Hypercube and demonstrate the high scalability of the ADR algorithm. Good relative speedups are also achieved using PVM on a cluster of networked workstations.

The objective of this paper is to evaluate the performance of a massively parallel implementation of the ADR algorithm using the 512-processor Intel Touchstone DELTA system. Relative speedups are presented for the test cases evaluated in reference [3]. Final results and completion times for each test case are also given.

The remainder of this paper is organized as follows. In Section 2, the formulation and sequential implementation of the ADR algorithm are reviewed, then the general parallel-processing approach is described. In Section 3, the Intel Touchstone DELTA system is described, a general flowchart for the complete parallel algorithm is presented, and interprocessor communication details are discussed. In Section 4, the test cases are reviewed and performance results are presented and evaluated. Conclusions are given in Section 5.

2. PARALLEL-PROCESSING APPROACH

This section begins with a review of the basic formulation and sequential implementation of the ADR algorithm. Afterwards, the general parallel-processing approach is described. The partitioning strategy is presented, load balancing characteristics are discussed, and details of the parallel solution process are given.

2.1 Formulation of the ADR Algorithm

The ADR algorithm is based on the following semi-discrete equations of motion governing structural dynamic response for the n^{th} time increment

$$\mathbf{M}\ddot{\mathbf{D}}^n + \mathbf{C}\dot{\mathbf{D}}^n + \mathbf{F}(\mathbf{D}^n) = \mathbf{P}^n \tag{2.1}$$

where $\mathbf{M}$ is a diagonal mass matrix, $\mathbf{C}$ is a damping matrix, $\mathbf{F}$ is the internal force vector, and $\mathbf{P}$ is a vector of external loads. The vectors $\ddot{\mathbf{D}}, \dot{\mathbf{D}}$, and $\mathbf{D}$ represent the acceleration, velocity, and displacement vectors, respectively. The internal force vector $\mathbf{F}$ is a function of the displacements and may be assembled on an element-by-element basis [1].

The ADR algorithm involves the use of an explicit numerical time integration technique to solve equation (2.1). In the current algorithm, a half-station, central-difference technique is used which provides the following approximations for the temporal derivatives

$$\dot{\mathbf{D}}^{n+\frac{1}{2}} = \frac{1}{h}(\mathbf{D}^{n+1} - \mathbf{D}^{n}) \tag{2.2}$$

$$\ddot{\mathbf{D}}^{n} = \frac{1}{h}(\dot{\mathbf{D}}^{n+\frac{1}{2}} - \dot{\mathbf{D}}^{n-\frac{1}{2}}) \tag{2.3}$$

where h is a fixed time increment, and $\dot{\mathbf{D}}^{n}$ is averaged over a time step as

$$\dot{\mathbf{D}}^{n} = \frac{1}{2}(\dot{\mathbf{D}}^{n+\frac{1}{2}} + \dot{\mathbf{D}}^{n-\frac{1}{2}}) \tag{2.4}$$

Substituting equations (2.3) and (2.4) into equation (2.1) and assuming mass-proportional damping ($\mathbf{C} = c\mathbf{M}$) yields the fundamental time marching equations for advancing the velocity and displacement vectors to the next time step. Thus,

$$\dot{\mathbf{D}}^{n+\frac{1}{2}} = \left(\frac{2-ch}{2+ch}\right)\dot{\mathbf{D}}^{n-\frac{1}{2}} + \left(\frac{2h}{2+ch}\right)\mathbf{M}^{-1}(\mathbf{P}^{n} - \mathbf{F}^{n}) \tag{2.5}$$

$$\mathbf{D}^{n+1} = \mathbf{D}^{n} + h\dot{\mathbf{D}}^{n+\frac{1}{2}} \tag{2.6}$$

Using an explicit time integration technique, the resulting system of equations are linear, even for nonlinear problems. Also, if a diagonal mass matrix is used, the matrix inverse of $\mathbf{M}$ is a trivial computation and these equations represent an uncoupled system of algebraic equations in which each solution component may be computed independently. For transient dynamic analysis, a time history of displacements (system response) is sought. Mass and damping vectors which best model the physical properties of the system are used. Techniques for estimating the maximum allowable time step size are available, such that the time step size may change during the transient dynamic analysis. As such, explicit time integration methods are attractive candidates for implementation on high-performance computers. They generally have low memory and communication requirements but are also only conditionally stable numerically.

The objective of a static analysis using the ADR algorithm is to obtain the steady-state solution of the pseudo-transient response. Thus, each time step is in fact an iteration or pseudo time step. The mass and damping parameters generally do not represent the physical system. Instead, they are defined so as

to produce the most rapid convergence, where convergence herein is based on a relative error of the force imbalance or

$$\epsilon = \frac{\|\mathbf{P} - \mathbf{F}^n\|}{\|\mathbf{P}\|} = \frac{\|\mathbf{R}^n\|}{\|\mathbf{P}\|} \leq \epsilon_{tol} \tag{2.7}$$

where $\mathbf{P}$ is the static load and $\mathbf{R}^n$ is the residual force vector for time step n. Note that when convergence (i.e., steady-state response) is obtained, the internal forces balance the external forces, and the inertial forces vanish.

A derivation of the fictitious mass and damping equations for the ADR algorithm is given in reference [1]. Based on Gerschgorin's theorem, this new mass matrix is defined as

$$M_{ii} \geq \frac{h^2}{4} \sum_{j=1}^{n} |K_{ij}| \quad \text{or} \quad \mathbf{M} \geq \frac{h^2}{4}\mathbf{S} \tag{2.8}$$

where $\mathbf{S}$ represents a lumped stiffness which may be computed on an element-by-element basis as follows

$$\mathbf{S} = \sum_{e=1}^{nelem} \mathbf{s_e} \quad \text{where} \quad (s_{ii})_e = \sum_{j=1}^{ndof} |k_{ij}|_e \tag{2.9}$$

The quantities k_{ij} in this expression correspond to the entries in the element stiffness matrix $\mathbf{k_e}$ (see reference [1]). As shown in equation (2.8), the fictitious mass $\mathbf{M}$ and time step size h are not independent. However, once either is specified, the other value may be readily computed. Herein, the time step size is arbitrarily set to one. Numerical experiments with other values (e.g., 10, 100, 1000) confirm that this choice is arbitrary and no change in algorithm performance or convergence characteristics is observed. The new damping coefficient c is computed using

$$c = 2\sqrt{\lambda_o} \quad \text{where} \quad \lambda_o \cong \frac{(\dot{\mathbf{D}}^{n-\frac{1}{2}})^T \bar{\mathbf{S}}^n \dot{\mathbf{D}}^{n-\frac{1}{2}}}{(\dot{\mathbf{D}}^{n-\frac{1}{2}})^T \mathbf{M} \dot{\mathbf{D}}^{n-\frac{1}{2}}} \tag{2.10}$$

The parameter λ_o represents the lowest eigenvalue estimated from the mass-stiffness Rayleigh quotient shown. The vector $\bar{\mathbf{S}}^n$ is a diagonal estimator of the directional stiffness after step n, the components of which are given by

$$\bar{S}_i^n = \frac{F_i^n - F_i^{n-1}}{h\dot{D}_i^{n-\frac{1}{2}}} \tag{2.11}$$

The new damping coefficient c is updated every time step since it only involves minimal computations. The new mass matrix must be evaluated at the beginning of the analysis, and in addition, subsequent updates are sometimes necessary to maintain numerical stability. Stability is determined using the perturbed apparent-frequency error indicator [6] where values of ϵ_i greater than one represent potential unstable numerical conditions. This error indicator is given by

$$\epsilon_i = \frac{h^2}{4} \frac{|\ddot{D}_i^n - \ddot{D}_i^{n-1}|}{|D_i^n - D_i^{n-1}|} \tag{2.12}$$

2.2 Solution Process

A flowchart of the sequential implementation of the ADR algorithm is shown in Figure 1. It begins with initialization of time step (or iteration) counter n, and the two key flags *istif* and *iend*. The *istif* flag controls evaluation of lumped stiffness matrix **S** (equation 2.9) and fictitious mass matrix **M** (equation 2.8). After the first iteration, these quantities are only updated as necessary to maintain numerical stability. The *iend* flag controls exits from the solution process. It is activated when convergence is achieved, when numerical problems (such as collapsing elements) are detected, or when the number of time steps (or iterations) has reached the user-specified limit.

As shown in Figure 1, a four-phase process is executed for each time step. Phase 1 consists of finite element computations in which internal force vector **F** and possibly the lumped stiffness matrix **S** are evaluated on an element-by-element basis. Phase 2 is the adaptive stage in which integration parameters of the ADR algorithm are updated. If necessary, **S** may be used to re-evaluate **M** based on equation (2.8). Scalar quantities $\dot{\mathbf{D}}^T\bar{\mathbf{S}}\dot{\mathbf{D}}$ and $\dot{\mathbf{D}}^T\mathbf{M}\dot{\mathbf{D}}$ are computed and used to evaluate the damping coefficient c in accordance with equation (2.10). Phase 3 is the solution step for the $n+1$ step in which displacements for the next time step are computed from equations (2.5) and (2.6). Contact conditions are also enforced in this phase using the solver-constraints technique described in reference [1]. If unstable conditions exist as determined from equation (2.12), *istif* is reset to 1. Phase 4 evaluates the force imbalance and convergence. The scalar quantity $\|\mathbf{R}\|$ is computed and then the error is determined using equation (2.7). The scalar quantity $\|\mathbf{P}\|$ in equation (2.7) is constant for the test cases considered herein and is evaluated once at the beginning of the analysis.

2.3 Parallel Implementation

The ADR algorithm is very amenable to parallel processing. All vector quantities needed for the solution step may be computed on an element-by-element basis. The calculations required for each element are completely independent, and the system of equations for computing the displacement solution at the next

time step are completely uncoupled. The primary objective for developing a parallel implementation is to maximize performance (in terms of relative speedup) by minimizing the frequency and extent of interprocessor communication and maximizing load balancing. To accomplish this objective, the ADR algorithm is parallelized by having each processor execute the complete sequential algorithm on a subset of elements.

Two different partitioning schemes are used to divide the problem domain among the available processors. The first is referred to as one-dimensional strip partitioning and views the system topology as a 1-D array of m processors. As described in reference [9], the finite element mesh is partitioned along the longest element-wise dimension into m strips, and the elements within each strip are assigned to a particular processor. The strip partitions of a cantilever beam test case are illustrated in Figure 2a for the case of 16 processors. The 1-D array view of the system topology is shown in Figure 2b, and the corresponding processor assignments for each partition are indicated. The test cases used in this research are very amenable to this type of partitioning as they all have one dominant dimension in terms of the number of elements in one direction. This method is relatively simple to implement and, as shown in Figure 2a, limits the neighbors of each partition to two.

One-dimensional strip partitioning is only feasible if the maximum number of available processors is less than or equal to the number of elements along the longest element-wise dimension. If a large number of processors are used such that this condition is no longer satisfied, then a second partitioning scheme, referred to as two-dimensional block partitioning, becomes necessary. This method views the system topology as an $n \times m$ array of processors, where n is the number of rows of processors and m is the number of processors per row. As described in reference [9], the finite element mesh is first partitioned along the longest element-wise dimension into n strips. Each strip is then partitioned along its length into m blocks, and the elements within each block are assigned to a particular processor. The block partitions of a cantilever beam test case are illustrated in Figure 3a for the case of 16 processors. In this example, the system topology is viewed as a 4×4 array of processors as shown in Figure 3b. It can be seen from Figure 3a that with this method, each partition has at most eight surrounding neighbors.

Balancing the workload among the processors of a concurrent computer is central to achieving high performance. Overall relative speedup is generally limited by the maximum CPU time required for any processor to complete its assigned work. Since identical processors are typically available on a given parallel processing system, the amount of computations or work assigned to each processor needs to be nearly the same in order to avoid processors being idle. Thus, an important consideration is to ensure that each processor performs the same amount of work. As noted in Section 1, only structured meshes of uniform size, type, and material are considered herein. Therefore, load balancing is primarily a function of the

number of elements assigned to each processor. As such, the partitioning is performed so that each processor receives the same number of elements. A more sophisticated mapping or partitioning strategy (e.g., domain decomposition) would have to be employed to achieve sufficient load balancing with an unstructured mesh or if different materials are involved in the same finite element model.

A flowchart of the solution procedure executed on a given processor is shown in Figure 4. All computations are now for the elements assigned locally to that processor. As shown, the algorithm is very similar to the sequential solution procedure given in Figure 1 with just a few important differences related to interprocessor communication.

Nearest-Neighbor Communication

One nearest-neighbor communication sequence is required for each time step (in Phase 1 of Figure 4). This may be illustrated in reference to Figures 2 and 3 which show the finite element mesh for a cantilever beam partitioned among 16 processors. Considering any two adjacent partitions, nodes along the interface are shared by elements that are allocated to two different processors. As a result, the internal force vector **F** and lumped stiffness matrix **S** computed for the interface nodes on a given processor are only partially complete. Contributions must be obtained from the neighboring processors before the solution process can continue.

