P-152

PARALLEL PROCESSORS AND NONLINEAR STRUCTURAL DYNAMICS ALGORITHMS AND SOFTWARE

Principal Investigator: Ted Belytschko

Department of Civil Engineering
Northwestern University
Evanston, Illinois 60208-3109

Final Technical Report

March 1, 1986 though May 31, 1990

NASA Research Grant NAG-1-650

PREFACE

This research was conducted under the direction of Professor Ted Belytschko. Participating research assistants were Bruce E. Engelmann, Noreen D. Gilbertsen, Mark O. Neal, and Edward J. Plaskacz. The help of Argonne National Laboratory, particularly Dr. James M. Kennedy who provided access to several parallel computing machines, is also appreciated.

The following presentations supported by NASA-Langley were made in 1989:

Martin R. Ramirez and Ted Belytschko, "Expert System and Error Estimates for Time Steps in Structural Dynamics", Texas Institute for Computational Mechanics Workshop on Reliability in Computational Mechanics, Austin, TX, October 27, 1989.

Ted Belytschko and Edward J. Plaskacz, "Observations Regarding CONNECTION Machine Performance for Nonlinear Dynamic Finite Element Analysis", Fourth SIAM Conference on Parallel Processing for Scientific Computing, Chicago, IL, December 11, 1989.

The following papers supported by NASA-Langley were submitted for publication:

Edward J. Plaskacz and Ted Belytschko, "Measurement and Exploitation of Mesh Structure on the CONNECTION Machine", Proc. of the ASME/Computers in Engineering International Exposition and Conference (Boston, August 1990), ASME, New York, NY, to appear, 1990.

Ted Belytschko, Edward J. Plaskacz, and James M. Kennedy, "Finite Element Computations on SIMD Machines with Hybrid Communication Schemes", Preprints of the Second World Congress on Computational Mechanics (Stuttgart, August 1990), to appear, 1990.

Noreen D. Gilbertsen and Ted Belytschko, "Explicit Time Integration of Finite Element Models on a Vectorized, Concurrent Computer with Shared Memory", Finite Elements in Analysis and Design, to appear, 1990.

Ted Belytschko, Edward J. Plaskacz, James M. Kennedy, and Donald L. Greenwell, "Finite Element Analysis on the CONNECTION Machine", Computer Methods in Applied Mechanics and Engineering, in press, 1990.

Martin R. Ramirez and Ted Belytschko, "An Expert System for Setting Time Steps in Dynamic Finite Element Programs", Engineering with Computers, 5(3/4), 205-219, 1989.

Mark O. Neal and Ted Belytschko, "Explicit-Explicit Subcycling with Non-Integer Time Step Ratios for Structural Dynamic Systems", Computers and Structures, 31(6), 871-880, 1989.

P. Smolinski, T. Belytschko, and M. Neal, "Multi-Time-Step Integration Using Nodal Partitioning", International Journal for Numerical Methods in Engineering, 26(2), 349-359, 1988.

Ted Belytschko, "On Computational Methods for Crashworthiness", Computers and Structures, submitted.

ABSTRACT

This report discusses techniques for the implementation and improvement of vectorization and concurrency in nonlinear explicit structural finite element codes. In explicit integration methods, the computation of the element internal force vector consumes the bulk of the computer time. The program can be efficiently vectorized by subdividing the elements into blocks and executing all computations in vector mode. The structuring of elements into blocks also provides a convenient way to implement concurrency by creating tasks which can be assigned to available processors for evaluation. The techniques were implemented in a three dimensional nonlinear program with one-point quadrature shell elements.

Concurrency and vectorization were first implemented in a single time step version of the program. An efficient implementation of subcycling, a mixed time integration method using different time steps for different parts of the mesh, was particularly difficult because of problems in scheduling processors and setting the optimal vector size. Techniques were developed to minimize processor idle time and to select the optimal vector length.

A comparison of run times between the program executed in scalar, serial mode

and the fully vectorized code executed concurrently using eight processors shows speed-ups of over 25. Using subcycling, the speed-up is three or more, depending on the problem and the number of processors used. Efficiency of concurrent execution decreases as the number of processors increase due to processor idleness, memory contention and the effect of nonparallelizable code.

Conjugate gradient methods for solving nonlinear algebraic equations are also readily adapted to a parallel environment. A new technique for improving convergence properties of conjugate gradients in nonlinear problems is developed in conjunction with other techniques such as diagonal scaling. It consists of an analytic continuation which forces a slow transition between plastic and elastic response. A significant reduction in the number of iterations required for convergence is shown for a statically loaded rigid bar suspended by three equally spaced springs. In larger problems, the improvement is not as dramatic, indicating additional refinement of the method.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Explicit finite element programs are very suitable to a parallel environment. In explicit methods, the internal force vector of an element is computed by the integration of the product of the stress matrix and the discrete gradient matrix. This calculation and preceding strain-displacement and constitutive evaluations can be efficiently vectorized by subdividing elements into blocks and performing all calculations on a block of elements instead of one element at a time. This structuring of the elements into blocks also provides a convenient way to create tasks which can be assigned to available processors for evaluation.

Significant speed-ups can also be achieved by vector-concurrent execution because blocks of elements can be assigned to different processors to execute simultaneously. Additional speed-ups can be realized by the implementation of subcycling, a mixed time integration method using different time steps for different parts of the mesh. Finally, explicit iterative methods, such as conjugate gradient methods, are also more attractive when viewed from the perspective of parallel processing. In these methods,

the stiffness matrix need not be stored or triangularized, while the calculation of the internal force vector is repeated many times. Because the internal force vector can be computed efficiently in vector-concurrent mode, iterative methods yield a substantial speed-up.

This report discusses techniques for the implementation and improvement of vectorization and concurrency in nonlinear structural codes. Concurrency is implemented using both compiler optimization and programming constructs called monitors. Vectorization is implemented by extensive recoding followed by compiler optimization. Because vectorization and concurrency tend to be competing processes in terms of solution speed-up, their relationship is examined and discussed. Finally, improvements of the convergence properties of explicit conjugate gradient methods are analyzed.

Chapter 2 reviews some fundamentals of parallel processing, including the classifications of computer architecture, the mechanics of vectorization and the techniques of parallelization. The coordination of processes using monitors is discussed in detail.

Chapter 3 describes the implementation of vectorization and concurrency in a nonlinear structural dynamics finite element program. Techniques required for the efficient vectorization of the internal force vector are discussed in detail. A description of the monitors required for the parallelization of the internal force vector is also given. Three versions of the program are used to study the benefits and disadvantages of vector-concurrent execution. The first version is the original scalar code run using one processor. The second version is the original code compiled using full

compiler optimization for concurrency and vectorization. This version uses 8 processors; however, no modifications were made to improve the coding. The third version is the vectorized, parallelized code using full compiler optimization and monitors to control concurrency. Four problems were used to study the effects of varying the number of processors, the number of elements per block and the problem size on the speed-up and efficiency of the execution.

Chapter 4 describes the implementation of subcycling into the vectorized, parallelized code described above. Additional vectorization techniques required for an efficient implementation are discussed as well as a new algorithm which minimizes processor idleness.

Finally, Chapter 5 discusses methods for improving convergence properties of conjugate gradient solution methods. A one-dimensional spring problem and a two-dimensional cantilever beam problem are used to compare the convergence properties of diagonal scaling and a new method called the zeta-parameter method.

The work presented in this report was performed on an Alliant FX/8 at Argonne National Laboratories. The Alliant is a shared-memory machine with 8 processors and a Fortran compiler capable of optimizing for concurrency, vectorization and scalar operations. All software required for the use of monitors was developed by Argonne National Laboratories.

Chapter 2

Fundamentals

2.1 Parallel Computer Architectures

Computers can be classified according to a number of architectural properties [19, 26]. A widely accepted classification was presented by Michael J. Flynn in 1966 who organized digital computers into four categories based on the multiplicity of instruction and data streams. An instruction stream is a sequence of instructions which are to be executed by a single processor, while a data stream is the sequence of data requested by the instruction stream. The four categories are listed below.

1. SISD: A single processor interprets one set of instructions and executes them on a single data set. Computers based on the classical von Neumann architecture, such as the VAX-11/780, fall in this category.

2. SIMD: Multiple processors interpret the same instructions and execute them on different data streams. Both array processors, such as the Burroughs ILLIAC IV, and pipeline processors, such as the CRAY-1 and the IBM 360, belong

to this category. From the viewpoint of a programmer, vectorization can be considered a SIMD process.

3. MISD: Multiple processors interpret different instructions and operate on a single data stream. This category is usually considered empty since it implies that a series of instructions operate on a single data item which is a characteristic of the SISD category.

4. MIMD: Multiple processors interpret different instructions and operate on different data streams. Multiprocessors such as the Alliant FX/8 and CRAY-2 are MIMD machines.

The three parallel computer structures mentioned above, i.e. array processors, pipeline processors and multiprocessors, achieve different types of parallelism. Note, however, that a parallel computer can have more than one of these architectural features. For example, the CRAY-2 and the Alliant FX/8 are multiprocessors with pipeline processing capability. These three computer structures are discussed in more detail below.

2.2 SIMD: Pipeline Processors and Array Processors

The vectorized execution of a do loop can be viewed as a SIMD process. Each operation in the loop is performed for a series of data pairs in parallel. Vectorized code can be executed on a pipeline processor or an array processor. From a programmer's

standpoint, the coding for both types of SIMD computers is the same. Architecturally, a pipeline processor differs from an array processor in the way the loop is executed. A pipeline processor divides each instruction into a number of stages and then executes the instruction for a sequence of data pairs in an overlapping fashion. This type of parallelism is referred to as *temporal parallelism*. An array processor has a control unit which distributes the data pairs to a number of processing units. Each processing unit executes the same instruction on the data pair, thus exploiting *spatial parallelism*. Because all processors in an array processor must execute the same instructions, it is difficult to achieve maximum performance on an array processor in a general computing environment. Therefore, most array processors are designed for a specific problem type or purpose.

A typical instruction involves retrieving data from memory, storing it in the vector or scalar registers, performing the specified operation and returning the result to memory. In a pipeline processor, each instruction is divided into a number of stages, such that the time required to complete each stage is a clock period (CP). A clock period is a unit of time which is specific to the computer. For example, a clock period for the CRAY-1 is 8.5 nanoseconds.

The following do loop will be used to illustrate how a pipeline processor can perform these functions in a more efficient manner. The multiplication of two numbers requires seven clock periods (CP).

```
      DO 10 J = 1,64
        C(J) = A(J) * B(J)
10 CONTINUE
```

Figure 2.1 shows a schematic of the multiplication instruction performed in scalar mode. A nonpipeline computer will process the entire instruction for the first data pair before beginning the second pair. The first pair of data, A(1) and B(1), are retrieved from memory and stored in scalar registers. They then enter the multiply functional unit. Once the seven stages have been completed, the result is stored in a third scalar register. The second data pair A(2) and B(2) are then retrieved from memory and the procedure continues until all data pairs have been processed. The total execution time required to execute the do loop in scalar mode is:

$$\begin{aligned} T_{np} &= \text{No. CP} * \text{vector length} \\ &= 7 * 64 = 448 \text{ clock-sec} \end{aligned}$$

Figure 2.2 [27] shows the schematic for the multiply in vector mode. All 64 pairs of data are retrieved from memory and stored in the vector registers. (Note: this example is based on the CRAY-1 computer which has vector registers of length 64. If the length of the vector registers is smaller than 64, the loop will be executed in groups of n pairs of elements, where n is the register length.) An additional stage, called a steering module, must be added before and after the multiply functional unit for a vector operation. The steering module requires one clock period. During the first clock period, the first data pair, A(1) and B(1) enter the steering module. In the second clock period, A(1) and B(1) enter the first stage of the multiply unit, and A(2) and B(2) enter the steering module. During each additional clock period, the

data pairs continue to move ahead to successive stages. This progression continues until all vector pairs have completed the instruction. The time required to execute the loop in vector mode is:

$$\begin{aligned} T_p &= \text{startup} + \text{vector length} \\ &= 9 + 64 = 73 \text{ clock-sec} \end{aligned}$$

The startup time is the number of clock periods required to fill up all stages of the multiply instruction and the two steering modules. The speed-up attributed to vector processing is computed by dividing the execution time required in scalar mode by the time required in vector mode. A speed-up of 6.1 was achieved for this example.

2.3 MIMD: Multiprocessors

A multiprocessor is a computer containing two or more processors which can operate asynchronously. Each processor is allowed to perform different instructions on different sets of data and can communicate with one another by sharing common memory or by sending messages. Multiprocessors are said to achieve *asynchronous parallelism.*

Multiprocessors can be categorized into three groups based on the manner in which the processors communicate with one another. The first is a *tightly coupled* multiprocessor where processors communicate via a shared main memory. In a *loosely coupled* multiprocessor, each processor possesses a small local memory or cache and

communicates by sending messages though a message-transfer system. The third group is comprised of loosely coupled clusters of processors. Within each cluster, the processors communicate though a shared memory. The clusters can then communicate with each other by sending messages. This third group uses a combination of tightly coupled and loosely coupled communication mechanisms. Work presented in this report was performed on the Alliant FX/8 which is a tightly coupled multiprocessor.

Different programming methods are required for each type of multiprocessor. Concurrency was implemented using both compiler optimization and programming constructs called monitors which were developed at Argonne National Laboratory.

2.3.1 Compiler Optimization

Parallelism can be implemented on a multiprocessor by simply invoking the concurrency option when compiling a program. The compiler will attempt to execute every do loop in either scalar-concurrent or vector-concurrent mode, depending on whether the vectorization option has also been selected. Compiler optimization allows the user to receive some benefit from multiple processors with little or no modifications to a program designed for a single processor.

The two modes of concurrent execution, scalar-concurrent and vector-concurrent, can be illustrated using the following loop.

```
      DO 10 I = 1,512
        C(I) = A(I) + B(I)
10 CONTINUE
```

In scalar-concurrent mode, each processor will perform the scalar addition for one data pair at a time, i.e. the first processor will add A(1) and B(1), the second processor will add A(2) and B(2), etc. In vector-concurrent mode, each processor performs a vector addition on n data pairs simultaneously, where n is the length of the vector registers. The Alliant FX/8 has vector registers of length 32 and 8 processors. Therefore, on the Alliant, the first processor will execute the addition for the first 32 pairs of data. The second processor will add the next 32 pairs of data. The theoretical speed-up of scalar-concurrent mode and vector-concurrent mode achieved over scalar execution is 8 and 32, respectively [29]. Vectorization alone has a nominal speed-up of four. Theoretical speed-up for concurrency can only be achieved for highly optimized test problems such as the example presented above. In standard coding, the actual speed-up falls far short of the theoretical speed-up. For example, compiler optimization was used for the nodal calculations in the finite element program presented in Chapter 3. The speed-up attributed to concurrency using 8 processors was approximately 4.75. Factors which limit speed-up are discussed in detail in Chapters 3 and 4.

2.3.2 Monitors

Greater benefit can be achieved from a multiprocessor by taking advantage of tools which have been developed to coordinate communication between processes.

The general procedure is to determine which sections of the program can be performed in parallel and which sections must be done sequentially. The parallel sections are divided into tasks and an algorithm is developed to determine the order in which the tasks are to be performed. Each task is then assigned to an available process for execution.

One approach developed to coordinate the communication between processes in shared-memory multiprocessors involves the use of a programming construct called a *monitor*. Monitors were originally developed in the early 1970's for the research in the area of operating systems. They have recently been applied to the assignment of tasks in a parallel environment. The monitors developed at Argonne National Laboratory were used for this research and are described in detail in [10, 21, 22].

Before discussing the characteristics of a monitor, it is useful to emphasize the differences between the definitions of certain terms which will be used in this section. A *processor* is the piece of hardware on a computer which executes instructions. The development of a parallel algorithm is independent of the number of processors. A *process* is the mechanism which performs a specified task and a *program* is the description of that task. The definition of a process is rather abstract; however, the concept should become clear within the context of the description of a monitor. The important distinction between these three terms is that processes can communicate with one another. Programs and processors can not.

A monitor is a programming construct which coordinates the communication between *processes*. The monitor must also ensure that each process has access to

globally-shared memory without interference from other processes. A monitor is formally defined as an abstract data type consisting of three components.

1. data shared by competing processes,

2. initialization of the shared data,

3. operations to be performed on the data.

To illustrate these three components, consider the following do loop to be performed in parallel using monitors.

```
      DO 10 J = 1,100
        C(J) = A(J)*B(J)
10 CONTINUE
```

The shared data are the loop indices which range from 1 to 100. The initialization code sets an integer SUB equal to 0. SUB indicates which task is ready to be performed. For this example, SUB is set equal to the loop index which will be assigned to a particular process. The operations on the shared data include incrementing the value of SUB and ensuring that SUB does not exceed 100. An important characteristic of a monitor is that the operations on the shared data can only be performed by one process at a time, thereby preventing the same index from being given to more than one process.

The monitor operations are implemented by using four basic macros. In order for a process to execute a monitor operation, the process must first take exclusive control of the monitor by issuing a *menter* command. If the process successfully

performs the monitor operation, a *mexit* command is issued which releases exclusive control, thus allowing another process to access the monitor. If the process enters the monitor and cannot successfully perform the monitor operation, or in other words, finds no task to perform, it issues a *delay* command. The delay command indicates the location where the delay was issued and releases control of the monitor. The process will remain "delayed" until another process enters the monitor and either adds new tasks or completes a task which allows the delayed process to continue. The new process executes a *continue* operation, giving the location where the old process was delayed, and exits the monitor. The first process can then reenter the monitor at the named location and continue the monitor operation.

An informal description of monitors given in [10, 21] is helpful in understanding the concept. The monitor can be thought of as a *room* with a number of attached *closets.* The room contains the shared data, the initialization code and all operations to be performed on the data. The room has a single door from which a process can enter or leave the room, however, only one process is allowed in the room at a time. Once a process has entered the room (*menter*), it can perform the monitor operation on the shared data and then leave the room (*mexit*), thus allowing another process to enter. However, if for some reason the process cannot carry out the operation (such as in the do loop example, if the value of SUB is greater than 100), a *delay* is issued and the process will exit the room by entering one of the closets. This allows a new process to enter the room. If the new process adds a task, it will issue a *continue* and leave the room. The old process can then exit the closet, operate on the new

task and exit the room.

Monitors have been developed for a number of common synchronization patterns, such as the *self-scheduling loop monitor* which can be used for the simple do loop described above. The task associated with the incremented subscript in the self-scheduling do loop can be quite complex. In fact, it is more efficient to write parallel codes with a relatively large task size or *large granularity.* For small task size, the contention for sequential access to the monitor can severely limit or eliminate the benefits of parallelism. The self-scheduling loop monitor does not allow the flexibility to add new tasks or to modify the algorithm which assigns tasks to processes. A monitor called the *askfor* monitor appears to be the most powerful monitor for implementing parallelism on a multiprocessor. This monitor and the *lock* monitor were used for this research and will be discussed in detail.

The *askfor* monitor uses the concept of a "pool of tasks" from which a process can "ask for" the next available task to perform. Each task can be subdivided into one or more subtasks. A major task can be solved either by completing all of its subtasks, referred to *solution by exhaustion*, or if the solution of a subtask gives the solution of the major task. The self-scheduling do loop is an example of solution by exhaustion. The major task (the entire loop) is solved when all of the subscripts have been assigned and the corresponding subtasks has been performed. An example of the second type of solution consists of several processes searching for the solution to the major task. As soon as one process finds the solution, the major task is considered solved and the remaining processes are terminated.

The askfor monitor is particularly flexible because it allows processes to add and subtract tasks from the pool. It also allows the user to define the algorithm which determines how the tasks will be assigned to the processes. The askfor monitor is invoked by the following expression:

```
askfor (<name>,<rc>,<nprocs>,<getprob>,<reset>)

where,

    <name>    : monitor name,

    <rc>      : return code which equals -
                      0 for successful acquisition of a
                          task from the pool,
                      1 if no tasks remain in the pool,
                     -1 if the program is completed,
                   n > 1 if the task is completed (used when multiple
                          processors work on the same task),

  <nprocs>    : number of competing processes,

  <getprob>   : user-defined macro describing logic required
                  to claim a task and set the return code,

  <reset>     : user-defined macro that resets the pool
                  of tasks.
```

A do loop can be executed using the askfor monitor by defining the following user-defined macros:

```
askfor(MO,RC,NPROCS,getprob(I,N,RC),reset)

define(getprob,
        [IF (SUB .GT. 0) THEN
          [IF (SUB .LE. $2) THEN
             $1 = SUB
             SUB = SUB + 1
             $3 = 0
           ENDIF]
         ENDIF] )

define(reset,
        [SUB = 0] )

define(probstart,
        [menter(MO)
          SUB = 1
          mcontinue(MO,1)
         mexit(MO)] )
```

The integers preceded by the $ refer to the arguments in the getprob macro invocation. The variable SUB represents the shared data and indicates which task or subscript is to be assigned next. SUB is assigned a value of 1 by the macro *probstart* which is issued prior to the parallel operation. The second argument in the getprob macro is the number of loop indices. For values of SUB less than the number of loop indices, the monitor will assign the task number to the process, increment SUB and set the return code to zero denoting a successful task acquisition. After all of the subscripts have been assigned, the *reset* macro sets SUB equal to zero which indicates to the monitor that all processes entering the monitor should be delayed until new tasks have been added to the pool.

It is frequently necessary to protect data stored in shared memory which can be accessed by more than one process simultaneously. Simultaneous access can result in one or more of the processes receiving erroneous data. It is also necessary to synchronize the update of a variable stored in shared memory so that one process does not void the update of another process. This protection is provided by a *lock* monitor which simply places the access instructions between a *menter* command and a *mexit* command. The macros invoking the lock monitor are *nlock(name)* and *nunlock(name)*. The following example illustrates how locks can be used in computing the dot product of two vectors A and B. The integer I refers to the subscript assigned to a particular process. Each process will have a different value for I but will access the same shared memory location to update D.

```
C(I) = A(I)*B(I)
nlock(L1)
  D = D + C(I)
nunlock(L1)
```

Different locks should be defined for different words and arrays in shared memory so that unrelated processes do not get delayed unnecessarily.

Monitors are implemented using low-level and high-level macros which define the behavior of the monitor and perform initialization functions required for execution. The low-level macros include those macros which are machine dependent, such as the four basic macros used to define the monitors: *menter, mexit, mcontinue* and *mdelay*. These relatively few macros are then used as the building blocks to create high-level macros such as the *askfor* macro. The use of macros allow the programmer to develop

codes which are machine independent and therefore can be easily transported to other shared-memory multiprocessors. Only the low-level macros are machine dependent and must be modified in order to run the program on a different computer.

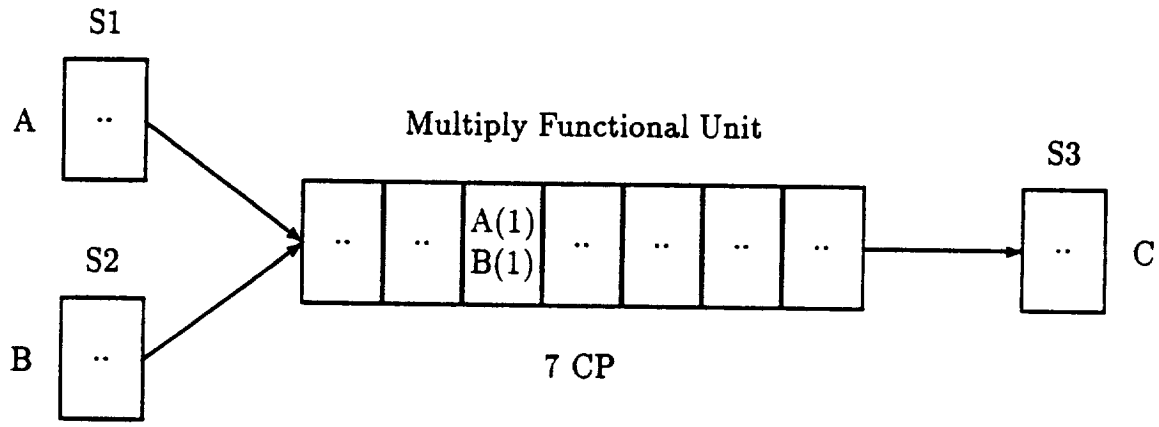


Figure 2.1: Stages of a Scalar Multiplication

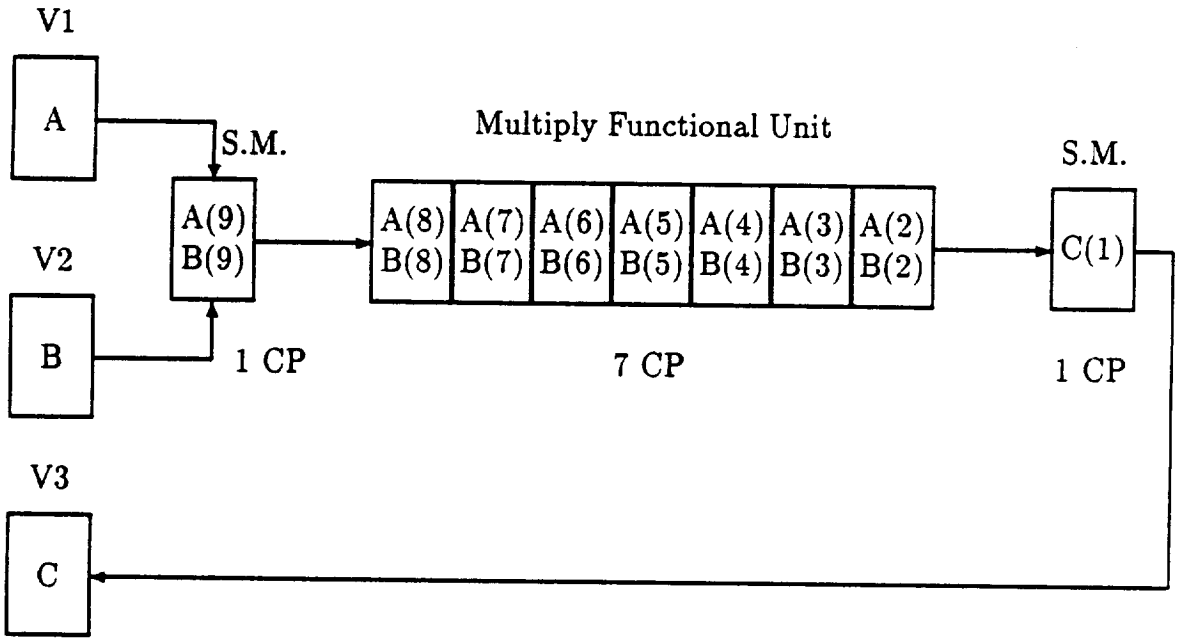


Figure 2.2: Stages of a Vector Multiplication

Chapter 3

Vectorized, Concurrent Finite Element Program

3.1 Introduction

A nonlinear structural dynamics finite element program has been developed to run on a shared-memory multiprocessor with pipeline processors. The program WHAMS [7] was used as a framework for this work. The program employs explicit time integration and has the capability to handle both the nonlinear material behavior and large displacement response of three dimensional structures. The elasto-plastic material model, described in [30], uses an isotropic strain hardening law which is input as a piecewise linear function. Geometric nonlinearities are handled by a corotational formulation in which a coordinate system is embedded at the integration point of each element. Currently, the program has an element library consisting of a beam element based on Euler-Bernoulli theory and triangular and quadrilateral plate elements based on Mindlin theory.