For 1-D strip-partitioned domains, each partition may have a left and right neighbor. The required nearest-neighbor communication can be accomplished in two steps as follows. In the first step, each processor sends **F** and **S** values for the left interface to its left neighbor and then receives the corresponding values being sent from its right neighbor as shown in Figure 5a. The second step consists of repeating this process in an analogous fashion for the right interface nodes (see Figure 5b). The end result is an exchange of the interface values between neighboring processors. This approach synchronizes the communication sequence among the processors, thereby improving efficiency and avoiding the possibility of deadlock (e.g., two processors sending different variables simultaneously, and each waiting for the other's variable to be sent before proceeding).

For 2-D block-partitioned domains, each partition may have eight surrounding neighbors. Even so, the nearest-neighbor communication can be accomplished in four steps [4]. First, the left and right interface values are exchanged in the horizontal direction as shown in Figure 6a. This is accomplished using the two-step sequence described earlier for strip-partitioned domains. After each processor has updated its left and right interface values with contributions received from its horizontal neighbors, the upper and lower interface values are exchanged in the vertical direction using the same two-step sequence as before (see Figure 6b). The end result is an exchange of interface values, first in the horizontal direction, and then in the vertical

direction. Using this procedure, interface values for the corner nodes of each block (along with the other left and right interface values) are updated after the horizontal exchange to include the contribution from horizontal neighbors. Since the vertical exchange operates on these updated values, communication between neighboring diagonal partitions is automatically satisfied.

Global Communication

Two global communication sequences are required for each time step. The first is needed for computation of the scalar damping coefficient c. Recall from the sequential implementation that c is evaluated for each time step or iteration based on the vector products $\dot{\mathbf{D}}^T\bar{\mathbf{S}}\dot{\mathbf{D}}$ and $\dot{\mathbf{D}}^T\mathbf{M}\dot{\mathbf{D}}$. These quantities may be expressed as

$$(\dot{\mathbf{D}}^{n-\frac{1}{2}})^T\bar{\mathbf{S}}^n\dot{\mathbf{D}}^{n-\frac{1}{2}} = \sum_{i=1}^{np}\sum_{j=1}^{ndof}\dot{D}_j^{n-\frac{1}{2}}\bar{S}_j^n\dot{D}_j^{n-\frac{1}{2}} = \sum_{i=1}^{np}A_i \tag{2.13}$$

$$(\dot{\mathbf{D}}^{n-\frac{1}{2}})^T\mathbf{M}^n\dot{\mathbf{D}}^{n-\frac{1}{2}} = \sum_{i=1}^{np}\sum_{j=1}^{ndof}\dot{D}_j^{n-\frac{1}{2}}M_j^n\dot{D}_j^{n-\frac{1}{2}} = \sum_{i=1}^{np}B_i \tag{2.14}$$

where np refers to the number of processors and $ndof$ denotes the degrees of freedom associated with the elements on a given processor (it is understood that contributions from the interface degrees of freedom are included only once).

Each processor must use the same value of the damping constant c, and this value must be the same as that computed in the sequential solution (i.e., it must be based on global $\dot{\mathbf{D}}, \bar{\mathbf{S}}$, and $\mathbf{M}$ vectors). To accomplish this, the following strategy is used. Each processor computes the local vector products A and B given by equations (2.13) and (2.14), respectively. A global sum of A and B is then performed across all processors such that they all end up with the same complete values for $\dot{\mathbf{D}}^T\bar{\mathbf{S}}\dot{\mathbf{D}}$ and $\dot{\mathbf{D}}^T\mathbf{M}\dot{\mathbf{D}}$. The correct damping coefficient may then be computed and utilized by each processor.

A second global communication sequence is needed after computation of the new local displacements (i.e., during Phase 4). Several events must occur at this point as discussed for the sequential implementation. Convergence must be evaluated based on the current residual force imbalance error, and flags must be set to control continuation of the solution process and updates of the fictitious mass.

The error computation given by equation (2.7) needed to assess convergence is similar to that of the damping coefficient. It must be based on global system response, and the final result is needed by each processor. In this case, the same global summation procedure is used to supply each processor with complete

identical values for the vector product $\mathbf{R}^T\mathbf{R}$. As mentioned earlier, the quantity $\mathbf{P}^T\mathbf{P}$ is constant for static analysis, therefore it is computed once and provided as input for each processor.

Flags *iend* and *istif* represent local conditions on each processor, however the actions they initiate must be performed by all processors. If one processor detects the need for a fictitious mass update, all processors must perform this update. Similarly, if collapsing elements occur on one processor, all processors must exit the solution process. This information is also communicated by a global sum procedure. The appropriate actions are then taken by each processor if the final summation result for each flag is greater than or equal to one.

Summary

In summary, the primary difference between the sequential and parallel solution process is that the parallel algorithm requires one nearest-neighbor and two global communication sequences each time step. For a given partitioning scheme, the nearest-neighbor communication is independent of the number of processors but is dependent on problem size as larger problems would imply more interface nodes. The global communication is independent of problem size, since it only involves scalar quantities, but is dependent on the number of processors. For large problems, the global communication time should represent only a small fraction of the total communication cost. Additional implementation details for nearest-neighbor and global communication is topology dependent and will be discussed in Section 3.

3. PARALLEL ALGORITHM

The parallel ADR algorithm is developed for implementation on the Intel Touchstone DELTA system referred to herein as simply the DELTA. The unvectorized code is used as the baseline code. Vectorization can be accomplished on the DELTA; however, it is not as straightforward to implement as on a Convex computer [2]. Thus, the effect of vectorization on parallel performance will not be investigated in this paper. This section begins with a description of the DELTA. Afterwards, a general description of the complete parallel algorithm is given and communication details are discussed.

3.1 Intel Touchstone DELTA System

The Intel Touchstone DELTA system is a multiple-instruction multiple-data (MIMD) type computer. The processor interconnection network represents a two-dimensional mesh topology rather than a hypercube topology and is illustrated for 12 processors in Figure 7. This architecture is advantageous for large finite element computations in two ways. First, it is scalable (i.e., its communication performance does not degrade as the number of processors is increased). This feature is important as the size of finite element problems

continues to increase, requiring more and more processors to achieve results in a reasonable time frame. Secondly, its use of local memory alleviates many of the problems and losses in efficiency associated with memory contention on shared-memory computers.

The DELTA system consists of a total of 512 numeric processors. Each processor is an Intel i860 processor with a 40 MHz clock speed and 16 megabytes of local nonshared memory. A peak performance of 60 MFLOPS per processor is attainable for double-precision floating-point computations. A diagram of the system is shown in Figure 8. The parallel processors are configured in a 16×32 array with a two-dimensional mesh interconnection network. In contrast to the hypercube interconnection topology in which each processor is directly connected to n neighbors for an n-dimension cube, with the mesh topology of the DELTA, each processor is directly connected to at most four neighbors regardless of the total number of processors being used (see Figure 3.11). This results in a lower interprocessor connectivity than the hypercube topology. Interprocessor communication is accomplished with mesh routing chips which enable messages to go directly to the receiving processor without interrupting any of the others. The mesh routing chips are designed to be faster than the routing modules of the Intel iPSC/860 hypercube, in order to make up for the additional distance messages may have to travel due to the lower connectivity of the mesh topology. The DELTA system has two gateway processors Delta1 and Delta2 which act as interfaces between the parallel processors and the ethernet network. These two systems may be viewed as two Unix systems on the network, and they serve as a base from which applications may be run on the system. The system is networked to remote workstations which are used for editing, compiling, and linking the parallel algorithms and also for any pre- and post-processing.

3.2 Complete Parallel Algorithm

A general flowchart for the complete parallel algorithm is shown in Figure 9. It consists of sequential pre- and post-processing programs which run on a remote workstation and a node program which runs on each of the parallel processors. The pre-processing program reads the input data required for the analysis and prepares this data for the parallel solution process. The input data is first mapped into the appropriate arrays needed for finite element computations. The resulting information is then partitioned and written out in the form of an input file for each processor. These files are transferred to one of the two gateway processors Delta1 or Delta2. From the gateway processor, an $n \times m$ array of parallel processors is allocated, and the node program is loaded and started on each. On a given processor, the node program reads its input data file and then cycles through the ADR solution process for its subdomain of elements. When the solution process is complete, each processor writes its local results to a file. These result files are then transferred

back to the remote workstation, and the post-processing program is executed. The post-processing program reads the result files for each processor, assembles the global results, and then writes these results to a file.

Nearest-Neighbor Communication

Depending on the partitioning method being employed, nearest-neighbor communication is accomplished using one of the two message-passing sequences described in Section 2. To implement this communication effectively on the DELTA, the domain-to-processor mapping must account for the number of rows and columns of processors to be utilized and the associated numbering scheme. The DELTA permits allocation of an $n \times m$ array of processors, where n is the number of rows of processors ($1 \leq n \leq 16$) and m is the number of processors per row ($1 \leq m \leq 32$). The numbering scheme is illustrated in Figure 7. As shown, processors are numbered sequentially from left to right in each row, starting at the upper left-hand corner of the $n \times m$ array.

For 1-D strip-partitioned domains, the mapping strategy may be illustrated by considering a cantilever beam to be analyzed using 16 processors. If a single row of 16 processors is allocated as shown in Figure 2b, the processors are assigned to each strip in consecutive order as illustrated in Figure 2a. If multiple rows of processors are allocated (such as a 4×4 array), they are viewed as a 1-D array with the configuration shown in Figure 10a. Based on this convention, the processors are assigned to each strip following the order indicated in Figure 10b. Regardless of whether a single row or multiple rows of processors are used, the mapping schemes just described ensure that neighboring domains are always allocated to neighboring processors.

To illustrate the mapping strategies employed in conjunction with 2-D block partitioning, consider again a cantilever beam to be analyzed using 16 processors. If a 4×4 array of processors is allocated (see Figure 3a), the beam is normally partitioned into a 4×4 array of blocks, and processors are assigned as indicated in Figure 3b. To utilize the maximum number of processors, some test cases are partitioned into $n \times 64$ blocks. For example, if a test case is discretized as an 8×64 element mesh, and 8×64 block partitioning is desirable in order to utilize all 512 processors. Since the DELTA is configured as a 16×32 processor array, the following modified 2-D mapping scheme becomes necessary. In this scheme, an $n \times m$ array of processors is allocated, but viewed as an $\frac{n}{2} \times 2m$ array. For example, if a 4×4 array of processors is allocated, the array of 16 processors is viewed as a 2×8 array with the numbering scheme shown in Figure 11a. The resulting processor assignments are shown in Figure 11b.

Global Communication

As already mentioned, two stages in the ADR algorithm require global sums, representing the addition of partial sums located on each processor. This operation may be easily and efficiently accomplished by using the high-level global-sum construct which the DELTA provides. This construct takes advantage of the two physical links between connected processors to overlap communication in two directions. For an $n \times m$ array of 2^p processors, its application yields the complete sum on each processor after only p communication steps.

The global sum process may be illustrated in reference to the 2×2 mesh shown in Figure 12. In the first step, partial sums are simultaneously exchanged between processors P_0 and P_1, and between processors P_2 and P_3. The second step is analogous to the first except now results are exchanged between processors P_0 and P_2, and between processors P_1 and P_3. Thus, after two communication steps the partial sums are present on all four processors.

4. NUMERICAL RESULTS

First, a description of the test cases used for evaluating performance. Next, an overview of the evaluation procedure is given. The primary factors which affect performance are then reviewed. Finally, parallel-processing results for the DELTA are discussed.

4.1 Test Cases

Thirteen test cases representing both straight and curved 2-D and 3-D geometries with elastic and hyperelastic materials are used to evaluate performance. They were developed and analyzed in reference [2] and are intended to represent some of the problems which occur in tire modeling and analysis. As such, they are designed to include contact, large deformations, and nonlinear hyperelastic materials. The key features of each test case are summarized in Table 1. Additional modeling details, including contact conditions, may be found in reference [2].

The straight test cases correspond to the 2-D plane stress and 3-D analysis of elastic and hyperelastic cantilever beams subjected to tip loading and frictionless contact with an inclined surface (see Figure 13). Two discretizations are considered for each problem - one with 1024 elements and another with 8192. The discretization is uniform such that all elements in a given mesh are the same size. Figure 14 shows the 8192 element test case of the 3-D hyperelastic beam in its final "steady-state" deformed configuration.

The curved test cases represent a 3-D hyperelastic circular arch, a 3-D elastic cylindrical thick shell or "tunnel," and a 3-D elastic and hyperelastic torus subjected to line load at the summit (see Figure 15). Similar to the straight test cases, two discretizations of the arch are considered consisting of 128 and 1024

8-node solid elements. Figure 16 shows the 1024 element circular arch in its deformed state. The tunnel is shown in Figures 17a and 17b in its undeformed and deformed configurations, respectively. The deformed configuration of the torus test case when loaded against a flat surface is shown in Figure 18.

4.2 Evaluation Procedure

The parallel performance of the ADR algorithm is evaluated (where possible) on all 512 processors of the DELTA. The elapsed time required to complete a fixed number of pseudo time steps or iterations for static analysis is determined for each test case. Relative speedup S and efficiency E are computed as

$$S = T_1/T_n \tag{4.1}$$

$$E = S/n * 100\% \tag{4.2}$$

where T_1 and T_n represent the elapsed time for a single processor and for n processors. Relative speedup and efficiency are both indicators of the extent to which unitary linear relative speedup is achieved, which implies relative speedups equal to the number of processors and an efficiency of 100 percent.