3.2 Explicit Finite Element Formulation

3.2.1 Finite Element Equations

The equations of motion for a structural system are given by:

$$\boldsymbol{M}\,\boldsymbol{a} + \boldsymbol{f}_{int} = \boldsymbol{f}_{ext} \tag{3.1}$$

where,

$\boldsymbol{M}$ = global mass matrix,

$\boldsymbol{a}$ = nodal accelerations,

$\boldsymbol{f}_{int}$ = assembled internal nodal forces,

$\boldsymbol{f}_{ext}$ = assembled external nodal forces.

The mass matrix is assumed to be diagonal and lumped so that the system equations are uncoupled. The internal nodal force is computed on the element level by

$$\boldsymbol{f}^{e}_{int} = \int_{\Omega^e} \boldsymbol{B}^T \boldsymbol{\sigma}\, d\Omega \tag{3.2}$$

and then assembled by

$$\boldsymbol{f}_{int} = \sum_{e} (\boldsymbol{L}^e)^T \boldsymbol{f}^{e}_{int} \tag{3.3}$$

where,

$$\begin{aligned} \boldsymbol{f}^e_{int} &= \text{element internal nodal force,} \\ \Omega^e &= \text{domain of the element,} \\ \boldsymbol{B} &= \text{gradient matrix,} \\ \boldsymbol{\sigma} &= \text{Cauchy stress matrix,} \\ \boldsymbol{L}^e &= \text{Boolean connectivity matrix.} \end{aligned}$$

Equation 3.3 gives the assembly of the element internal nodal forces into the global array. The array $\boldsymbol{L}^e$ is never computed; instead the operation indicated by Eqn. 3.3 is implemented by simply adding the entries of the element array into the appropriate locations in the global array as described in Section 3.3.2.

The element stresses are computed from the corotational components of the velocity strain $\boldsymbol{d}$ given by

$$\hat{d}_{ij} = \frac{1}{2}\left(\frac{\delta \hat{v}_i}{\delta \hat{x}_j} + \frac{\delta \hat{v}_j}{\delta \hat{x}_i}\right) \tag{3.4}$$

where the superposed 'hat' signifies components expressed in terms of the base vectors of the corotational coordinate system. The velocity at any point in the plate is given by Mindlin theory as

$$\boldsymbol{v} = \boldsymbol{v}^m \; - \hat{z}\boldsymbol{e}_3 \; \times \boldsymbol{\theta} \tag{3.5}$$

where,

$$\begin{aligned} \boldsymbol{v}^m &= \text{velocity of plate midsurface,} \\ \hat{z} &= \text{distance from midsurface,} \\ \boldsymbol{e}_3 &= \text{base vector perpendicular} \\ &\quad \text{to the plane of the plate,} \\ \boldsymbol{\theta} &= \text{angular velocity.} \end{aligned}$$

Once the corotational components of the velocity strain have be computed, the appropriate constitutive law can be applied to calculate the element stresses.

3.2.2 Time Integration

The following central difference equations are used to update the nodal velocities and displacements in time. Note that an average time step is used to update the velocities. This allows for the capability of changing the time step during the problem solution.

$$\begin{aligned} \boldsymbol{v}^{n+\frac{1}{2}} &= \boldsymbol{v}^{n-\frac{1}{2}} + \Delta t^n \, \boldsymbol{a}^n \\ \boldsymbol{u}^{n+1} &= \boldsymbol{u}^n + \Delta t^{n+\frac{1}{2}} \, \boldsymbol{v}^{n+\frac{1}{2}} \\ \Delta t^n &= \tfrac{1}{2}\left(\Delta t^{n-\frac{1}{2}} + \Delta t^{n+\frac{1}{2}}\right) \end{aligned} \tag{3.6}$$

where,

$$\begin{aligned} \boldsymbol{u}, \boldsymbol{v}, \boldsymbol{a} &= \text{nodal displacement, velocity and} \\ &\quad \text{acceleration, respectively,} \\ \Delta t^n &= \text{time increment for step } n. \end{aligned}$$

The superscripts in the above equations designate time steps. The fractional superscripts indicate a midstep value. An outline of the explicit time integration scheme is given below.

Flow Chart for Explicit Integration

1. Initial conditions : $\boldsymbol{v}^{-\frac{1}{2}}, \boldsymbol{u}^o$

2. Compute external force

3. Compute internal force vector $\boldsymbol{f}_{int}^{n+1}$

   Loop over element blocks

   (a) compute velocity strains

$$\boldsymbol{d}^{n+\frac{1}{2}} = \boldsymbol{B}\, \boldsymbol{v}^{n+\frac{1}{2}} \tag{3.7}$$

   (b) compute frame invariant stress rates

$$\overset{\nabla}{\boldsymbol{\sigma}}{}^{n+\frac{1}{2}} = \boldsymbol{\mathcal{S}}(\boldsymbol{\sigma}, \boldsymbol{d}) \tag{3.8}$$

(c) convert frame invariant rate to time derivative of Cauchy stress

$$\dot{\boldsymbol{\sigma}}^{n+\frac{1}{2}} = \overset{\nabla}{\boldsymbol{\sigma}}^{n+\frac{1}{2}} + \boldsymbol{W}^{n+\frac{1}{2}} \cdot \boldsymbol{\sigma}^{n} - \boldsymbol{\sigma}^{n} \cdot \boldsymbol{W}^{n+\frac{1}{2}} \tag{3.9}$$

(d) update stress

$$\boldsymbol{\sigma}^{n+1} = \boldsymbol{\sigma}^{n} + \Delta t \, \dot{\boldsymbol{\sigma}}^{n+\frac{1}{2}} \tag{3.10}$$

(e) compute element internal nodal force : Eqn. 3.2

(f) assemble into global array

4. Compute accelerations by equation of motion : Eqn. 3.1

5. Update velocities and displacements using central difference equations: Eqn. 3.6

6. Go to 2.

REMARK: In Eqn. 3.8, $\overset{\nabla}{\sigma}$ is a frame invariant rate such as the Jaumann rate and $\boldsymbol{W}$ is the spin tensor.

3.2.3 Evaluation of Critical Time Step

For explicit problems, the time step is calculated based on a numerical stability criterion. The critical time step for a undamped linear system of equations updated

using central difference equations is given by [2]:

$$\Delta t_{cr} = \frac{2}{\omega_{max}} \tag{3.11}$$

where ω_{max} is the maximum frequency of the system

$$\mathbf{Ku} = \lambda \mathbf{Mu} \tag{3.12}$$

and,

$$\lambda = \omega_{max}^2 \tag{3.13}$$

The element eigenvalue inequality theorem states that the maximum absolute system eigenvalue is bounded by the maximum element eigenvalue, i.e.,

$$|\lambda| \leq |\lambda_{E_{max}}| \tag{3.14}$$

where,

λ = maximum system eigenvalue,

and,

$\lambda_{E_{max}}$ = maximum λ_e for all elements.

The maximum frequency for a one dimensional rod element with linear displacements and diagonal mass can be easily calculated as $2c/l$ where c is the dilatational wave

speed and l is the length of the element. The critical time step for the element is l/c. Physically, this time step corresponds to the amount of time required for a stress wave to traverse the smallest dimension of the element. Therefore, the critical time step for explicit time integration is calculated based on the dimensions and material properties of the element with the largest frequency. The critical time step decreases as the size of the element decreases.

Maximum element frequencies for the bending, shear and membrane response of the 4-node Mindlin plate element are presented in [12, 3] and summarized below. The critical time step of the element corresponds to the largest of the computed frequencies.

Bending:

$$\omega_{max} = \left[\frac{\frac{D}{A} \left\{ R_1 + [R_1^2 - 4\,(1-\nu^2)\,R_2^2]^{\frac{1}{2}} \right\}}{\rho\, A\, h\, \alpha} \right]^{\frac{1}{2}} \tag{3.15}$$

Membrane:

$$\omega_{max} = \frac{1}{A} \left\{ \frac{E}{(1-\nu^2)\,\rho} \left[R_1 + \left(R_1^2 - 4\left(1-\nu^2\right) R_2^2 \right)^{\frac{1}{2}} \right] \right\}^{\frac{1}{2}} \tag{3.16}$$

Shear:

$$\omega_{max} = \left[\frac{\frac{4\,c\,\alpha}{A}\,\hat{a}_{11} + c\,A}{\rho\, A\, h\, \alpha} \right]^{\frac{1}{2}} \tag{3.17}$$

where,

$$\begin{aligned}
\rho &= \text{density of the material,} \\
A &= \text{cross-sectional area,} \\
h &= \text{thickness of the element,} \\
\alpha &= \text{rotational inertia scaling factor,} \\
&= \tfrac{h^2}{12}, \\
D &= \tfrac{E h^3}{12(1-\nu^2)}, \\
\hat{a}_{11} &= \tfrac{1}{4}(R_1 + R_3), \\
R_1 &= y_{24}^2 + y_{31}^2 + y_{24}^2 + y_{31}^2, \\
R_2 &= y_{32}\, x_{24} - y_{24}\, x_{31} \\
R_3 &= (R_1^2 - 4\, R_2^2)^{\frac{1}{2}},
\end{aligned}$$

and,

$$y_{IJ} = y_I - y_J.$$

The lumped mass matrix used in the frequency calculations is given by

$$M = \frac{\rho A h}{4} \begin{bmatrix} I_{4\times 4} & 0 \\ 0 & \alpha I_{8\times 8} \end{bmatrix} \tag{3.18}$$

The rotational inertia scaling factor, α is given in [17] as

$$\alpha = \frac{I}{A} = \frac{h^2}{12} \tag{3.19}$$

for a plate element with uniform material properties. I is the moment of inertia of the element.

The stability analysis performed to estimate the critical time step is based on a linear system of equations. However, experience has shown that a linearized analysis provides good estimates of the stable time step. For nonlinear problems, the critical time step is reduced 5% to 10% to compensate for potential destabilizing effects due to nonlinearities. In addition, an energy balance is performed for every time step in order to monitor the stability of the system.

3.3 Vectorization

3.3.1 Compiler Vectorization

When compiling a program on a computer with vector processors, options are available for automatic vectorization. The compiler will attempt to vectorize each do loop in the program. Compilers differ in their ability to vectorize programming constructs such as IF statements in loops. However, current compilers will not vectorize loops which contain any of the following statements:

1. Data dependencies
2. Ambiguous subscripts
3. Certain IF statements such as
   (a) block IF, ELSE, ENIDIF with nesting at a level greater than 3

(b) ELSE IF statements

(c) IF, GOTO label outside of loop

4. READ or WRITE statements

5. Subroutine calls

The compiler will usually issue an explanation if it is unable to vectorize the loop. Additional details about vectorization can be found in [27].

In order to maximize the benefits of vectorization, modifications to the program are frequently required. In many cases, minor changes are sufficient to enable a loop to vectorize or to improve the efficiency of the loop. The following examples illustrate two typical situations in which an existing do loop can be easily modified for efficient vectorization.

In nested do loops, only the innermost loop will vectorize. Therefore, the inner loop should have the largest range of indices. If this is not the case, the inner and outer loops can sometimes be interchanged without affecting the calculations. If the range of the inner loop is sufficiently small, the inner loop can be "unrolled," thus allowing the outer loop to vectorize. In the following do loop, the compiler will attempt to vectorize the inner loop, leaving the remaining calculations to be performed in scalar mode.

```
      DO 10 I = 1,1000
              .
              .      (other calculations)
              .

        A(I) = I*I
        DO 10 K = 1,3
          A(I) = A(I) + B(K)
   10 CONTINUE
```

Unrolling the inner loop allows all calculations to vectorize.

```
      DO 10 I = 1,1000
              .
              .      (other calculations)
              .

        A(I) = I*I
        A(I) = B(1) + B(2) + B(3)
   10 CONTINUE
```

Programming techniques are frequently different for vectorized codes than for scalar codes. For scalar programs, efficient coding consists of minimizing the number of calculations performed. In a vectorized code, it is more important to retain the vector structure of the computations. For example, in the following scalar loop, it is worthwhile to use an IF statement to check whether the component of A is equal to zero and if it is, omit the computation. A GO TO statement avoids unnecessary calculations.

```
      DO 10 I = 1,1000
        IF (A(I) .EQ. 0.0) GO TO 10
        D(I) = D(I) + A(I)*C(I)/(I*I)
   10 CONTINUE
```

In a vectorized version of this loop, the IF statement can be eliminated to preserve vectorization.