Correctness of the parallel algorithm is verified in two ways. The displacement results obtained for each test case after the specified number of time steps are compared to ensure that they are independent of the number of processors used. In addition, all test cases are executed to completion on the DELTA, and the final results are compared with those obtained in reference [2] using a single processor.

In preface to discussing parallel performance of the ADR algorithm, the interacting factors which govern this performance should be considered. A review of the primary factors is given next.

4.3 Performance Factors

Best performance (in terms of relative speedup) is achieved when the ratio of communication time to computation time is minimized, and when the computational load among the processors is balanced.

Computation time

The material and dimensionality of the elements affects computation time. Hyperelastic elements are more computationally demanding than elastic elements. For a given mesh, the hyperelastic test cases should lead to higher relative speedups than the elastic test cases due to increased computations relative to communication. Trilinear elements require more computations than bilinear elements, however they also require more communication since they have four nodes (as opposed to two) on each face.

The number of processors affects computation time. As the number of processors increases for a given problem, the number of elements (and therefore the computational load) per processor decreases. This decrease is mostly linear, although there is a small amount of computational cost associated with adding more processors, due to an increase in redundant computations. Since each processor executes the complete ADR algorithm, the kinematic variables (displacements, velocities, and accelerations) for the nodes on a given interface are computed by both processors sharing that interface. This redundant computational effort is increased as more processors (and therefore more interfaces) are added.

Communication Time

The time for nearest-neighbor communication is only a function of message length, which is defined for the ADR algorithm by the number of nodes on a given interface and the degrees of freedom associated with each node. For 1-D strip partitioning, the message length is mesh dependent and remains fixed regardless of the number of processors added. However, for 2-D block partitioning, the message length depends on both the mesh and the number of processors utilized. As described earlier, only two send-receive operations (in the case of 1-D strip partitioning) or four send-receive operations (in the case of 2-D block partitioning) are needed to implement all nearest-neighbor exchanges in each time step regardless of the mesh or number of processors. The time for global communication is a function of the number of processors. As mentioned previously, for an $n \times m$ array of 2^p processors, the number of associated messages is equal to p. These messages only involve scalar quantities, and the message lengths are independent of problem size.

Load Balance

Relative speedup may be reduced due to load imbalances in which one or more processors are waiting on others to finish. For the applications considered here, load balancing is primarily a function of the number of elements assigned to each processor as discussed in Section 2. However, some degree of load imbalance may be introduced from the following two sources. Processors with elements which form the boundaries of the given mesh in the XY plane (elements along the left and right ends or top and bottom surfaces of the cantilever beams, for example) have fewer neighboring processors, and therefore perform less of the nearest-neighbor communication process. In addition, the contact enforcement algorithm represents additional computations for some processors.

4.4 DELTA Results

Parallel performance results for the DELTA are presented in Table 2, and plots of relative speedup versus number of processors are shown for each test case in Figure 19. For each test case, results were obtained using an increasing number of processors (i.e., a single processor, 16, 32, 64, 128, 256, and 512 processors)

to obtain the data in Figure 19. Two-dimensional block partitioning was used in all cases, therefore the total processor numbers given are expressed in the $n \times m$ format, where n denotes the number of rows of processors and m denotes the number of processors per row. The relative speedups shown are based on a total of 512 processors with five exceptions. The small 3-D arch test case is limited to 64 processors when 2-D block partitioning is used, since it represents a 2×32 element mesh in the XY plane. Likewise, the small 3-D beam test cases, the large 3-D arch and tunnel test cases represent 4×64 element meshes in the XY plane and are therefore limited to 256 processors. The message lengths shown in Table 2 refer to the total number of internal force values each processor must exchange for nearest-neighbor communication each time step. The number of time steps or iterations executed is also indicated. Some general trends are now discussed.

The small 2-D beam test cases and the 3-D 128-element arch test case exhibit the lowest relative speedups. These test cases have a small number of elements and therefore do not entail significant computational effort. As a result, even though the message lengths are short, the overall communication-to-computation ratios are relatively high. The relative speedup achieved for the arch test case using 64 processors represents a parallel-processing efficiency of 39 percent. That is, the relative speedup is 39 percent of the theoretical or linear-relative-speedup limit of 64. The beam test cases exhibit an average efficiency of 50 percent for 64 processors, 18 percent for 256 processors, and 9 percent for 512 processors. Accordingly, scalability beyond 64 processors is very low for these test cases, as reflected by the slope of the relative speedup curves in Figure 17.

Much higher relative speedups are exhibited by the large 2-D beam test cases, as well as by the small 3-D beam test cases and the 3-D 1024-element arch test case. The communication-to-computation ratio for the large 2-D beam test cases is very low due to the large number of elements and relatively small message lengths. The small 3-D beam test cases have the same number of elements as the small 2-D beam test cases, but the messages are now much longer. Even so, these test cases achieve higher relative speedups due to increased computations associated with the trilinear element. The small hyperelastic 3-D beam test case has the same material, message length, and number of elements as the 1024-element arch test case and exhibits equivalent performance. The relative speedups achieved for all five of these test cases represent an average efficiency of 53 percent for 256 processors. For 512 processors, the large 2-D beam test cases exhibit an average efficiency of 43 percent. These results indicate moderate scalability through 512 processors as shown in Figure 17. The potential exists for even higher relative speedups beyond 512 processors, although the parallel processing efficiency measure may become small in this range.

The highest relative speedups are exhibited by the large 3-D beam, tunnel, and torus test cases. These test cases have the lowest communication-to-computation ratios because of the large number of elements and the increased computational cost associated with the trilinear element. The two cantilever beam test cases achieve the highest relative speedups within this group since they have the shortest message lengths. The relative speedups achieved for all five of these test cases represent an average efficiency of 84 percent for 256 processors. For 512 processors, the beam and torus test cases exhibit an average efficiency of 75 percent. Thus, the scalability of the ADR algorithm for these test cases remains high through 512 processors (see Figure 17), and efficient execution on many more processors (leading to much higher relative speedups) should be possible.

As expected, the hyperelastic versions of each test case exhibit higher relative speedups than their elastic counterparts. The hyperelastic test cases require more computations due to material, and as such achieve better performance. The effect of the partitioning method on performance has been investigated. Relative speedup results for several different partition or processor arrangements are presented in Table 3 for the large 3-D elastic beam. As shown, the effect is minimal (less than three percent).

All thirteen test cases were run to completion on the DELTA in order to demonstrate correctness of the parallel implementation of the ADR algorithm. The nonlinear static analysis results are given in Table 4. As shown, each test case was analyzed using the maximum number of processors allowed with the present 2-D block partitioning scheme. The number of time steps (or iterations) is given, as is the elapsed time to completion. The Y values are maximum vertical displacements for the free end of the beam and for the summit of the arch, tunnel, and torus. The X values are the corresponding horizontal displacements for the free end of the beam. The results shown are consistent with those of the single-processor implementation [2]. For some of the test cases involving contact, differences exist in the number of iterations and small differences exist in the final displacements when compared with the results given in reference [2]. This is because all aspects of the final contact formulations given in reference [1] were not in place at the time the results in reference [2] were generated.

The completion times for all of the test cases are less than one hour and demonstrate both the computing power of the DELTA and the ability of the ADR algorithm to exploit this capability fully. As mentioned earlier, these results were achieved using unvectorized, baseline code. As shown in reference [2], a vectorized version could further increase the speed of execution by a factor of the O(5), assuming that the problem size is such that each processor has enough elements to achieve efficient vectorization.

5. CONCLUSIONS

The overall goal of this research is to develop efficient single-processor and multiprocessor implementations of the ADR algorithm and evaluate their performance for the static analysis of nonlinear, hyperelastic systems involving frictionless contact. For problems of this nature, the ADR algorithm may represent one of the best approaches for parallel processing. Performance evaluations on single-processor computers have shown that the ADR algorithm is reliable and highly vectorizable, and that it is competitive with direct solution methods for the highly nonlinear problems considered. In contrast to direct solution methods, it has minimal memory requirements, is easily parallelizable, and is scalable to more processors. It also avoids the ill-conditioning related convergence problems of other iterative methods for nonlinear problems. The objective of the present paper is to evaluate the performance of a massively parallel implementation of the ADR algorithm.

A parallel ADR algorithm is developed for nonlinear structural analysis and implemented on the 512-processor Intel Touchstone DELTA system. It is based on the sequential code described in reference [2] and is designed such that each processor executes the complete sequential algorithm on a subset of elements. One-dimensional strip partitioning and two-dimensional block partitioning are used to divide the problem domain among the available processors. Load balancing is ensured by the use of structured, uni-material meshes. Efficient schemes are developed to accomplish the required nearest-neighbor and global communication. The parallel algorithm is used to solve for the nonlinear static response of 2-D and 3-D cantilever beam problems and 3-D arch, tunnel, and torus problems.

Correctness of the parallel algorithm is verified by running all test cases to completion on the DELTA. Final results are consistent with those obtained using a single-processor. Completion times for the large 3-D test cases are minimal and demonstrate both the computing power of the DELTA and the ability of the ADR algorithm to fully exploit this power. Moreover, the current multiprocessor implementation is not vectorized. A vectorized version should lead to further increases in performance. The minimal memory requirements of this method are again demonstrated as the largest test case runs successfully on a single DELTA processor equipped with 16 megabytes of memory. Relative speedups are based on a fixed number of time steps, during which contact does not occur. However, the contact algorithm used in this study is very simple and efficient [1]. As such, its effect on performance would most likely be negligible since the additional computations it represents are trivial.

Impressive relative speedups are achieved using the DELTA, especially for the large 3-D test cases. This performance may be attributed to the minimal interprocessor communication required by the ADR algorithm relative to computations and the efficient schemes with which this communication is accomplished. These relative speedup results demonstrate the high scalability of the ADR algorithm and show that the algorithm

can be implemented on at least 512 processors without significant performance degradations. Thus, the ADR algorithm provides the potential for efficiently exploiting large numbers of processors to substantially reduce the solution time of highly nonlinear problems. In this context, it represents a very promising approach for parallel-vector processing.

Acknowledgement

This research was performed in part using the Intel Touchstone DELTA System operated by Caltech on behalf of the Concurrent Supercomputing Consortium. Access to this facility was provided through NASA Grant NAG-1-1505 and Mr. Ronnie E. Gillian was the technical monitor. The authors gratefully acknowledge this support. In addition, the first author gratefully acknowledges the support provided by the Dean's Scholar Program at Clemson University.

REFERENCES

1. D. Oakley, N. Knight [1994]. Adaptive Dynamic Relaxation Algorithm for Nonlinear Hyperelastic Structures - Part I. Formulation, (in review).

2. D. Oakley, N. Knight [1994]. Adaptive Dynamic Relaxation Algorithm for Nonlinear Hyperelastic Structures - Part II. Single-Processor Implementation, (in review).

3. D. Oakley, N. Knight, D. Warner [1994]. Adaptive Dynamic Relaxation Algorithm for Nonlinear Hyperelastic Structures - Part III. Parallel Implementation, (in review).

4. P. Sadayappan, F. Ercal [1987]. Nearest-Neighbor Mapping of Finite Element Graphs onto Processor Meshes, *IEEE Transactions on Computers*, Vol. 36, pp. 1408-1424.

5. G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, D. Walker [1988]. *Solving Problems on Concurrent Processors*, Volume 1, Prentice Hall, New Jersey.

6. P. Underwood [1983]. Dynamic Relaxation, *Computational Methods for Transient Dynamic Analysis*, T. Belytschko and T.J.R. Hughes, editors. North Holland, Amsterdam, pp. 246-265.

7. M. Papadrakakis [1981]. Post-Buckling Analysis of Spatial Structures by Vector Iteration Methods, *Computers and Structures*, Vol. 14, pp. 393-402.

8. M. Papadrakakis [1982]. A Family of Methods With Three-Term Recursion Formulae, *International Journal for Numerical Methods in Engineering*, Vol. 18, pp. 1785-1799.

9. C. Aykanat, F. Ozguner, F. Ercal, P. Sadayappan [1988]. Iterative Algorithms for Solution of Large Sparse Systems of Linear Equations on Hypercubes, *IEEE Transactions on Computers*, Vol. 37, pp. 1554-1568.

10. V. Sunderam [1990]. PVM: A Framework for Parallel Distributed Computing, Department of Mathematics and Computer Science, Emory University, Atlanta, GA.