Suppose that the IF statement in the above example read

```
IF (A(I) .GE. 3.2*B(I)) GO TO 10
```

It is no longer possible to simply remove the IF statement. Some compilers will vectorize this type of do loop by doing a *gather/scatter* operation on the vector A. In gather/scatter, the compiler creates a temporary array which contains all values of A(I) less than 3.2 times B(I) and computes the update on D(I) only for this subset of A. If the compiler does not have a gather/scatter capability, it is possible to maintain vectorization by defining a temporary vector for the calculation.

```
      DO 10 I = 1,1000
        TEMP(I) = A(I)
10    CONTINUE

      DO 20 I =1,1000
        IF (A(I) .GE. 3.2*B(I)) TEMP(I) = 0.0
20    CONTINUE

      DO 30 I = 1,1000
        D(I) = D(I) + TEMP(I)*C(I)
30    CONTINUE
```

In the preceding loops, only the first and third loops will vectorize for compilers without gather/scatter capabilities. The first loop is overhead required to retain vectorization in the third loop. The second loop is performed in scalar mode. Although this example is rather trivial, the technique can be quite useful for vectorizing many

types of loops. As will be discussed in the section on concurrency, creating temporary arrays also helps minimize memory contention problems inherent in shared-memory multiprocessors.

Minor modifications, such as those presented above, will yield moderate improvements in speed-up due to vectorization. However, in order to best exploit the vectorization capabilities of the computer, it is frequently necessary to restructure the flow of the program by replacing calculations for a single element or node by loops which perform the calculations for a group of elements or nodes. This restructuring is discussed in the following section.

3.3.2 Vectorization of Internal Nodal Force Array

One way to approach the vectorization of a large program is to determine which portions of the code require the most computational time. The longer the computational time, the more effort should be devoted to vectorization. For an explicit finite element program, a large part of the time is consumed by the computation of the internal nodal force vector, $\boldsymbol{f}_{int}$. In the scalar code, the element internal force vector is calculated and assembled into the global array for one element at a time. Since the internal force vectors of all elements are independent at a given time step, the internal force calculations are very conducive to vectorization. Instead of performing the calculations for individual elements, the internal force computation can easily be vectorized by placing the operations within a loop and performing the calculations for a block of elements.

The procedure is as follows: The elements are divided into blocks of identical element type and material model. It does not matter if material properties for each element are identical as long as the model (i.e., elastic or von Mises elastic-plastic) is the same. The number of elements placed in each block depends on the length of the vector registers and certain characteristics such as problem size. The criteria for selecting block size are discussed later.

Once the elements have been divided into blocks, the scalar calculations can be transformed to vector calculations by converting scalar variables to arrays and placing operations in do loops. For example, trial stresses for an elastic-plastic material model are computed in scalar mode for one element by:

```
BOX 1 : Trial Stress Computation in Scalar Mode

        SNEW1 = SOLD1 + SDEL1
        SNEW2 = SOLD2 + SDEL2
        SNEW3 = SOLD3 + SDEL3
```

In vectorized form, the calculations are modified as:

```
BOX 2 : Trial Stress Computation in Vector Mode

        DO 10 J = 1,NEPB
          SNEW1(J) = SOLD1(J) + SDEL1(J)
          SNEW2(J) = SOLD2(J) + SDEL2(J)
          SNEW3(J) = SOLD3(J) + SDEL3(J)
     10 CONTINUE
```

where NEPB is the number of elements in the block. The computed arrays are stored in common blocks so that they can be accessed by any subroutine. Note that vectorization substantially increases the amount of memory required because of the large number of arrays that are created. In older computers, the small core capacity would have significantly limited the size of problems which could run using a vectorized code. However, recent technological advances have made large memory cores available and practical, thus eliminating size limitations except for extremely large problems.

Vectorization is fairly straightforward for many computations, however certain modifications must be made to exploit vectorization in some algorithms. The calculation of the plastic constitutive equation is an illustration of this situation. In a scalar code, a trial stress state is computed for the element and compared to the yield stress. If the element is elastic, the stresses are updated and the subroutine is exited. However if the element is plastic, additional calculations are required. In a vectorized code, the same calculations must be performed for all elements in the block. When a block contains a mixture of elastic and plastic elements, the elastic elements must perform the plastic calculations without modifying the elastic stresses.

This was accomplished by creating two arrays, KE(NEPB) and KP(NEPB) which indicate whether the element is elastic or plastic (KE = 1 and KP = 0 if the element is elastic and visa versa if the element is plastic). If all elements in the block are elastic, the stresses are updated and the plastic calculations are omitted. Otherwise, all elements perform the plastic stress calculations and the appropriate stress is

stored. The following coding illustrates the flow of the vectorized calculations of the updated stresses.

```
C
C     VECTORIZED SINGLE PROCESSOR VERSION
C     OF STRESS COMPUTATION
C
C     YIELD FUNCTION:
C
      SIGEF2(XX,YY,XY) = XX*XX + YY*(YY - XX) + 3.*XY*XY
C
C     COMPUTE TRIAL STRESS STATE
C
      DO 10 J = 1,NEPB
        SNEW1(J) = SOLD1(J) + SDEL1(J)
        SNEW2(J) = SOLD2(J) + SDEL2(J)
        SNEW3(J) = SOLD3(J) + SDEL3(J)
C
C     APPLY YIELD CRITERION
C     IF ELASTIC : KP = 0, KE = 1
C     IF PLASTIC : KP = 1, KE = 0
C
        S1(J) = SQRT(SIGEF2(SNEW1(J),SNEW2(J),SNEW3(J)))
        KP(J) = 0.5 + SIGN(0.5,S1(J) - YIELD(J))
        KE(J) = 1. - KP(J)
 10   CONTINUE
C
C     IF ALL ELEMENTS ARE ELASTIC, RETURN
C
      DO 20 J = 1,NEPB
        IF (KE(J) .EQ. 0) GO TO 40
 20   CONTINUE
      RETURN
 40   CONTINUE
C
C     COMPUTE PLASTIC STRESS
C
```

```
      SPL1(J) = . . .
      SPL2(J) = . . .
      SPL3(J) = . . .
C
C     UPDATE PLASTIC STRESS FOR PLASTIC ELEMENTS AND
C     ELASTIC STRESS FOR ELASTIC ELEMENTS
C
      DO 60 J = 1,NEPB
        SNEW1(J) = KP(J)*SPL1(J) + KE(J)*SNEW1(J)
        SNEW2(J) = KP(J)*SPL2(J) + KE(J)*SNEW2(J)
        SNEW3(J) = KP(J)*SPL3(J) + KE(J)*SNEW3(J)
 60   CONTINUE
```

REMARK 1: The DO 20 loop will not vectorize because it contains a GO TO statement to a label outside of the loop.

REMARK 2: Radial return is a particularly simple plasticity algorithm that is easily vectorized. However, radial return is not readily adapted to plane stress [16].

Some calculations, such as those containing data dependencies, should not be vectorized. Most compilers will check for data dependencies and automatically supress vectorization. However, a compiler directive which prevents vectorization can also be placed immediately preceding the location where vectorization should be stopped. Options are available to enforce the directive for a single loop, for the rest of the subroutine or for the rest of the program. Data dependencies occur frequently when nodal arrays stored in global memory are updated.

For example, after the internal nodal forces for an element are computed, they

must be assembled into the global array. The assembly procedure for the x-component of the element internal force is:

```
      DO 10 J = 1,NEPB
        FINT(N1(J) + 1) = FINT(N1(J) + 1) + F1X(J)
        FINT(N2(J) + 1) = FINT(N2(J) + 1) + F2X(J)
        FINT(N3(J) + 1) = FINT(N3(J) + 1) + F3X(J)
        FINT(N4(J) + 1) = FINT(N4(J) + 1) + F4X(J)
   10 CONTINUE
```

where,

N1,N2,N3,N4	=	Shared memory location indices for local nodes 1,2,3 and 4,
FINT	=	Global internal nodal force vector,
F1X,F2X,F3X,F4X	=	Internal nodal force increment for local nodes 1,2,3 and 4 of element J.

If this loop were to vectorized, the pipeline processor would first retrieve from memory the values of the internal nodal force vector for local node 1 of all elements in the block. These values are then stored in a vector register. The temporary element array F1X is added to the internal nodal force vector. The result is then replaced in global memory. An error would arise when two elements in a block have the same global node for local node 1. For example, suppose elements 1 and 3 have global node 35 as their local node 1. The internal force increment for both element 1 and 3 will be added to the same value of the internal force of node 35. However, when the updated

value for node 35 is returned to memory, only the contribution from element 3 is saved. The update from element 1 is stored first and then overwritten by the update from element 3. Vectorization must be prevented in all loops containing updates to nodal variables in the element internal force calculations. It is not necessary to inhibit vectorization in arrays which pertain to element variables such as stress, strain and thickness because there will be no data dependencies among elements in a block.

Techniques have been developed to avoid data dependencies when updating arrays stored in global memory such as the internal force vector discussed above. In [18], an algorithm is presented which divides elements into blocks based on the criterion that no two elements in a block share a common node. This algorithm eliminates the data dependencies in the update of the nodal array and allows the loop to vectorize. Note, that a gather-scatter operation is still required to update the array because of the nonconstant stride between entries stored in global memory. Therefore, for a simple update of a globally stored array, an algorithm eliminating data dependencies will not yield significant speed-up. However, if the elements of the nodal array are used for additional computations, such as the matrix multiplication presented in [18], substantial benefit can be achieved. However, the sorting of arrays prior to assembly can be quite awkward, particularly if rearrangements are needed for other purposes, such as blocking elements by type or material [13].

3.4 Concurrency

Concurrency can be implemented using compiler options, as described in Section 2.3.1, for all calculations except for the assembly of the internal force vector. In compiler implementations of vectorization and concurrency, loops whose indices exceed the length of the vector registers will be executed in vector-concurrent mode. However, an effective implementation of vectorization-concurrency requires reprogramming with *monitors* which allow the scheduling of calculations among processors.

Two monitors were used for the parallelization of the code. The *askfor* monitor controls the assignment of tasks to the available processes. There are two types of processes. The master process is created by the operating system and performs all of the scalar operations as well as part of the parallel operations. The slave processes are created by the master process for parallel computations only. The task assigned to each process is to compute the internal force vector for one block of elements. Note that processes involve blocks of elements because of vectorization. Prior to entering the parallel operations, two macros are called by the master process. The first, *probstart*, initializes the task number. The task number refers to the block of elements to be assigned to a process. The macro is defined in Section 2.3.2.

The second macro, *create_and_work*(NPROCS) creates NPROCS-1 slave processes, where NPROCS is the number of competing processes. NPROCS also corresponds to the number of available processors and is an input variable. Each of the slave processes calls SUBROUTINE WORK, which is the subroutine which invokes

the askfor monitor. The master process then calls SUBROUTINE WORK. Therefore, NPROCS processes execute the operations in WORK simultaneously. The create_and_work macros is defined by:

```
      define(create_and_work,
        [DO 30 I = 1,NPROCS-1
           create(SLAVE)
30      CONTINUE
        CALL WORK ] )
```

In SUBROUTINE WORK, the askfor monitor is invoked using the following expression, discussed in detail in Section 2.3.2:

```
askfor (MO,RC,NPROCS,getsub(I,NBLOCKS,RC),reset)
```

The master process enters the askfor monitor with SUB = 1, the current task number. SUB is the shared data and is initialized to 1 by the macro *probstart*. The macro *getprob* assigns the task number to the process in the monitor. The subcript SUB is then incremented and the return code RC is set to 0 indicating a successful acquisition of the task. The process exits the monitor allowing the next process to enter. This procedure continues until the incremented subscript exceeds NBLOCKS, in which case the processes are delayed. When all slave processes are delayed, the master process exits the monitor operations and returns to the nonparallelized code.

The second monitor used is the *lock* monitor which protects access to shared memory. When variables from shared memory are required for parallel calculations, they are first stored in temporary arrays using a "gather" operation discussed in

Section 3.3.1. This minimizes memory contention problems encountered in shared-memory multiprocessors and also benefits vectorization. However, if two or more processes attempt to access the same memory location simultaneously, an error will occur. Therefore, the gather operation is placed within a lock monitor and only one process is allowed to access a particular subset of memory at a time. In other words, the instruction to access global memory becomes the monitor operation.