Table 1. Test Case Characteristics

test case	total elems	total dof	mesh	mat	contact	instab
2D beam	1024	2304	128 × 8	E	Y	
				H	Y	
	8192	17408	512 × 16	E	Y	
				H	Y	
3D beam	1024	4800	64 × 4 × 4	E	Y	
				H	Y	
	8192	31104	128 × 8 × 8	E	Y	
				H	Y	
3D arch	128	855	32 × 2 × 2	H		Y
	1024	4775	64 × 4 × 4	H		Y
3D tunnel	8192	31515	64 × 4 × 32	E		Y
3D torus	8192	27744	32 × 16 × 16	E	Y	Y
				H	Y	Y

E = elastic, Hookean material

H = hyperelastic

Table 2. Parallel Performance Results Summary

model	total elems	mat	message lengths	time steps	no. of procs	speedup S	efficiency E
2D beam	1024	E	20	400	8 × 64	43	8
		H	20	400	8 × 64	50	10
	8192	E	48	200	8 × 64	210	41
		H	48	200	8 × 64	228	45
3D beam	1024	E	120	200	4 × 64	119	46
		H	120	200	4 × 64	127	50
	8192	E	270	100	8 × 64	382	75
		H	270	100	8 × 64	391	76
3D arch	128	H	72	400	2 × 32	25	39
	1024	H	120	200	4 × 64	127	50
3D tunnel	8192	E	792	100	4 × 64	211	82
3D torus	8192	E	408	100	16 × 32	381	74
		H	408	100	16 × 32	389	76

Table 3. Relative Speedup versus Partitioning

model	total elems	mat	message lengths	time steps	no. of procs	speedup
3D beam	8192	E	486	100	1 × 128	116.48
			432	100	2 × 64	117.05
			432	100	4 × 32	116.68
			594	100	8 × 16	114.23

Table 4. Nonlinear Static Analysis Results

model	total elems	mat	procs	ADR steps	Time (secs)	Y_{max} (mm)	X_{max} (mm)
2D beam	1024	E	8 × 64	5258	64	43.26	14.79
		H	8 × 64	5755	66	40.57	12.84
	8192	E	8 × 64	19854	443	43.26	14.77
		H	8 × 64	24291	558	40.64	12.85
3D beam	1024	E	4 × 64	2958	66	42.97	14.63
		H	4 × 64	36069	883	40.41	12.76
	8192	E	8 × 64	5805	285	43.28	14.80
		H	8 × 64	57281	3481	40.76	12.97
3D arch	128	H	2 × 32	14492	199	18.59	—
	1024	H	4 × 64	22199	471	18.87	—
3D tunnel	8192	E	4 × 64	3782	328	17.24	—
3D torus	8192	E	16 × 32	1824	90	80.23	—
		H	16 × 32	17651	1039	13.04	—

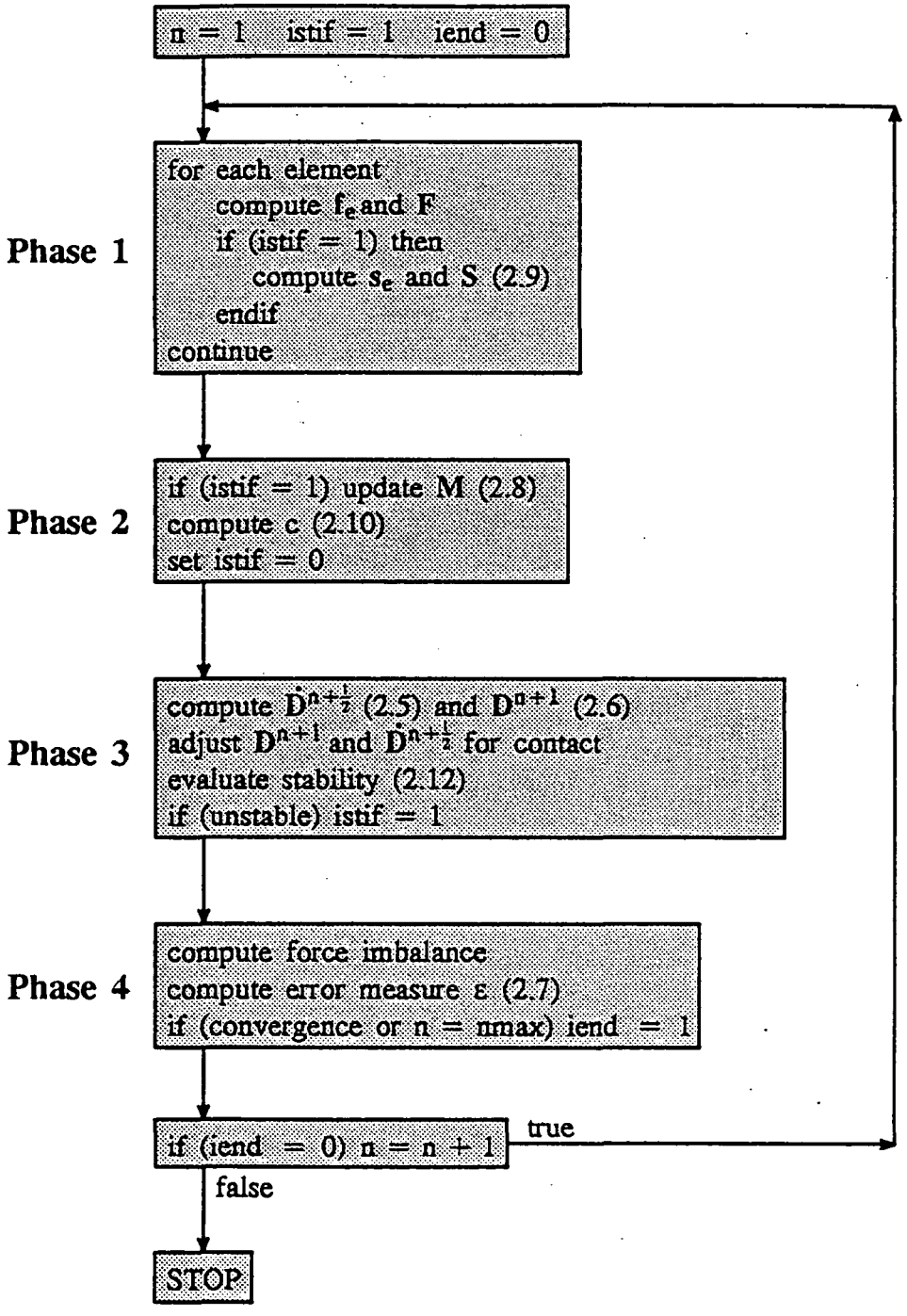
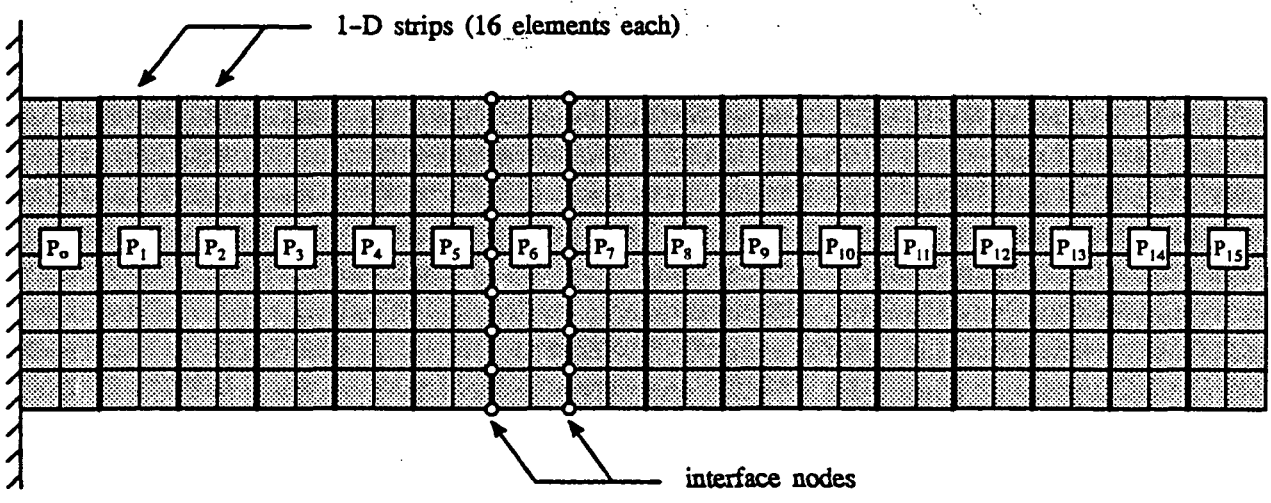
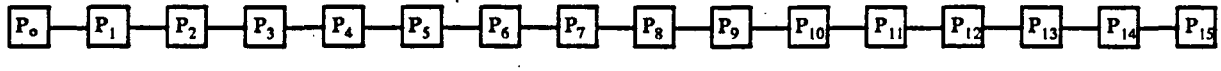


Figure 1. Sequential ADR Algorithm. Numbers in parentheses refer to equation numbers in the text.