A lock is invoked by a *nlock* macro immediately preceding the operation. The locks are named so that different subsets of memory can be associated with different lock monitors. Control of the monitor is released after the operation by the *nunlock* macro. The following example shows the "gathering" of the x-coordinates of the nodes in a quadrilateral plate element into temporary arrays labeled X1, X2, X3 and X4. The coordinates of the nodes are stored in shared array AUX and the nodal locators are stored in local arrays N1, N2, N3 and N4 for nodes 1, 2, 3 and 4, respectively. L1 is the name of the lock which is associated with the memory locations containing the x-coordinates of all nodes. LL2(NID) is the number of elements in the block assigned to process NID.

```
      nlock(L1)
      DO 10 J = 1,LL2(NID)
        X1(J,NID) = AUX(N1(J,NID) + 1)
        X2(J,NID) = AUX(N2(J,NID) + 1)
        X3(J,NID) = AUX(N3(J,NID) + 1)
        X4(J,NID) = AUX(N4(J,NID) + 1)
10    CONTINUE
      nunlock(L1)
```

Several named locks are used; each corresponding to a different component of the nodal arrays used in the internal force calculations. By applying a lock to each component, the number of operations within a given lock monitor can be reduced, thus minimizing slowdown due to the locks. Note that the components of the nodal arrays are retrieved from global memory only once during the element block calculations. After they have been stored in temporary arrays, memory contention problems are eliminated.

Note also that in this example, the one dimensional arrays created for vectorization have been converted to two dimensional arrays. The added dimension is required to create a local memory for each process. Although some compilers may do this automatically in the future, these two dimensional arrays cannot be avoided in the Alliant FX/8 if direct control of the processors is needed. Thus in a vectorized, concurrent program, the coding in Box 2 of Section 3.3.2 becomes

BOX 3 : Trial Stress Computation in Vector-Concurrent Mode

```
      DO 10 J = 1,LL2(NID)
        SNEW1(J,NID) = SOLD1(J,NID) + SDEL1(J,NID)
        SNEW2(J,NID) = SOLD2(J,NID) + SDEL2(J,NID)
        SNEW3(J,NID) = SOLD3(J,NID) + SDEL3(J,NID)
   10 CONTINUE
```

The askfor monitor assigns an identification number NID to the process which indicates which process is performing the operations. The identification numbers range from 1 through the number of competing processes used in the problem solution. The askfor monitor then assigns a block of elements to each available process. Each process will perform the same calculations for different data. By dimensioning the temporary arrays as TEMP(NEPB,NPROCS) where NEPB is the number of elements per block and NPROCS is the number of processes, unique memory is created for each process.

3.5 Numerical Studies

Numerical studies were made to determine the speed-up on a multiprocessor due to both vectorization and concurrency. For comparisons, three versions of the program were used:

1. the original version of WHAMS run in scalar, serial mode : WHAM0,

2. the original version of WHAMS compiled using full optimization for concurrency and vectorization : WHAM_OPT

3. the vectorized, parallelized version of WHAMS using both full compiler optimization and monitors to control concurrency in the element calculations : WHAMS_VECPAR.

Four problems were considered:

1. a spherical cap loaded by a uniform pressure;

2. a pressurized containment vessel with a nozzle penetration;

3. an impulsively loaded cylindrical panel;

4. an automobile impact problem.

Results are presented in terms of the total run time for the problem and analyzed by the *speed-up* and *efficiency* of the program. Speed-up is defined as the ratio of the computing time of the program on a serial machine to the computing time on a parallel machine. The efficiency of the program is defined as the speed-up divided by the number of processors. Speed-up and efficiency are strongly influenced by the degree of parallelism and vectorization achieved in the program.

The first problem is the spherical cap shown in Figure 3.1. The material properties and parameters are listed in Table 3.1. The problem has 91 nodes and 75 elements and was run primarily to ensure that the vectorized version of the code gave the same results as the nonvectorized code. Because the problem is small, the elements were divided into 8 blocks in order to maximize the benefits of parallelization. However, with only 10 elements per block, the benefits due to vectorization were diminished. Table 3.2 compares the execution times of the three versions of WHAMS run on the Alliant FX/8 as well as execution times of WHAM0 on the VAX 11/780 and IBM 3033. A speed-up of 11 was achieved by the VECPAR version of WHAMS when compared with the scalar version WHAM0 on the Alliant.

The second problem studied was a pressurized containment vessel with a nozzle penetration shown in Figure 3.2. The problem has a wide range of element sizes and will presented again in Chapter 4 when subcycling is discussed. The problem has 344 elements and 407 nodes, and is subjected to a uniform pressure. The material properties and mesh dimensions are presented in Table 3.3 and timings are shown in Table 3.4.

Comparison of the run times of WHAM0 and WHAM_OPT shows that using compiler optimization for concurrency and vectorization provides a speed-up of almost three. This is only 3/8 of the speed-up which should be achieved by concurrency alone. However, by vectorizing the code and using monitors to control parallelism of the internal force vector, a total speed-up of 18.7 was achieved.

Three element block sizes were used for the WHAM_VECPAR version of the code using 1, 4 and 8 processors. Efficiencies due to parallelization were calculated by comparing run times using multiple processors with those using a single processor. Using four processors, the efficiencies for 12, 24 and 32 elements per block were 81%, 78% and 77%, respectively. With eight processors, the efficiencies decreased to 66%, 60% and 51%, respectively. The trend indicates that efficiencies decrease as the number of processors increase and as the number of elements per block increase. Note, however, that the efficiency due to vectorization increases as the number of elements per block increases. This illustrates the trade off between optimizing a code for vectorization and concurrency. These trends will be discussed in further detail in the next section.

The third problem is a 120 degree cylindrical panel shown in Figure 3.3 which is hinged at both ends and fixed along the sides. The panel is loaded impulsively with an initial velocity of 5650 in/sec over a portion of the shell. An elastic-perfectly plastic constitutive model was used with four integration points through the thickness. The material properties are shown in Table 3.5. Further details can be found in [15]. Due to symmetry only half of the cylinder was modeled. Three different uniform mesh discretizations were used so that the effects of problem size and element block size could be studied. Table 3.6 shows the number of elements and nodes for each mesh as well as time step used and total number of time steps.

All mesh discretizations were run using the three versions of WHAMS described previously. The results are presented in Tables 3.7 through 3.9. The VECPAR version of the program was run using 1,4 and 8 processors and various element block sizes.

A comparison of run times between the original version of the code, WHAM0, and the code using compiler optimization, WHAM_OPT, shows a speed-up of approximately 2.5 for all mesh discretizations. Comparing the original version of the code with the VECPAR version using eight processors gives speed-ups of 17.4, 24.2 and 26.4 for mesh 1, 2 and 3, respectively. Total speed-ups increase as the problem size increases.

Full advantage of vectorization on the Alliant FX/8 can be taken by using vectors of at least 24 elements in length. The optimum vector length is 32 which is the size of the vector registers. As the number of elements per block increases, the run times

decrease due to vectorization. However, the efficiency attributed to concurrency also decreases and the temporary storage which is required increases. In going from 1 processor to 4, the average speed-up is 3.53 with an efficiency of 88%, while with 8 processors, the average speed-up is 5.70 with an efficiency of 71%.

Assigning one element to a block eliminates the benefits of vectorization in the internal force calculations. The average speed-up achieved for this case using 4 and 8 processors was 3.92 and 7.55, yielding efficiencies of 98% and 94%, respectively. Run times for one element per block were quite high illustrating the fact that the execution of vectorized do loops with only a few loop iterations is slower than performing the operations in scalar mode.

The final problem is the simulation of a front end collision of an automobile. The mesh shown in Figure 3.4 models one-half of the front end of the vehicle. The nodes at the front of the model are subjected to an initial velocity consistent with an impact at approximately 60 miles per hour.

The mesh contains 1429 nodes and 1500 elements. The elements include 879 quadrilateral plate elements, 587 triangular plate elements and 34 beam elements. Plate thicknesses range from 0.7 to 2.0 mm. The material properties for the elastic-perfectly plastic constitutive model are given in Table 3.10.

The impact simulation was run for 20 msec. real time, requiring 40,000 time steps of 5.0E-7 sec. The computational times required for the three versions of the code are presented in Table 3.11. Note that the computational times are expressed in terms of CPU hours. This example illustrates the advantages of exploiting vectorization

and concurrency in real engineering design problems. The scalar version of the code requires 103 hours to complete the solution, while the vectorized-concurrent version using 8 processors requires only 4.4 hours.

The speed-up and efficiency calculated for the vectorized-concurrent version of the code were consistent with those found in the previous problems. For a block size of 32 elements, the speed-up for 4 processors was 3.66 giving an efficiency of 92 % and the speed-up for 8 processors was 5.66, yielding an efficiency of 71 %. A block size of 64 elements was also used, however, the results did not change significantly.

The decrease in efficiency as the number of processors increases is caused by factors inherent in the design of parallel algorithms. One factor is that processors remain idle when the number of tasks is smaller than the number of available processors. For example, the internal force calculations for 8 blocks of elements will take the same amount of time as 1 block of elements if eight processors are available. In the latter case, seven of the processors will be idle while the eighth performs the computations. The problem of processor idleness also illustrates an advantage to using relatively small element blocks (24 to 32 elements/block) as opposed to a few very large blocks. The more blocks that are available for computation, the less likely a processor will remain idle. Also, the larger the problem, the less significant processor idleness becomes. In terms of storage requirements, smaller block size is also preferable because as can be seen in Section 3.4, each single-dimensional array must be augmented to a two-dimensional array with the second dimension equal to the number of processes.

An algorithm was developed to select the optimal block size based on the criterion of minimizing processor idleness. The following flowchart illustrates the algorithm for a problem with several types of elements.

Flow Chart for Determining Optimal Block Size

1. Assume an optimal block size equal to the length of the vector registers. The Alliant FX/8 has vector registers of length 32.

2. Determine the number of blocks required for each element type.

$$\text{No. of blocks} = (\text{No. of elements} - 1)/32 + 1$$

3. Determine the average block size for each element type.

$$\text{Block size} = (\text{No. of elements} - 1)/(\text{No. of blocks}) + 1$$

4. Sum the number of blocks for all element types and determine the number of extra blocks required to fully utilize all processors. For example, if there are 8 processors and 30 element blocks, two extra blocks are required to utilize all processors.

5. In order to maximize block size for all elements, assign the extra blocks to element types which have the largest block size and recompute block size.

This algorithm was used to determine optimal block size for the problems considered in this section. For the three mesh discretizations of the cylindrical panel problem, the computed optimal block sizes for 8 processors were 12, 24 and 32 elements per block, respectively. Tables 3.7 through 3.9 show that the fastest run times for each discretization were those corresponding to the optimal block size. The optimal block size for the containment vessel problem was determined to be 22 elements per block. The run time corresponding to the optimal block size was 3 % smaller than the run time using 24 elements per block as shown in Table 4. Finally, the initial block sizes for the automobile impact problem, as computed using the formula in step 3 of the flow chart for the quadrilateral plate, triangular plate and beam elements, were 32, 31 and 17 elements per block, respectively. After optimizing the number of blocks, the block sizes for the three element types were modified to 28, 27 and 17 elements per block. However, run times did not change significantly. This is due to the fact that the number of calculations associated with the beam element is much smaller than those required by the plate elements. Therefore, using 32 elements per block for the plate elements yields 47 element blocks which is nearly optimal for the problem. Also as the problem size increases, the relative time that processors remain idle is minimized. This procedure can also be used for computers with longer optimal vector lengths such as the CRAY-1 which has a vector length of 64.

Another factor which decreases efficiency is memory contention. If more than one processor attempts to access a shared memory location simultaneously, an error

will occur. This happens in internal force calculations when two elements in different blocks share a node. A "lock" monitor must be used to ensure that only one processor will access the memory location at a time. However, the locks cause a slowdown if substantial interference exists.

Probably the most significant factor for the decreasing efficiency with the number of processors is the effect of nonparallelizable computations. Once the most time consuming computations have been effectively recoded for parallel computations, other portions of the program require an increasingly larger fraction of the computation time. These sections of the code may not be conducive to parallel execution and will prevent further speed-up.

To illustrate this effect, the program was divided into three parts: the calculations performed before, during and after the element internal force computation. The first part included the calculation of the external force array as well as the update of the nodal coordinates. The second part was comprised of all calculations listed in step 3 of the flow chart in Section 2.2. The final section included the computation of the accelerations and the update of the velocity and displacement vectors. The times required for each section was monitored for the cylindrical panel problem with 24 elements per block and are presented in Table 12 for 1, 4 and 8 processors. The speed-up (efficiency) calculated for the internal force calculations was 6.1 (76%), whereas the speed-up (efficiency) of the computations before and after these calculations was 2.5 (31%) and 4.7 (60%), respectively. The average speed-up and efficiency for the time step was 5.8 and 73%, respectively. Furthermore, as the number of processors

increase, the percentage of time spent in the internal force calculations decreases. Therefore, the less efficient coding takes up an increasingly greater percentage of the total execution time.