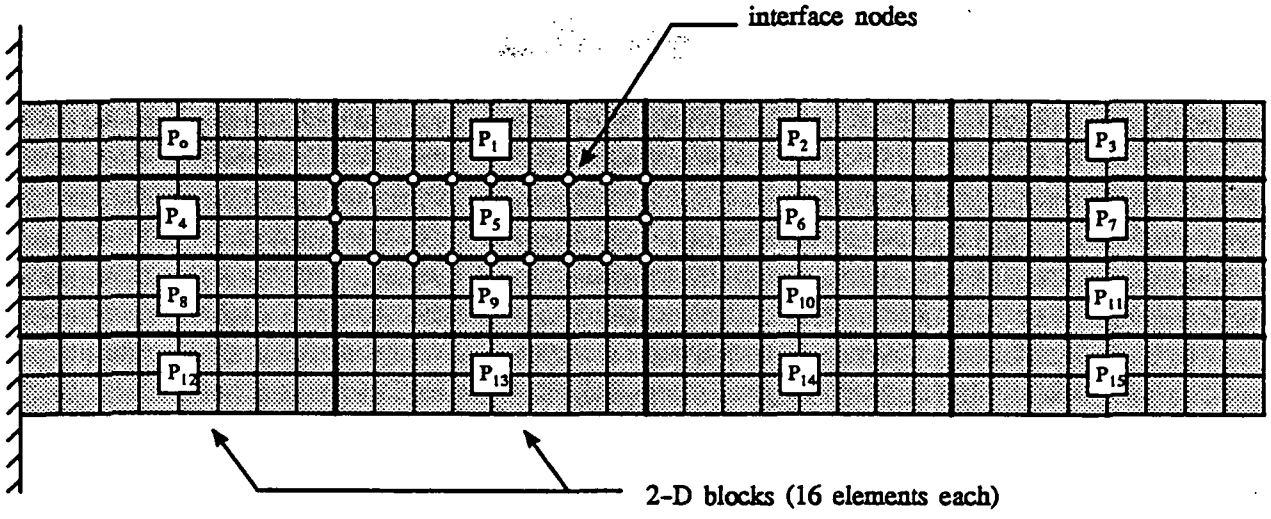


(a) 1-D Strip Partitioning Example for Cantilever Beam

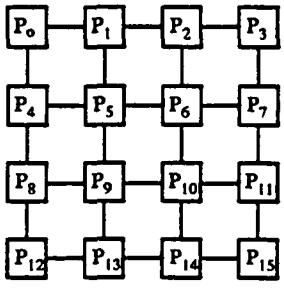


(b) 1-D Array of Processors

Figure 2. Illustration of 1-D Strip Partitioning



(a) 2-D Block Partitioning Example for Cantilever Beam



(b) 4 × 4 Array of Processors

Figure 3. Illustration of 2-D Block Partitioning

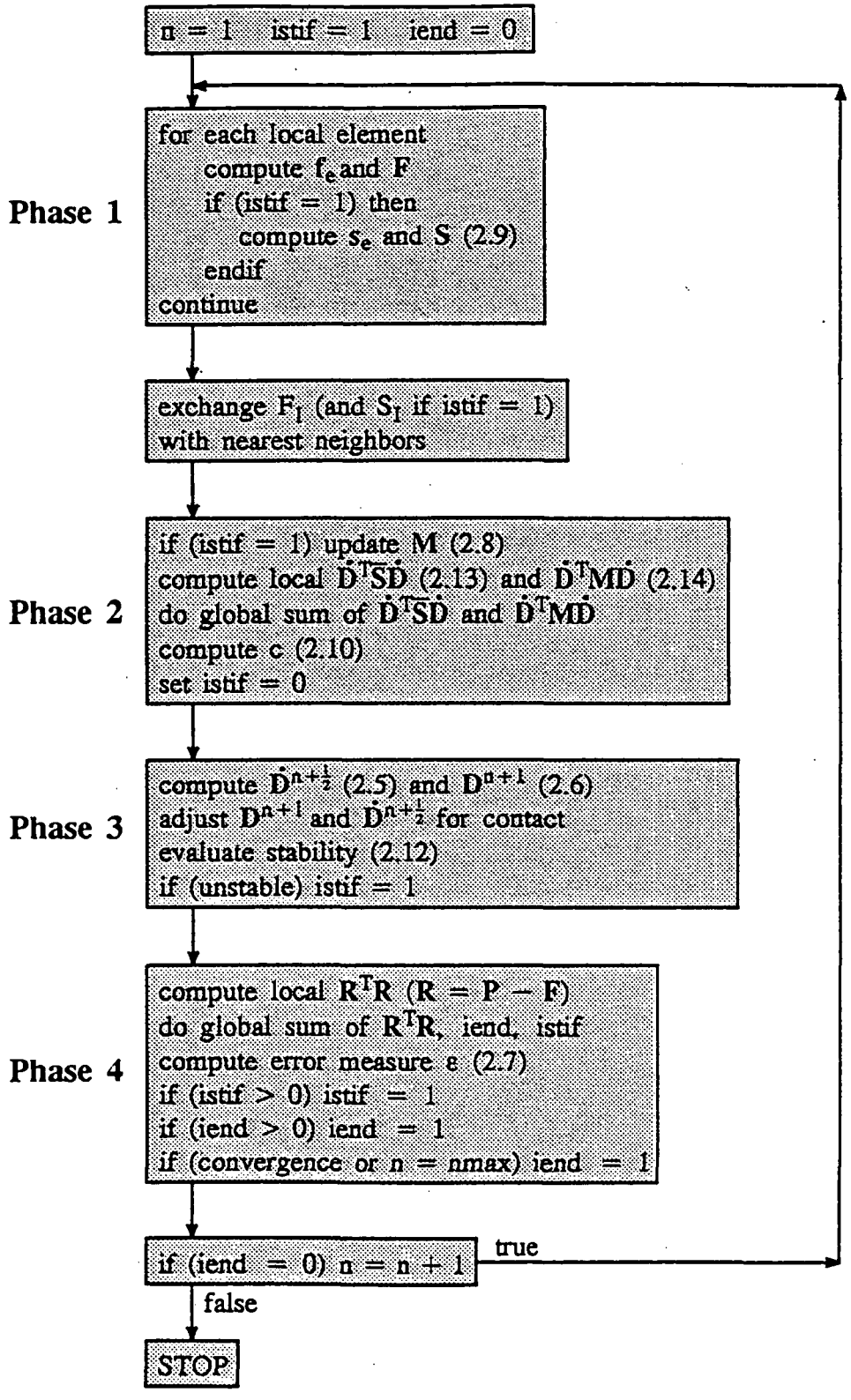
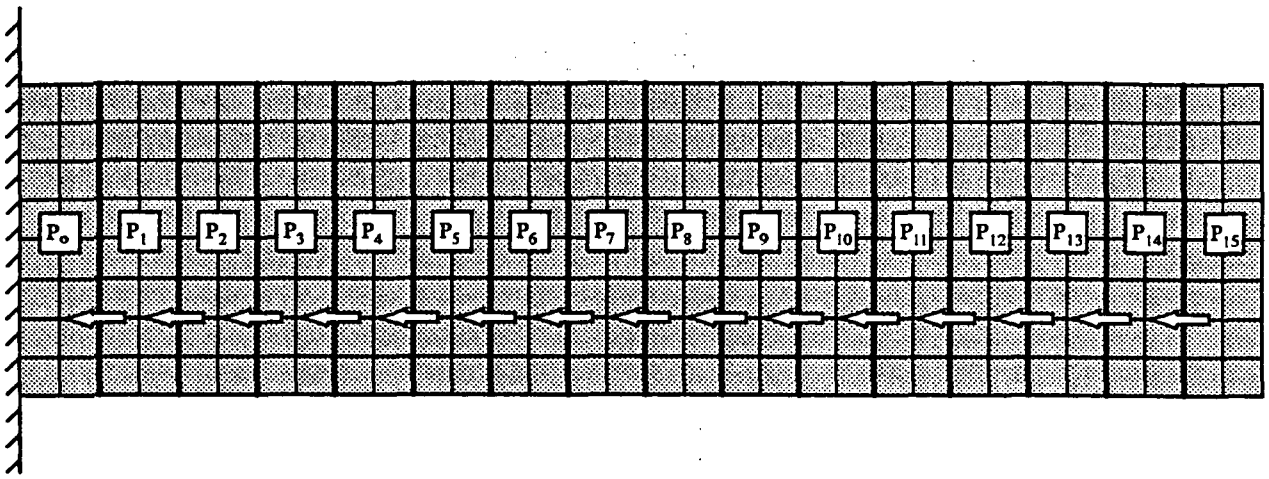
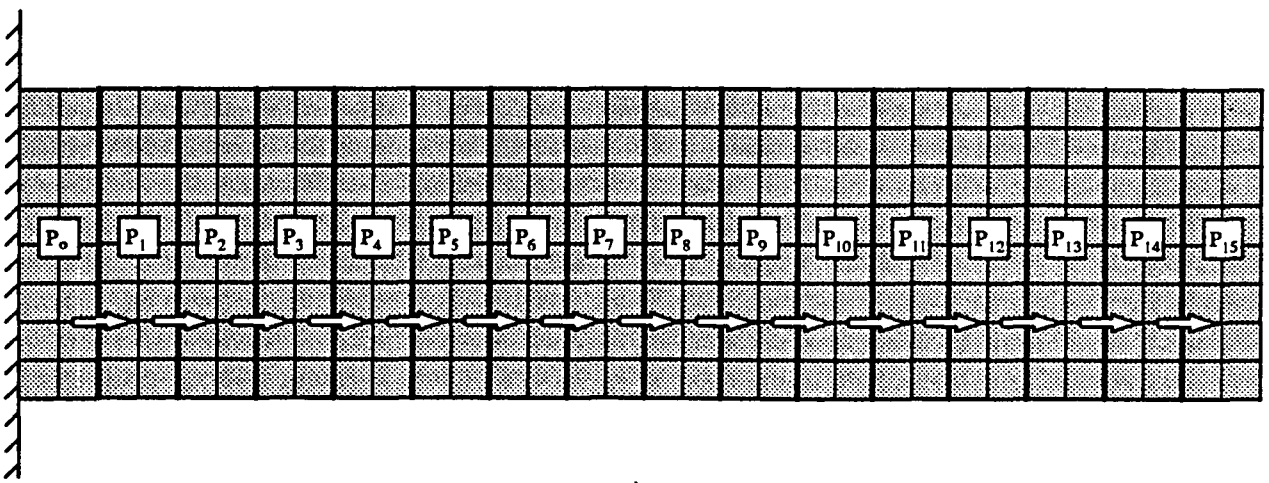


Figure 4. Parallel ADR Algorithm. Numbers in parentheses refer to equation numbers in the text.

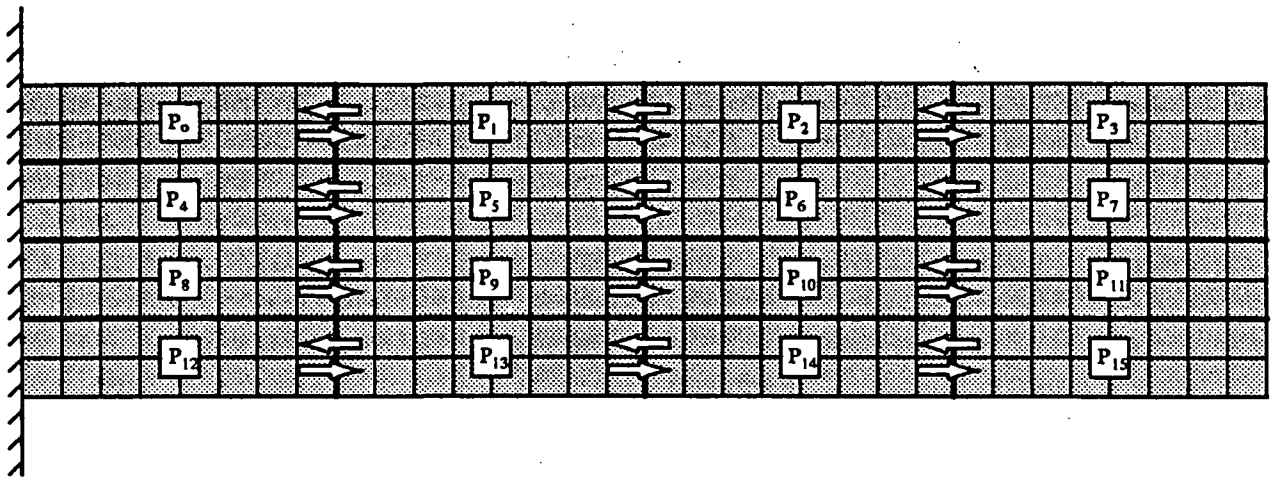


(a) Left Interface Values Sent to Left Neighbors (Step 1)

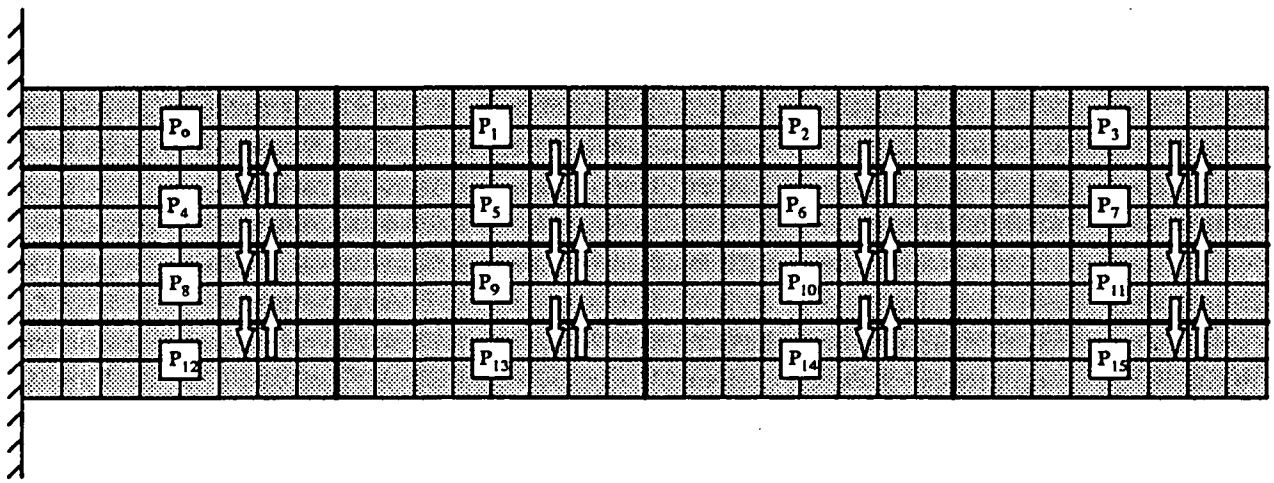


(b) Right Interface Values Sent to Right Neighbors (Step 2)

Figure 5. Nearest-Neighbor Communication for 1-D Strip-Partitioned Domains



(a) Horizontal Exchange of Interface Values (Steps 1 and 2)



(b) Vertical Exchange of Interface Values (Steps 3 and 4)