3.6 Conclusions

Vectorization and concurrency significantly speed up the execution of explicit finite element programs. However, these techniques tend to be competing processes. To optimize a code for concurrency, element block size should remain small in order to minimize processor idleness. On the other hand, to optimize a code for vectorization, the block size should be at least one to two times the size of the computer's vector registers. As the number of elements per block increases, the efficiency of the concurrent execution decreases. The element block size for a vectorized-concurrent execution depends primarily on the number of elements in the mesh. For small problems (less than 200 elements) the block size can be reduced to approximately 15 elements without eliminating the benefits of vectorization. Element blocks with only a few elements will require more computation time in vectorized execution than in scalar execution. For larger problems, the block size should be chosen to correspond to the size of the vector register. Note that as the number of elements in the problem increases, the effect of element block size diminishes.

Efficiency of concurrent execution also decreases as the number of processors increase due to processor idleness, memory contention problems and the effect of

nonparallelizable code. Processor idleness can be reduced by using smaller block size as mentioned previously, or by employing fewer processors for the problem solution. An algorithm described here optimizes block size and has been shown to give the best running times. In parallel computers, memory contention is also an issue. Memory contention problems within an element block can be eliminated by a sorting algorithm used in dividing elements into blocks. However, for memory contention problems between processes, a lock mechanism which prevents simultaneous access to global memory locations is probably unavoidable since it is almost impossible to schedule processes so that no common data is updated.

Table 3.1: Material Properties and Parameters for Spherical Cap

Radius	r	=	22.27 in
Thickness	t	=	0.41 in
Angle	α	=	26.67 deg
Density	ρ	=	2.45×10^{-4} lb-sec^2/in^4
Young's modulus	E	=	1.05×10^7 psi
Poisson's ratio	ν	=	0.3
Yield stress	σ_y	=	2.4×10^4 psi
Plastic modulus	E_p	=	2.1×10^5 psi
Pressure load	P	=	600 psi
Time Steps	N	=	1000

Table 3.2: Solution Times for Spherical Cap Problem

Alliant - WHAM0	310.9	sec
Alliant - WHAM_OPT (8 Procs.)	116.5	sec
Alliant - WHAM_VECPAR (8 Procs.)	28	sec
VAX 11/780 - WHAM0	901.8	sec
IBM 3033 - WHAM0	75	sec

Table 3.3: Material Properties and Parameters for Containment Vessel

Vessel diameter	d_v	=	264.0 in
Vessel height	h	=	399.0 in
Penetration diameter	d_p	=	40.0 in
Penetration length	l	=	29.3 in
Thickness	t	=	0.25 in
Density	ρ	=	7.5×10^{-4} lb-sec^2/in^4
Young's modulus:			
Nozzle	E	=	40.0×10^7 psi
Pressure vessel	E	=	3.0×10^7 psi
Collar	E	=	9.0×10^7 psi
Poisson's ratio	ν	=	0.3
Yield stress	σ_y	=	6.01×10^4 psi
Plastic modulus	E_p	=	4.4×10^4 psi

Table 3.4: Run Times (Efficiency) for Containment Vessel Problem (in CPU Sec.)

Program Version	Number of Processors	12 elements per block	24 elements per block	32 elements per block
WHAM0	1	3768		
WHAM_OPT	8	1291		
WHAM_VECPAR	1	1167	959	907
	4	356(81%)	309(78%)	295(77%)
	8	222(66%)	201(60%)	223(51%)

Table 3.5: Parameters for Cylindrical Panel Problem

Density	ρ	=	2.5×10^{-4}lb-sec^2/in^4
Young's modulus	E	=	1.05×10^7 psi
Poisson's ratio	ν	=	0.33
Yield stress	σ_y	=	4.4×10^4 psi
Plastic modulus	E_p	=	0.0 psi

Table 3.6: Sizes and Time Steps for Mesh Discretizations for Cyl. Panel Problem

Mesh	No. Elements	No. Nodes	Time Step	No. Steps
1	96	119	2.0E-6 sec	500
2	384	429	1.0E-6 sec	1000
3	1536	1625	0.5E-6 sec	2000

Table 3.7: Run Times (Efficiency) for Cylindrical Panel Problem - Mesh 1
96 Elements ; 500 Steps (in CPU Sec.)

Program Version	Number of Processors	1 element per block	12 elements per block
WHAM0	1	347	
WHAM_OPT	8	141	
WHAM_VECPAR	1	529	103
	4	137(97%)	31(83%)
	8	74(89%)	20(64%)

Table 3.8: Run Times (Efficiency) for Cylindrical Panel Problem - Mesh 2
384 Elements ; 1000 Steps (in CPU Sec.)

Program Version	Number of Processors	1 element per block	12 elements per block	24 elements per block	32 elements per block
WHAM0	1	2658			
WHAM_OPT	8	1072			
WHAM0_VECPAR	1	4189	785	631	594
	4	1071(98%)	213(92%)	176(90%)	168(88%)
	8	552(95%)	126(78%)	110(72%)	125(59%)

Table 3.9: Run Times (Efficiency) for Cylindrical Panel Problem - Mesh 3
1625 Elements; 2000 Steps (in CPU Sec.)

Program Version	Number of Processors	1 element per block	12 elements per block	24 elements per block	32 elements per block
WHAM0	1	20860			
WHAM_OPT	8	8484			
WHAM0_VECPAR	1	34495	6030	4807	4496
	4	8470(100%)	1671(90%)	1391(86%)	1275(88%)
	8	4364(99%)	939(80%)	812(74%)	789(71%)

Table 3.10: Material Properties and Parameters for Auto Impact Problem

Density	ρ	=	7.835×10^{-5}N-sec^2/cm^4
Young's modulus	E	=	$2.0 \times 10^7 N/\text{cm}^2$
Poisson's ratio	ν	=	0.28
Yield stress	σ_y	=	$2.0 \times 10^4 N/\text{cm}^2$
Plastic modulus	E_p	=	$0.0 N/\text{cm}^2$
Pressure load	P	=	$600 N/\text{cm}^2$
No. of Time Steps	N	=	40000
Time Step	Δt	=	5.0×10^{-7} sec.

Table 3.11: Run Times (Efficiency) for Automobile Impact Problem (in CPU hrs)

Program Version	Number of Processors	32 element per block	64 elements per block
WHAM0	1	103.1	
WHAM_OPT	8	41.3	
WHAM_VECPAR	1	24.9	24.4
	4	6.8(92%)	7.2(85%)
	8	4.4(71%)	4.4(69%)

Table 3.12: Comparison of Computation Times For a Single Time Step (in CPU Sec.)

Section	1 Proc.	4 Procs.	8 Procs.	Speed-up
Before f_{int}	.0032	.0017	.0013	2.46
During f_{int}	.5364	.1483	.0883	6.07
After f_{int}	.0714	.0224	.0150	4.76
Total	.6110	.1724	.1046	5.84

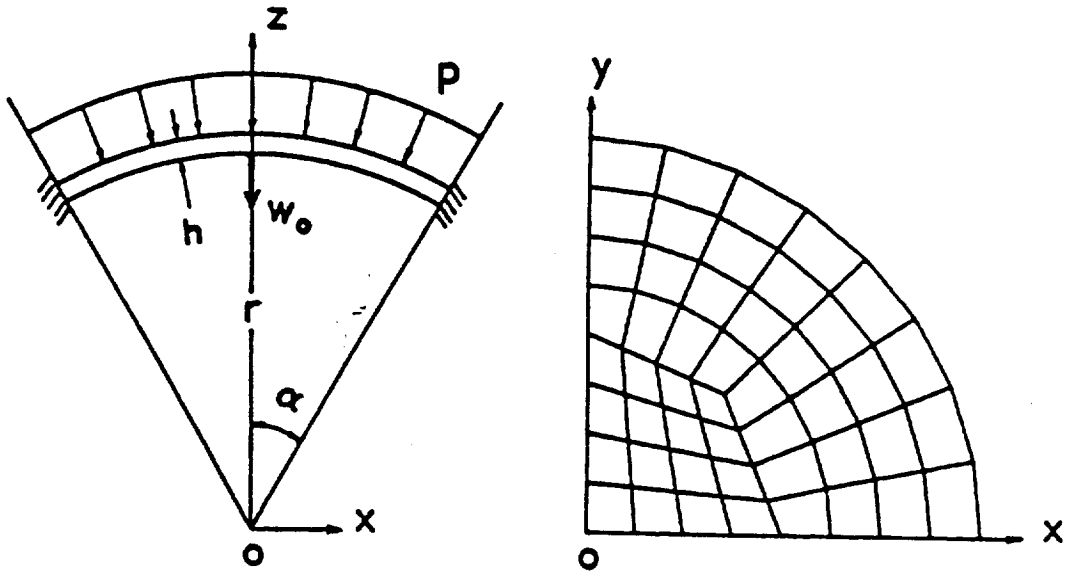


Figure 3.1: Sample Problem 1: Spherical Cap

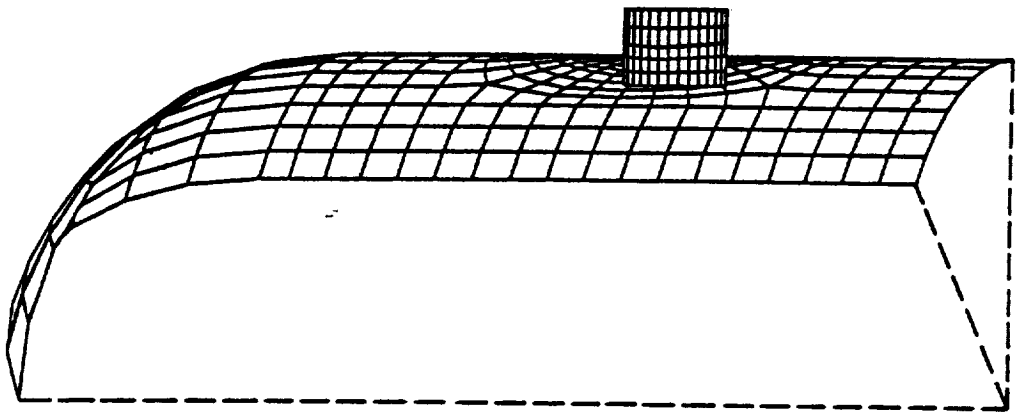


Figure 3.2: Sample Problem 2: Containment Vessel with Nozzle Penetration

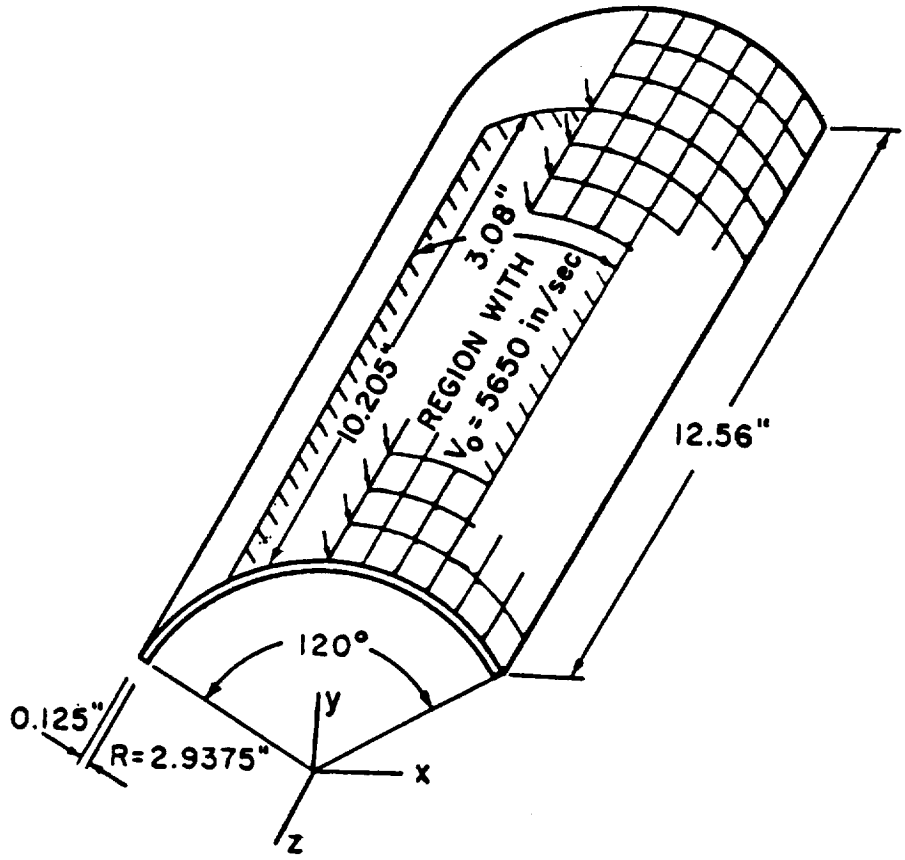


Figure 3.3: Sample Problem 3: Impulsively Loaded Panel

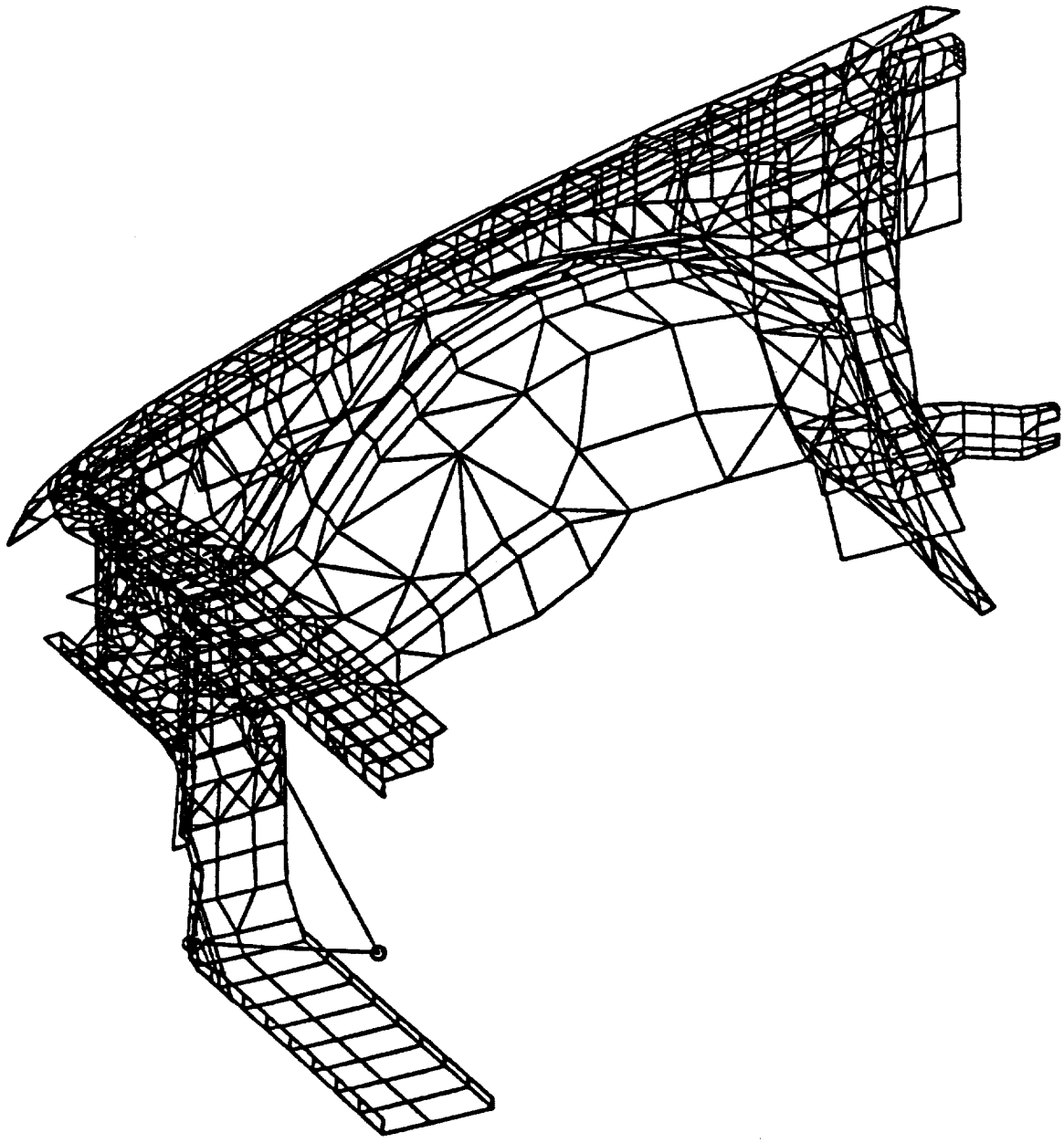


Figure 3.4: Sample Problem 3: Automobile Impact Problem

Chapter 4

Subcycling

4.1 Subcycling Formulation

Subcycling is a mixed time integration method which uses the same integrator but different time steps for different parts of the mesh. When an explicit integrator is used with different time steps, it is sometimes called explicit-explicit partitioning, since the mesh is partitioned into *subdomains* with different time steps. Mathematicians sometimes call these methods subdomain techniques. The explicit-explicit partitioning procedure used here was first presented in [1, 5, 8]. Substantial improvement in computational efficiency can be achieved when using mixed time integration methods on problems which contain elements of varying sizes and material properties. Subcycling techniques allow each element or group of elements to be integrated using a time step close to its critical time step. Without subcycling, all elements in the mesh must be integrated using the smallest critical element time step.

4.1.1 Finite Element Equations

We assume that the domain of the structure, Ω, is subdivided into elements. The element domains are designated by Ω_e, where the subscript e indicates an element-related variable. The structure of the finite element equations is as follows:

Strain-displacement equations (in rate-form):

$$\boldsymbol{d} = \boldsymbol{B}\,\boldsymbol{v}_e \tag{4.1}$$

Constitutive equations (in rate form):

$$\overset{\nabla}{\boldsymbol{\sigma}} = \boldsymbol{\mathcal{S}}(\boldsymbol{\sigma}, \boldsymbol{d}) \tag{4.2}$$

Momentum equations:

$$\boldsymbol{f}^e_{int} = \int_{\Omega^e} \boldsymbol{B}^T \boldsymbol{\sigma}^e \, d\Omega \tag{4.3}$$

$$\boldsymbol{a} = \boldsymbol{M}^{-1} \left(\boldsymbol{f}_{ext} - \boldsymbol{f}_{int}\right) \tag{4.4}$$

where,

$\boldsymbol{B}$ = strain-rate velocity matrix,

$\boldsymbol{d}$ = velocity strains (strain rates),

$\boldsymbol{f_{int}}$ = internal nodal forces,

$\boldsymbol{f_{ext}}$ = external nodal forces,

$\boldsymbol{M}$ = mass matrix (assumed diagonal and lumped),

$\boldsymbol{\sigma}$, $\overset{\nabla}{\boldsymbol{\sigma}}$ = Cauchy stress matrix and its frame invariant rate,

$\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{a}$ = nodal displacement, velocity and acceleration vectors, respectively,

$\boldsymbol{\mathcal{S}}$ = the constitutive equation (or algorithm) and frame invariance correction.

The central difference method uses the following equations to update the nodal displacements and velocities in time:

$$\begin{aligned} \boldsymbol{v}^{n+\frac{1}{2}} &= \boldsymbol{v}^{n-\frac{1}{2}} + \Delta t^n \, \boldsymbol{a}^n \\ \boldsymbol{u}^{n+1} &= \boldsymbol{v}^n + \Delta t^{n+\frac{1}{2}} \, \boldsymbol{v}^{n+\frac{1}{2}} \\ \Delta t^n &= \tfrac{1}{2}\left(\Delta t^{n-\frac{1}{2}} + \Delta t^{n+\frac{1}{2}}\right) \end{aligned} \tag{4.5}$$

The superscripts in the above equations designate time steps. The fractional subscripts indicate a midstep value. Note that an average time step is used to update the velocities. This provides the capability of changing the time step during the solution.

An outline of the explicit time integration algorithm with a single time step for all elements in the mesh is given below. As can be seen from the flow chart, the major opportunity for parallelization appears in the loop over the elements. The number of multiplications per element can vary from 50 to the order of 10^3. Since the internal force vector for each element is independent during a given time step in an explicit code, these calculations form tasks which yield a coarse-grained parallelism which is ideal for concurrent processors. However, if the parallelism is exploited on an element level, then the opportunities for any significant vectorization are lost. To exploit vectorization in conjunction with parallelism, it is necessary to arrange the elements in groups. The number of groups should exceed the number of processors, but the group size is limited by the auxiliary arrays which are needed for vectorization. Furthermore, in an efficient implementation of vectorization, all elements in a group must be of the same type.

Flow Chart for Explicit Integration

1. Initial conditions : $v^{-\frac{1}{2}}, x^o$

2. Compute external force

3. Compute internal force vector f_{int}^{n+1}

   Loop over element groups

   (a) compute velocity strains by Eqn. (4.1)

   (b) evaluate constitutive law, Eqn. (4.2)

(c) evaluate $B^T \sigma$ and add to integrand of Eqn. (4.3)

(d) add $f_{int,e}$ to total f_{int} array

4. Compute accelerations by equation of motion : Eqn. (4.4)

5. Update velocities and displacements using central difference equations: Eqn. (4.5)

6. Go to 2.

4.1.2 Implementation of Subcycling

The explicit-explicit partitioning procedure is implemented by dividing the elements into groups according to their critical time steps. Each element group can then be integrated with a different time step subject to the following restrictions:

1. The largest group time step must be an integer multiple of all time steps.

2. If any node is shared by elements in two different integration groups, the time steps of the groups must be integer multiples of one another.

In this work, the first restriction has been modified by requiring that all group time steps be integer multiples of each other. This additional constraint eliminates the need to keep track of interface nodes and allows elements be assigned to groups according to their critical time steps and not physical proximity. In adapting this method to a parallel computer, it was decided to make the alignment between the

element groupings for vectorization and the time step partitioning coincident. The element grouping is performed by a preprocessor.

For the purpose of describing the explicit-explicit partitioning procedure, the following variables are defined:

NBLOCK: number of groups into which the finite element mesh is subdivided,

Δt_G : the time increment for element group G,

Δt_N : the time increment for node N,

Δt : the time increment corresponding to the minimum Δt_G,

Δt_{mast} : the master time increment, which corresponds to the maximum Δt_G,

$time$: the current time,

t_{mast} : the master time,

t_N : the nodal clock,

t_G : the element group clock.

Time steps Δt_N and Δt_G are assigned to nodes and element groups, respectively, at the beginning of the solution and may be modified during the first subcycle of any master time step. The master time step is set to the largest element group time step

and is used to determine when a cycle has been completed. The critical time steps may change as the element deforms or if the material properties change with time. However, in this procedure, the time steps are not allowed to change during a cycle.

In order to assign nodal and element group time steps, the critical time step for each element is calculated. These time steps are then converted to an integer multiple of the smallest element time step and stored in the array NLIM(1:NELE) where NELE is the number of elements in the mesh. The integer multiples in NLIM are adjusted to satisfy the restriction that all time steps must be integer multiples of one another.

The critical time steps are calculated by the following procedure. For each node N, a preliminary time step is computed by

$$\Delta t_N = \min_e \left(\frac{2}{\omega^e_{max}} \right)$$

for all elements e connected to node N. After all Δt_N are computed, the element time step is determined by

$$\Delta t_e = \min_n (\Delta t_n) \tag{4.6}$$

for all nodes n of element e. The nodal time steps are then computed by

$$\Delta t_N = \min_e (\Delta t_e) . \tag{4.7}$$

The elements are sorted in terms of increasing time step. An efficient sort algorithm involves creating a two-dimensional table NTEMP(1:NELE,1:NMAX) where NMAX is the largest entry in the NLIM array, i.e., the number of subcycles required to complete a master time step. The element numbers are placed in the column corresponding to their calculated time step ratio (NLIM). The array INBLK(1:NMAX) keeps track of how many elements are in each column. After the table has been completed, the element numbers are reassigned to array NSORT(1:NELE) in a column-wise progression, resulting in an array of sorted elements with time steps in ascending order.

The elements are divided into groups of NEPB, where NEPB is the number of elements per group which is selected by the user. For large problems, NEPB is usually chosen to equal the length of the vector registers in order to achieve maximum benefit from vectorization. However, when using subcycling techniques, smaller groups may be more efficient. In a parallelized code, it is important to minimize processor idleness. Since the internal force vector is computed for a subset of the element groups during a subcycle, the likelihood of idle processors increases. Smaller group size increases the total number of groups during each subcycle, thus improving the efficiency due to concurrency. Minimizing processor idle time will be discussed in detail later.

In addition to time steps, a clock is assigned to each node and element group. These clocks, when compared to the current time, indicate when a node or group of elements is ready to be updated. At the beginning of each subcycle, the nodal

and element group clocks are advanced for those nodes and element groups which were updated in the previous subcycle. By design, all nodes and element groups are updated in the first subcycle (although only the nodes *must* be updated as discussed in Section 4.4).

Once the nodal and element group clocks have been updated, the integration procedure continues as described in the flow chart in Table 4.1. For each element group whose clock is behind the current time, i.e,

$$t_G \leq time \tag{4.8}$$

new velocity strains, stresses and internal forces are computed. The element internal forces are then added into the global internal force array.

After all element groups which satisfy Eqn. 4.8 have been updated, the nodal loop is executed. For each node whose clock is behind the current time, i.e,

$$t_N \leq time \tag{4.9}$$

accelerations, velocities and displacements are computed and nodal constraints are applied.

The algorithm assumes that a velocity strain formulation is used for all element calculations. When an element needs to be updated, the latest available velocity is used to compute the velocity strain. This means that if an element is connected to a

node with a larger time step, it uses the same nodal velocity for all intermediate time steps. This corresponds to a constant velocity interpolation or a linear displacement interpolation. If displacements at a node are needed by an element at a time when the node is not being updated, linear interpolation based on the last cycle displacement with the current velocity as a slope is used. Note that a node which has a larger time step than the element will never be behind the element (i.e., its clock will never be behind) because the larger time step nodes are integrated first.

The subcycling procedure continues until all nodes and element groups have been updated to the master time. After the final subcycle, all nodal variables have been updated to the same point in time and results can be output. Critical time steps of the elements and nodes are then recalculated and the elements are resorted before the next time step. An outline for the subcycling procedure is given in Table 4.1.