Figure 6. Nearest-Neighbor Communication for 2-D Block-Partitioned Domains

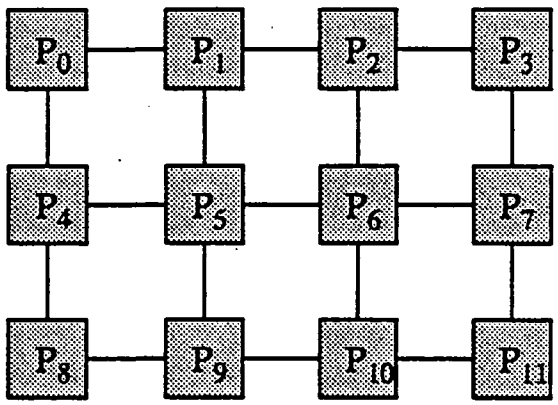


Figure 7. 3 × 4 Processor Example of DELTA Mesh Topology

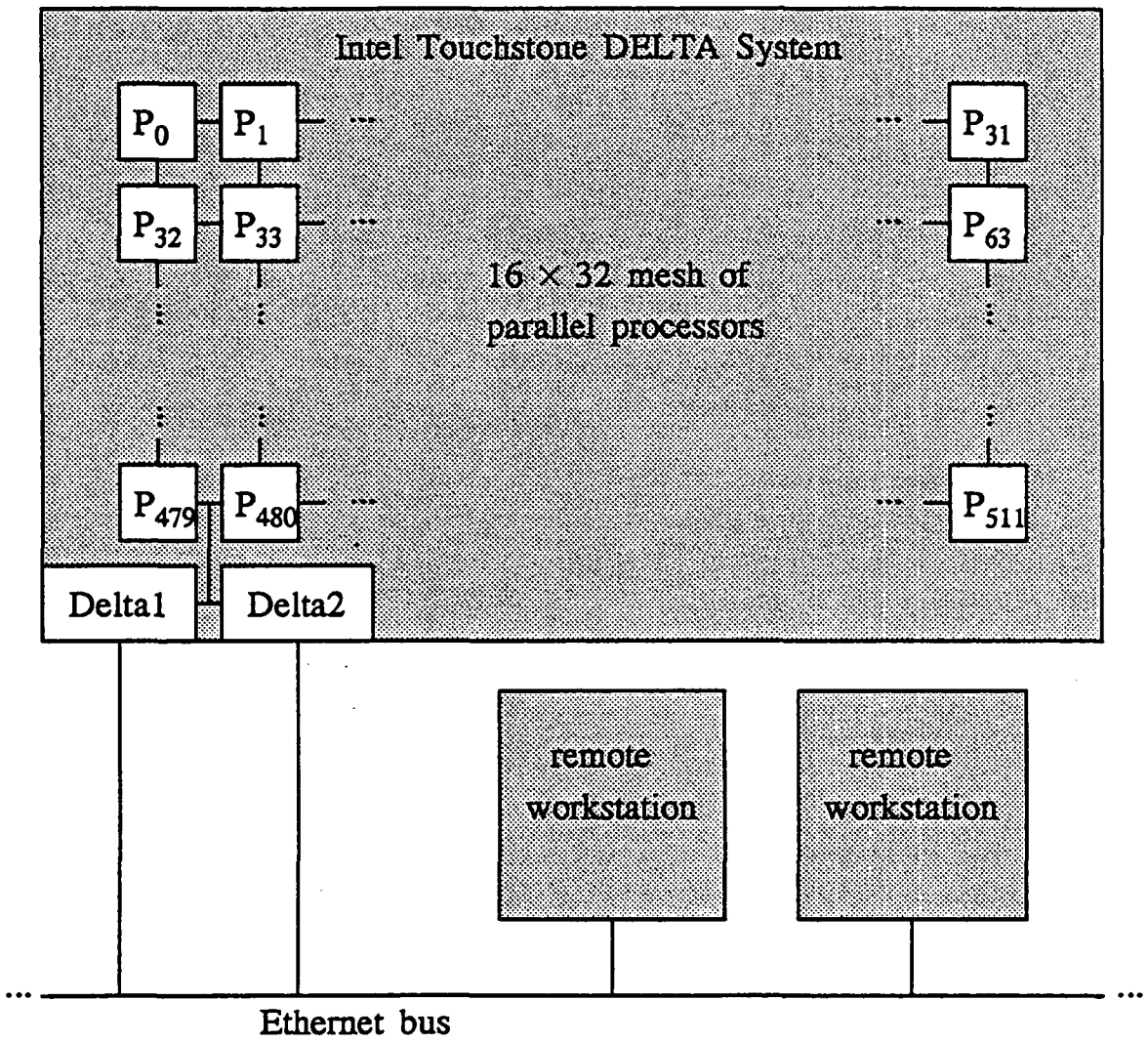


Figure 8. Intel Touchstone DELTA System Diagram

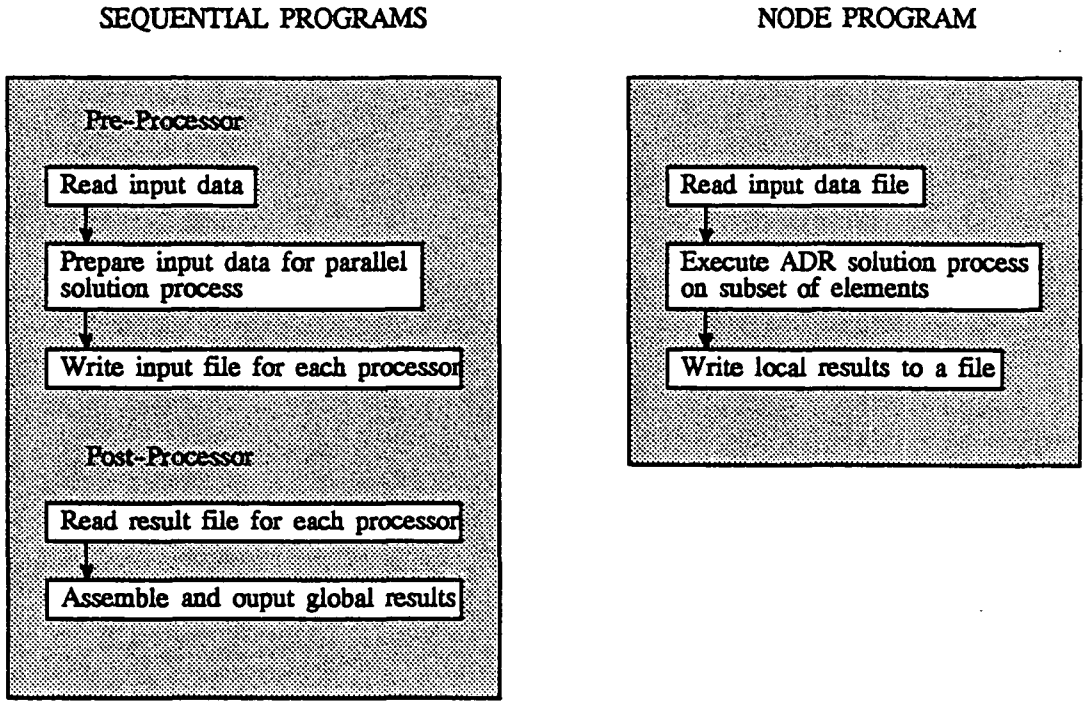
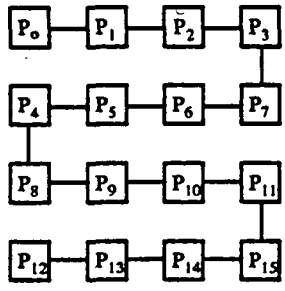
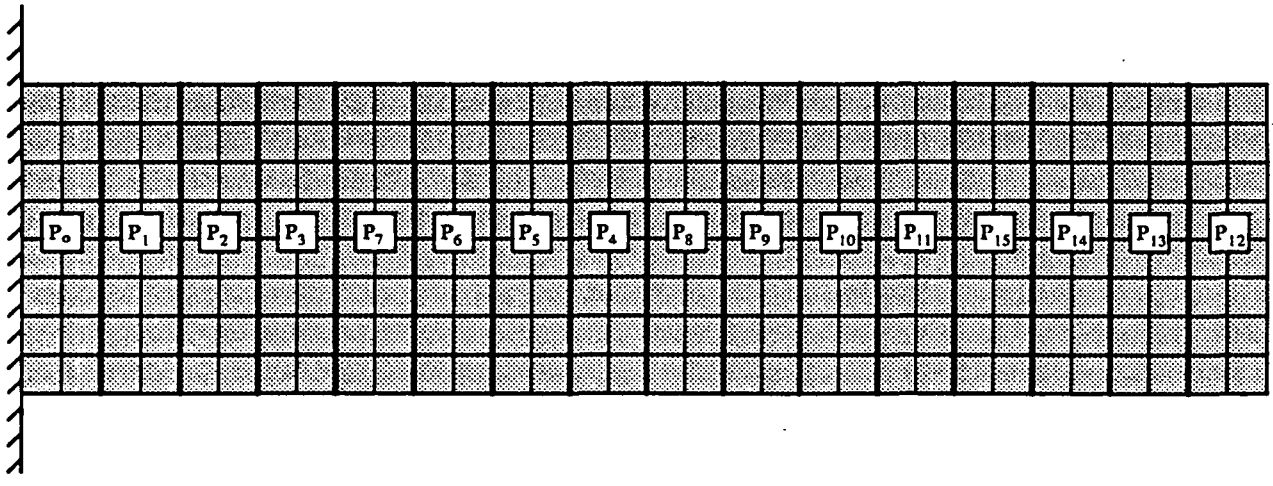


Figure 9. General Parallel Algorithm Flowchart

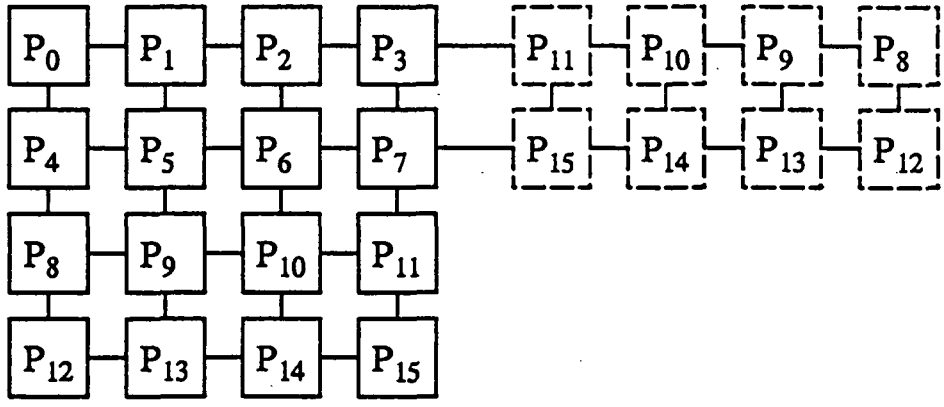


(a) 1-D View of 4 × 4 Processor Array

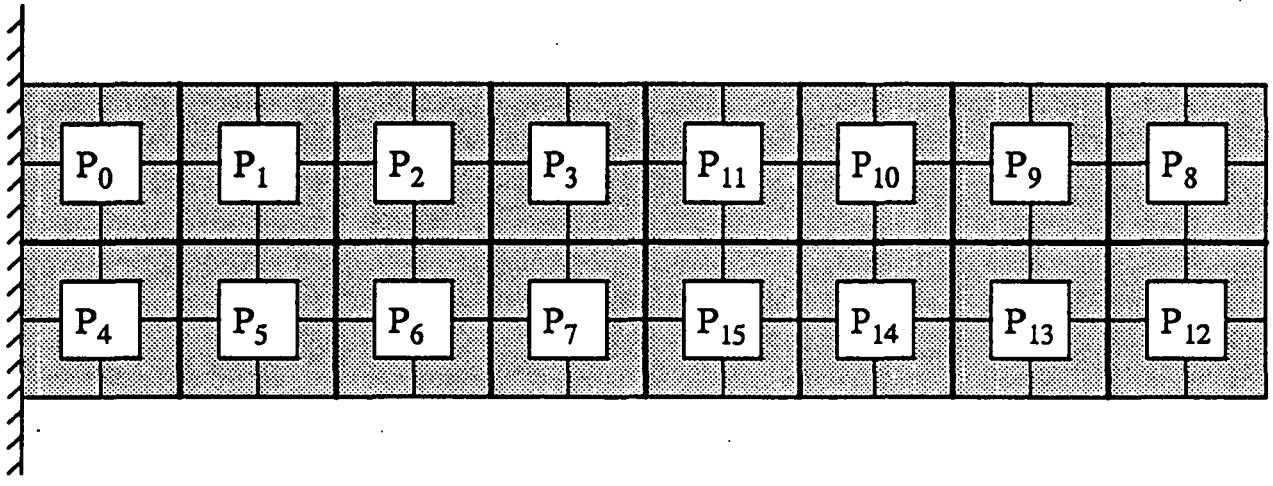


(b) Corresponding Processor Assignments for Cantilever Beam

Figure 10. Example of 1-D Strip Partitioning Using a 2-D Processor Array



(a) 2 × 8 View of 4 × 4 Processor Array



(b) Corresponding Processor Assignments for Cantilever Beam

Figure 11. Example of $\frac{n}{2} \times 2m$ Partitioning Using an $n \times m$ Processor Array

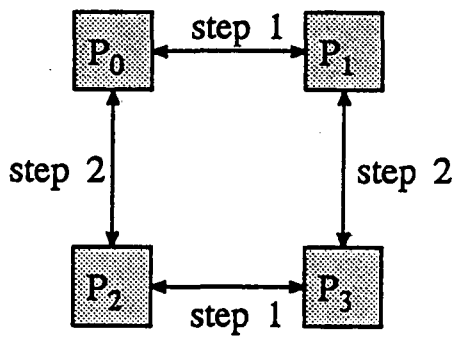
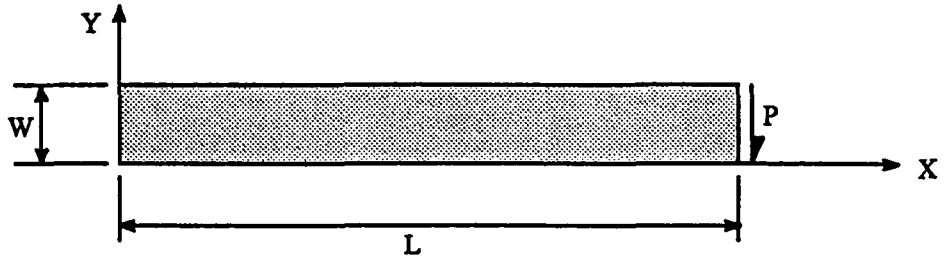
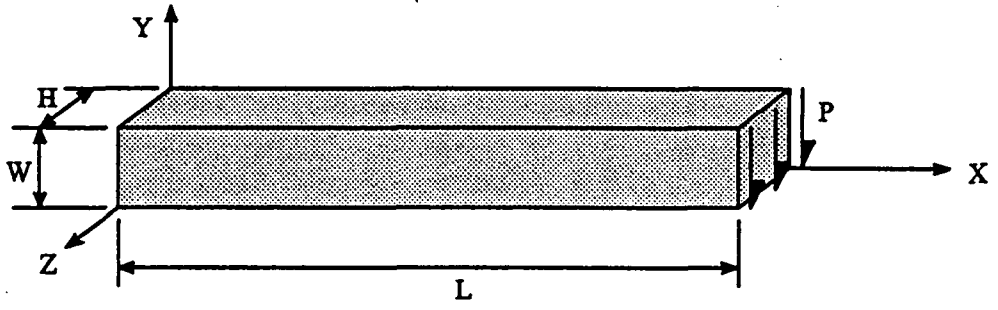


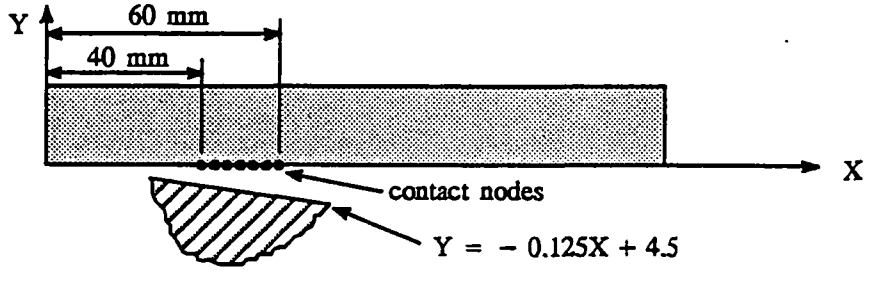
Figure 12. Global Communication Illustration for 2 × 2 Processor Array



(a) 2-D Geometry (Dimensions: L × W)



(b) 3-D Geometry (Dimensions: L × W × H)



(c) Contact Problem

Figure 13. Straight Test Cases

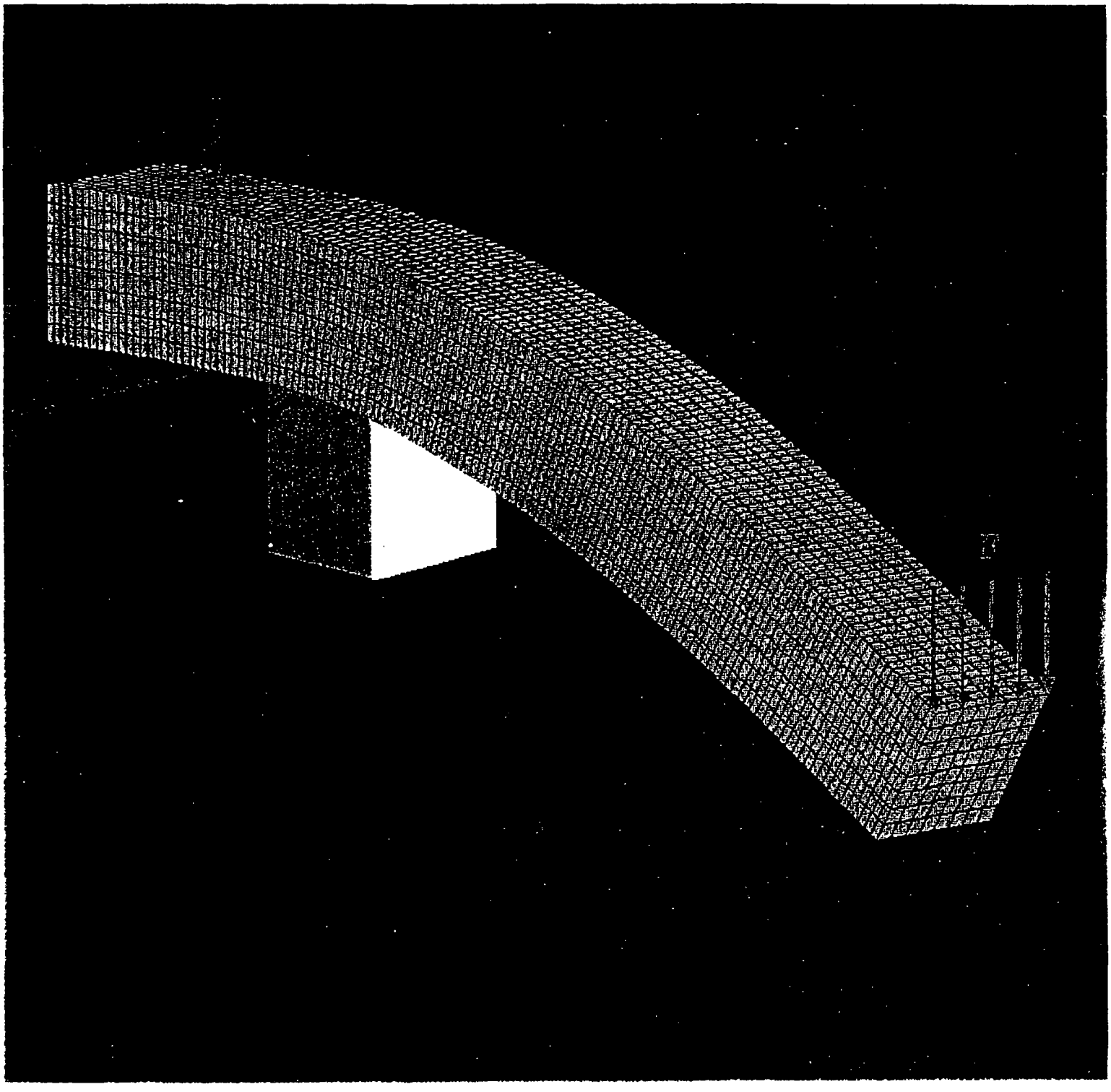
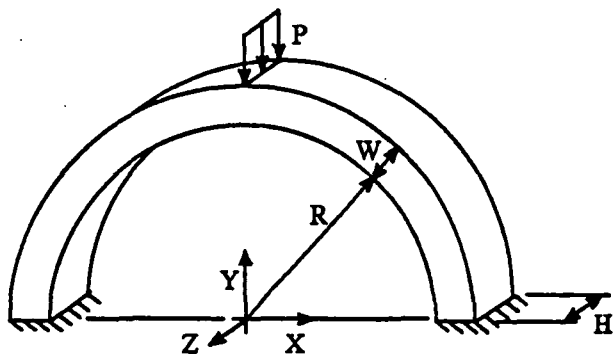
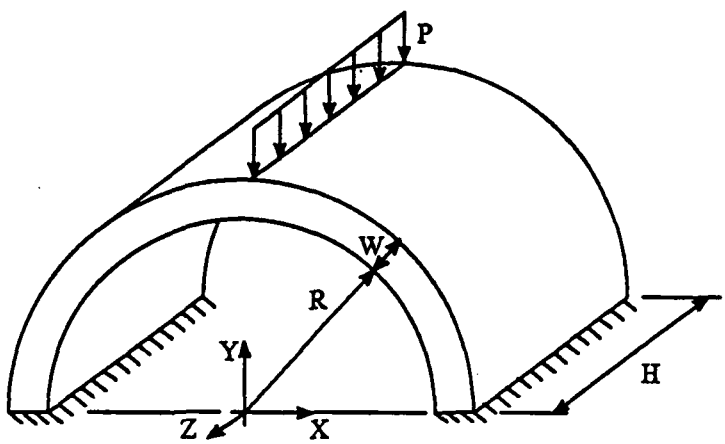


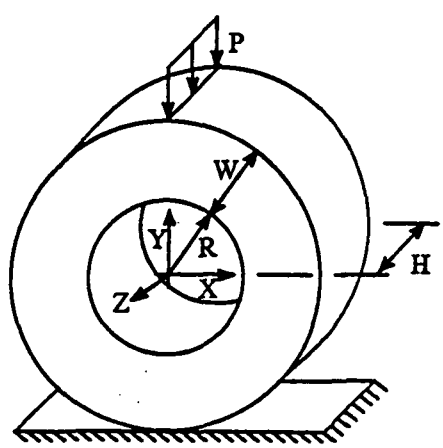
Figure 14. Deformed Configuration of the Large 3-D Cantilever Beam Test Case



(a) 3-D Circular Arch



(b) 3-D Cylindrical ThickShell or Tunnel



(c) 3-D Circular Torus with Rectangular Cross-Section

Figure 15. Curved Test Cases

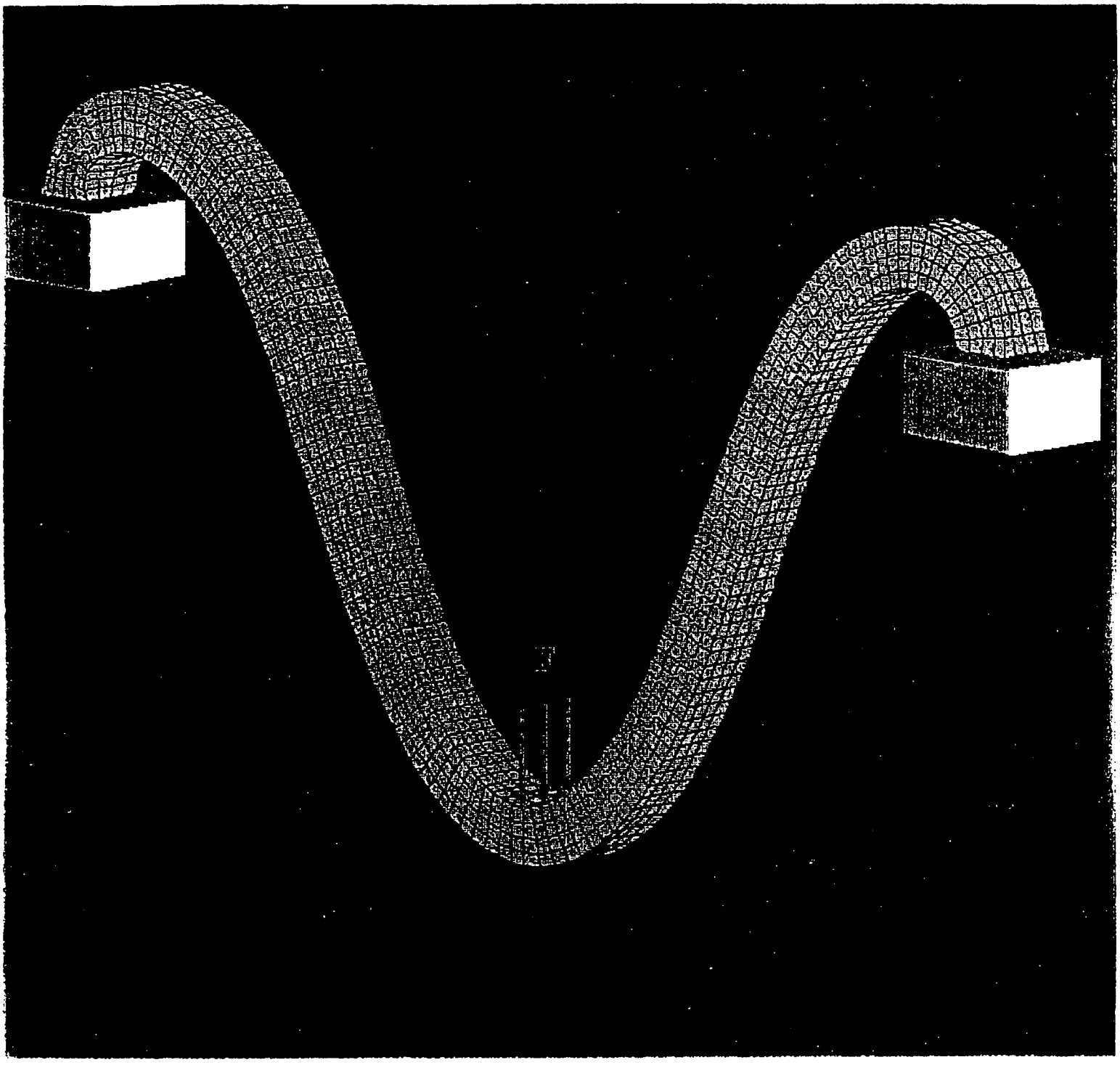


Figure 16. Deformed Configuration of the 3-D Circular Arch Test Case

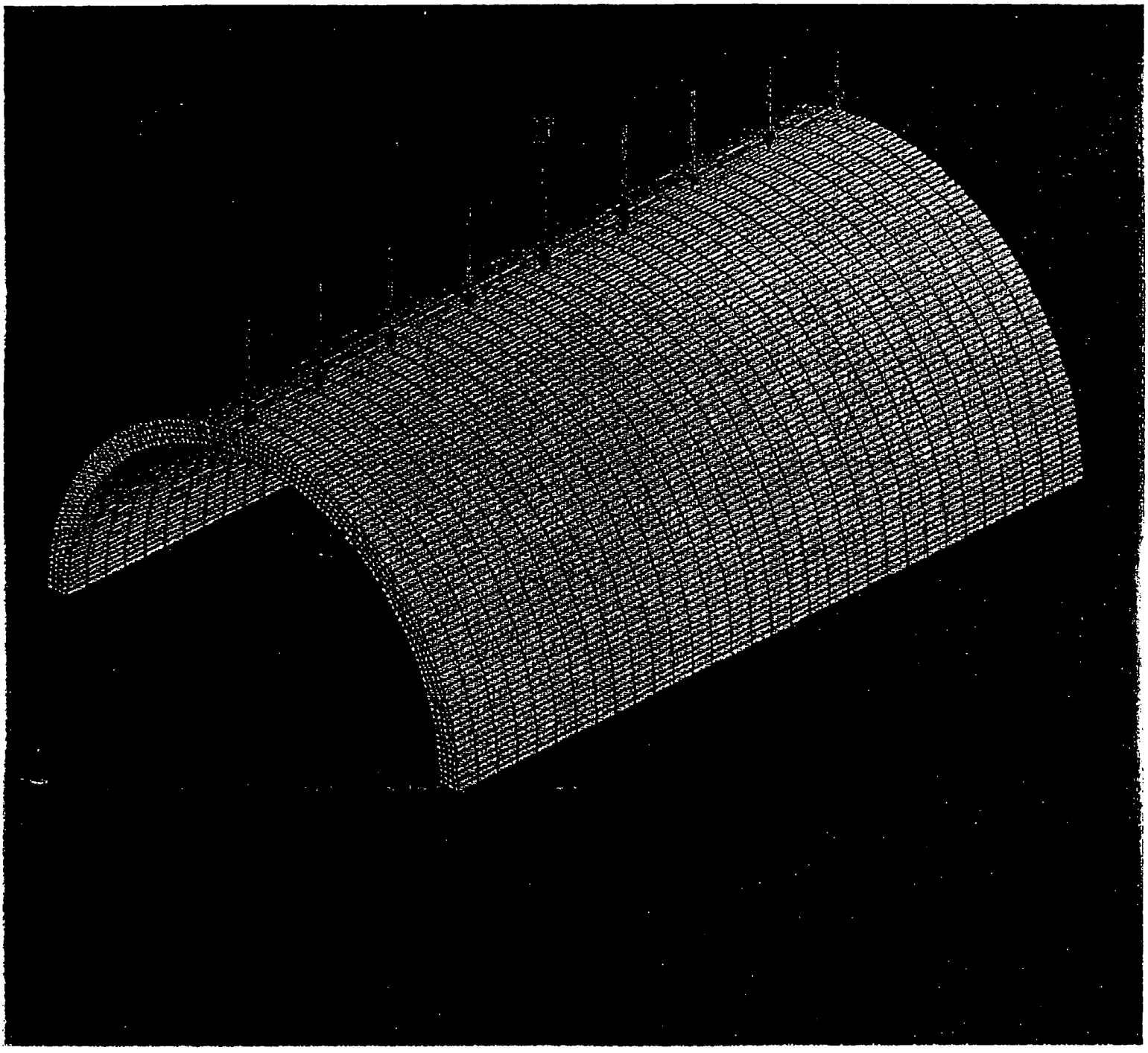


Figure 17a. Undeformed Configuration of the 3-D Tunnel Test Case

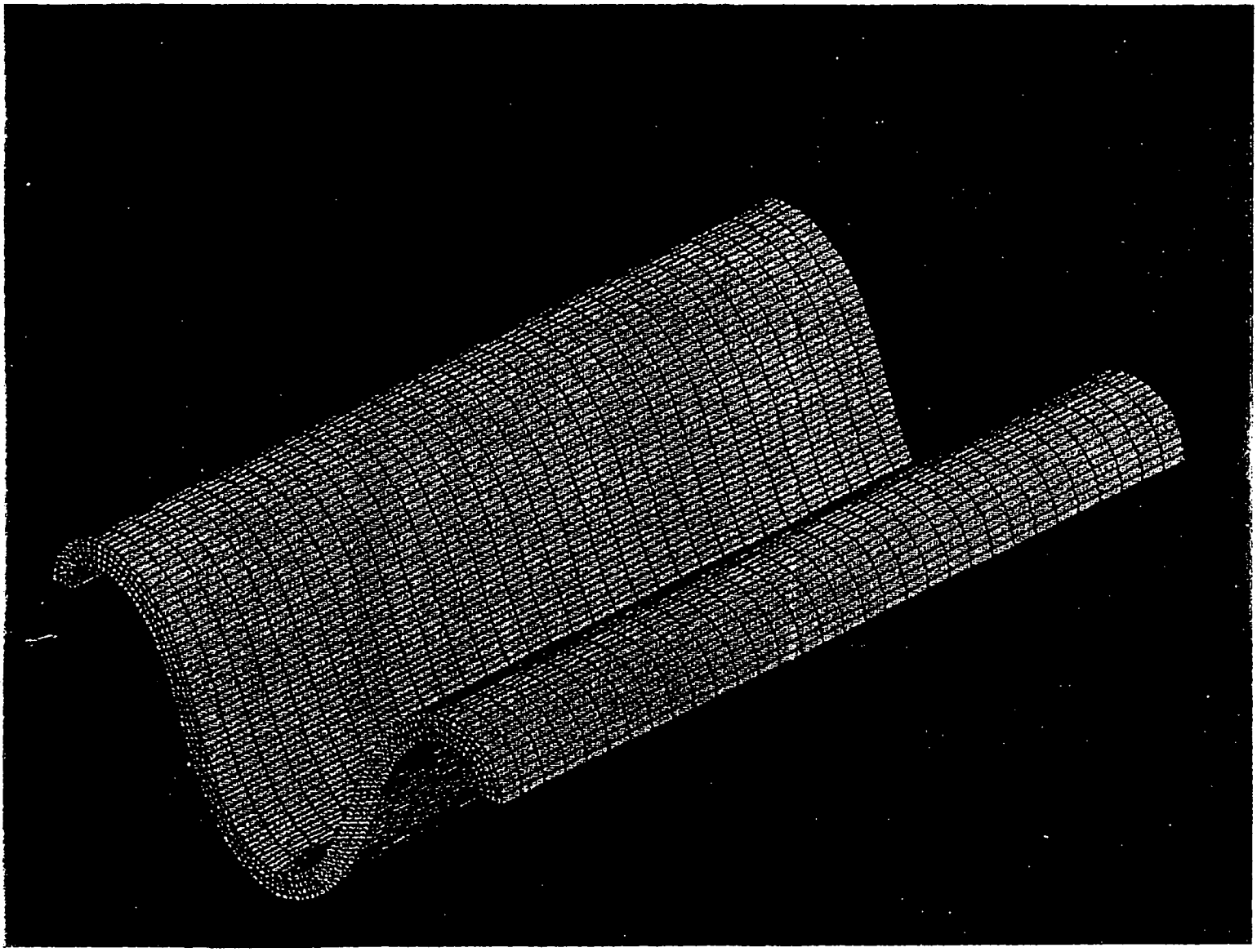


Figure 17b. Deformed Configuration of the 3-D Tunnel Test Case

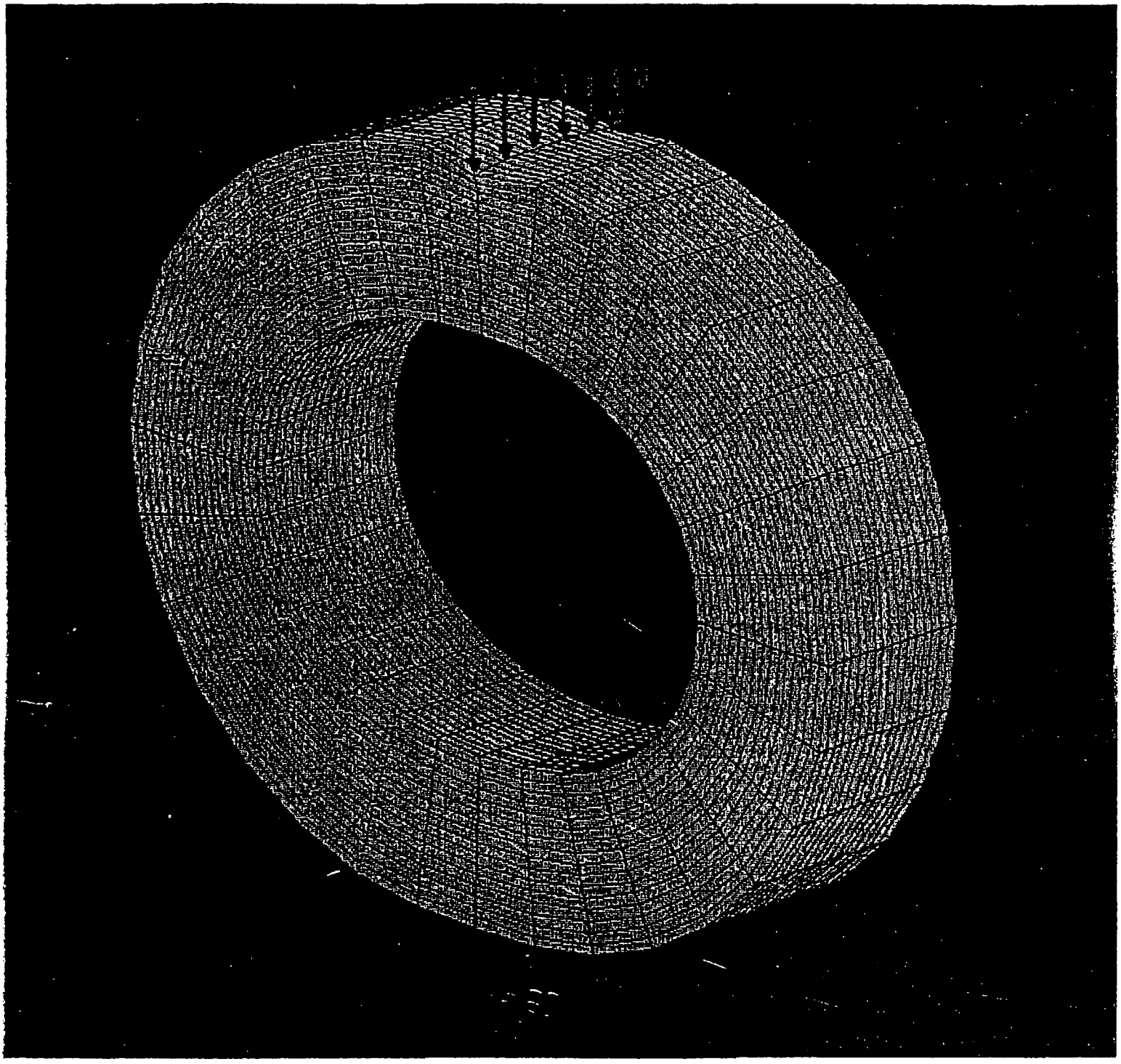


Figure 18. Deformed Configuration of the 3-D Torus Test Case

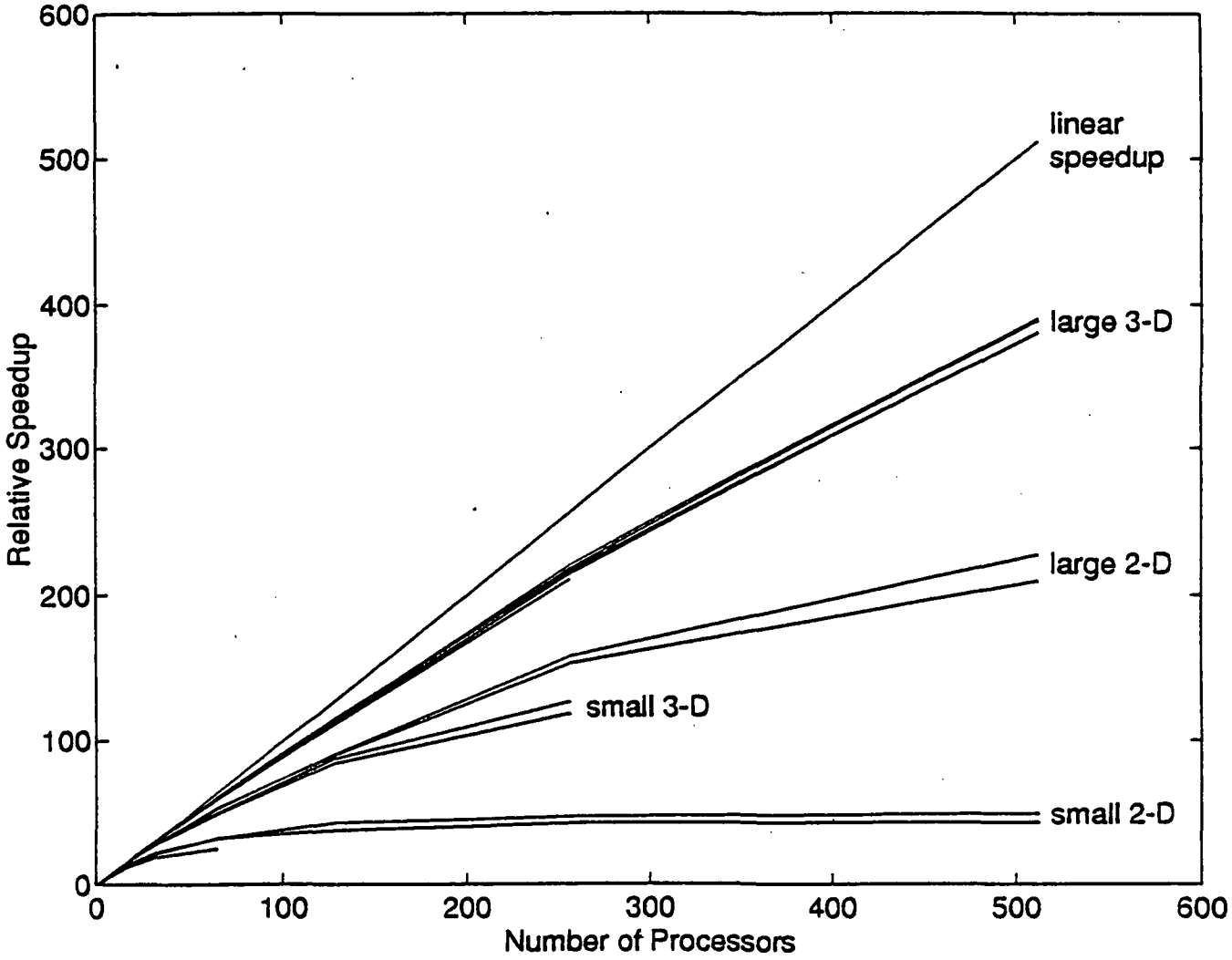


Figure 19. Relative Speedup versus Number of DELTA Processors