Table 4.1: Flow Chart for Subcycling Algorithm

1. Initial conditions : $\boldsymbol{v}^{-\frac{1}{2}}, \boldsymbol{x}^{o}$

2. Initialize nodal and element group clocks

   (a) $t_G = 0$ for all element groups

   (b) $t_N = 0$ for all nodes

   (c) time = 0

3. Update time and clocks; if necessary resort elements and reassign element groups

(a) Subcycle 1

i. if required, resort elements (See Section 4.1.2)

ii. update all nodal and element group clocks

iii. update time

(b) Subcycle n, n > 1

i. update nodal clocks if $t_N <$ time

ii. update element group clocks if $t_G <$ time

iii. update time

4. Impose external force

5. If $t_G <$ time, compute $\boldsymbol{f}_{int}$ for element group G

(a) compute velocity strains

$$\boldsymbol{d}^{n+\frac{1}{2}} = \boldsymbol{B}\,\boldsymbol{v}^{n+\frac{1}{2}} \tag{4.10}$$

(b) compute frame invariant stress rates

$$\overset{\nabla}{\boldsymbol{\sigma}}^{n+\frac{1}{2}} = \boldsymbol{\mathcal{S}}(\boldsymbol{\sigma}, \boldsymbol{d}) \tag{4.11}$$

(c) obtain time derivative of Cauchy stress

$$\dot{\boldsymbol{\sigma}}^{n+\frac{1}{2}} = \overset{\nabla}{\boldsymbol{\sigma}}^{n+\frac{1}{2}} + \boldsymbol{W}^{n+\frac{1}{2}} \cdot \boldsymbol{\sigma}^{n} - \boldsymbol{\sigma}^{n} \cdot \boldsymbol{W}^{n+\frac{1}{2}} \tag{4.12}$$

where $\boldsymbol{W}$ is the spin tensor.

(d) update stress

$$\boldsymbol{\sigma}^{n+1} = \boldsymbol{\sigma}^{n} + \Delta t \, \dot{\boldsymbol{\sigma}}^{n+\frac{1}{2}} \tag{4.13}$$

(e) compute element internal nodal force : Eqn. 4.2

(f) assemble into global array

6. If $t_N <$ time, compute acceleration, velocity, displacements and apply nodal constraints

7. If time $< t_{mast}$, go to 3b

8. If time $< t_{final}$, output, go to 3a.

4.1.3 Graphical Representation of Subcycling

Figure 4.1 illustrates the subcycling procedure for three element groups of one element each with critical time steps of Δt, $4\,\Delta t$ and $4\,\Delta t$, respectively. The nodes are represented by solid circles. Time is represented by the vertical axis. Note that the time step of node 2 is Δt because that is the minimum critical time step of any element connected to it. During the first subcycle, all nodal and element group

clocks are set to t while the current time is updated to $t + \Delta t$. Therefore, all nodal and element clocks are behind the current time, so all nodes and element groups are updated in subcycle 1. In Figure 4.1, a vector signifies a nodal update and a σ signifies an element update. The number associated with the symbol indicates the subcycle in which the node or element was updated. In subcycle 2, the nodal clocks for nodes 1 and 2 are updated to $t + \Delta t$ while the nodal clocks for nodes 3 and 4 are updated to $t + 4\Delta t$. The element group clocks for groups 1, 2 and 3 are updated to $t + \Delta t$, $t + \Delta t$ and $t + 4\Delta t$, respectively. The time is then updated to $t + 2\Delta t$. During the second subcycle, the clocks for element groups 1 and 2 and nodes 1 and 2 are behind the current time, so only these updates will be performed. In subcycles 3 and 4, element groups 1 and 2 and nodes 1 and 2 are again updated so by the end of subcycle 4 all nodes and element groups have been updated to $t + 4\Delta t$ and the cycle is complete.

As discussed previously, a velocity strain formulation which corresponds to a constant velocity interpolation is used to compute the element internal force vector. When an element is updated, the latest available velocity is used to compute the velocity strain. When node 2 is updated to $t + \Delta t$, the velocity is assumed constant from time $= t$ to time $= t + \Delta t$. In the same manner the velocity of node 3 is constant from time $= t$ to time $= t + 4\Delta t$. If element 2 is updated using a time step of $4\Delta t$, the velocity strain would be calculated using the velocity from node 2 at time $= t + \frac{1}{2}\Delta t$ and from node 3 at time $= t + 2\Delta t$. However the velocity at node 2 changes in each subcycle so the computed internal force for element 2 would

be incorrect. If element 2 is updated using a time step of Δt, the most current velocity will always be used in the internal force calculations.

4.1.4 Stability

Stability of mixed method integration techniques is proven for first order ordinary differential equations emanating from FEM semidiscretizations of linear systems in [6, 28]. Since no general proofs of stability are available for second order systems such as structural dynamics, numerical studies were made. In this study of the stability of explicit-explicit partitioning, four discretizations of an axially loaded rod were analyzed. All discretizations consisted of thirty 4-node Mindlin plate elements which were divided into three groups of ten elements each. The first discretization contained elements of uniform size, while the others had varying element size. The following table shows the time step used to compute the internal force vector for each element group in the four meshes. The time step Δt_{cr} corresponds to the critical time step of the smallest element in the mesh as computed by Eqn. 3.11. The critical time step is reduced by 20% to compensate for potential destabilizing effects due to nonlinearities. The size of the remaining elements was selected to yield time steps which are integer multiples of the smallest time step Δt.

Mesh	Δt_{cr}	Δt	Group 1	Group 2	Group 3
1	0.464	0.370	Δt	Δt	Δt
2	0.0507	0.0440	$16\Delta t$	$4\Delta t$	Δt
3	0.0346	0.0270	$25\Delta t$	$5\Delta t$	Δt
4	0.0098	0.0078	$100\Delta t$	$10\Delta t$	Δt

The mesh dimensions are shown in Figure 4.2. The total length of the rod is fifteen inches in all meshes so that the displacements and stresses at a given point can be compared. Note that there are eleven elements of the largest size and only nine elements of the smallest size in the meshes containing varying element sizes. Element 11 is updated using the time step of element 12 and element 21 is updated using the time step of element 22. This results in the formation of three element groups where all elements in a group have the same time step. Both elastic and elastic-plastic materials were considered. The applied loads and material properties are listed in Table 4.2.

Figures 4.3 through 4.6 show the plots of displacement and stress for the elastic and plastic analyses. The displacement at node 1 (at free end of beam) and the stress at element 30 (at fixed end of beam) were plotted verses time for the four mesh discretizations. Also plotted are the analytical solutions. In all cases, the numerical results agree well with the analytical results. In the elastic analysis (Figures 4.3 and 4.4), the curves for both displacement and stress were nearly identical for all cases

considered.

Figures 4.5 and 4.6 show the plots of displacement and stress, respectively, for the elastic-plastic rod. The curves for the stresses are very consistent for the four mesh discretizations. However, the displacements obtained with subcycling are approximately two percent less than the non-subcycling analysis. The large variation in size for elements in the mesh used for subcycling seems to be a factor in this discrepancy. The mesh for case 2 ($16\Delta t - 4\Delta t - 1\Delta t$) was run using a uniform time step for all elements. The resulting displacement at Node 1 was 2 percent higher than the displacement for the uniform mesh. Other discretizations showed similar differences.

These studies indicate that subcycling is stable for a reasonable range of time steps. In the code presented in this dissertation, the maximum time step allowed for the subcycling analysis is 64 times the minimum time step.

4.1.5 Speed-up Due to Subcycling

The maximum theoretical speed-up which can be achieved by subcycling can be calculated for a problem based on the normalized time required to update all nodes and element groups. Let the time required to solve a problem without subcycling be given by:

$$T^{ns} = \#\text{time steps} \times 1.0 \tag{4.14}$$

The minimum computer time required for subcycling can be calculated in terms of the number of subcycles, NSUB, and the percentage of elements being updated at

each subcycle, $PCTS_i$.

$$T^s = \frac{\#\text{time steps}}{NSUB} \times [1.0 + \sum_{i=1}^{NSUB-1} PCTS_i] \tag{4.15}$$

For example, if 10 subcycles are required to integrate 25% of the elements, the minimum computer time is:

$$T^s = \frac{\#\text{time steps}}{10} \times [1.0 + 9 \times 0.25] \tag{4.16}$$

The optimal speed-up due to subcycling is simply the ratio between the two times

$$\text{Speed-up} = \frac{T^{ns}}{T^s} \tag{4.17}$$

which for this example is 3.08.

4.2 Vectorization Considerations

Initial timings for problems using subcycling on a vectorized, concurrent computer yielded relatively poor speed-ups compared to the maximum theoretical speed-up. In the early version of the code, only the internal force calculations were vectorized because they required the majority of the computational time. The nodal calculations were not modified, so any improvement in speed for the nodal updates was due only to the vectorization of existing loops. The vectorization of the internal force calculations

is not affected by subcycling because they are performed only for the element groups requiring update.

However, the nodes are not divided into groups, so each node must be checked in every calculation to determine whether it should be updated. For example, the update of the nodal velocities and displacements was computed by the following loop where NEQ is the number of degrees of freedom:

```
C
C       UPDATE VELOCITIES AND DISPLACEMENTS
C
        DO 150 I = 1,NEQ
          IF (CLKNOD(I) .LE. TIME) THEN
            V(I) = V(I) + A1(I)*DELT
            X1(I) = XO(I) + V(I)*DELT
          ENDIF
  150 CONTINUE
```

The process of checking the status of the each node as well as the nonvectorized coding significantly reduced the speed-up due to subcycling.

The nodal calculations were reorganized in order to optimize the benefits of vectorization. In the above nodal loop, an IF statement is executed to determine whether the node should be updated. In order to vectorize the loop, the compiler must perform a gather operation to separate the nodes which require update from those which do not. This gather operation slows down the execution of the loop.

Efficiency was achieved in the nodal calculations by performing the gather operation explicitly outside of the loops. An array NUPD, with length NNUPD, was created to store all nodes which require updating in the current subcycle. This array

was formed at the beginning of each subcycle. The following code illustrates the formation of the NUPD array.

```
C
C     STORE NODES REQUIRING UPDATES IN ARRAY NUPD
C
      NNUPD = 0
      DO 100 J = 1,NNODE
        IF (CLKNOD(J) .LE. TIME) THEN
          NNUPD = NNUPD + 1
          NUPD(NNUPD) = J
        ENDIF
  100 CONTINUE
```

Subsequent calculations on nodal variables are then performed only on the nodes requiring an update and all IF statements are removed from the do loops. The nodal calculations take the form of:

```
C
C     UPDATE VELOCITIES AND DISPLACEMENTS
C
      DO 150 J = 1,NNUPD
        NUP = NUPD(J)
        V(I) = V(I) + A1(I)*DELT
        X1(I) = X0(I) + V(I)*DELT
  150 CONTINUE
```

These modifications to the nodal calculations increased the speed-up due to subcycling by 50%.

4.3 Timing Studies

Two problems presented in Chapter 3 were used to study the behavior of subcycling in a parallel environment:

1. pressurized containment vessel

2. automobile impact problem.

The pressurized containment vessel with a nozzle penetration is shown in Figure 3.2 and the material properties and mesh dimensions are listed in Table 3.3. This problem contains elements of various sizes and material properties, so the computed element time steps will differ significantly from element to element. In the solution presented in Chapter 3, all elements were integrated using the time step corresponding to the smallest element.

The computed time steps for the elements in the containment vessel ranged from 3.8E-6 sec to 5.6E-5 sec. The largest integer ratio of element time steps is 14. In order to satisfy the restriction that the largest time step be an integer multiple of all time steps, the maximum integer ratio was reduced to 8.

The elements were divided into groups using 12, 24 and 32 elements per group. The time step assigned to the group corresponds to the smallest of the element time steps. Table 4.3 lists the number of subcycles required for each group of elements and the corresponding time step for each group.

Table 4.4 shows the run times and efficiencies for the containment vessel problem run with and without subcycling. Efficiency due to parallelization decreases as the

number of processors increases and as the size of the element groups increases. This trend was also seen in the problems discussed in Chapter 3. However, speed-up due to subcycling, computed as the ratio between the run time using mixed time integration and the run time using a single time step, no longer follows this trend. Using four processors, the speed-up due to subcycling increases as the group size increases. However, with eight processors, the speed-up decreases as the group size increases. This decrease in speed-up results from the fact that with a large number of processors and relatively few groups of elements, processors will be idle while groups with smaller time steps are subcycling.

To illustrate this, Table 4.3 shows the number of groups integrated using the smallest time step Δt. For 12, 24 and 32 elements per group, the number of groups is 8, 4 and 3, respectively. These groups are updated every subcycle and for half of the subcycles, these are the only groups which are updated. Using 12 elements per group and 8 processors, eight groups are updated resulting in no idle processors. However, using 32 elements per group, only 3 groups are being updated in one cycle, so 5 processors remain idle. When a smaller group size is used, the number of groups increases, thus decreasing the number of idle processors. Note, that the containment vessel problem is not a large problem. As the size of the problem increases, the number of element groups increase and the effect of processor idleness diminishes. An algorithm for minimizing processor idleness is presented in Section 4.4.

Another disadvantage of large groups when subcycling is that the time step assigned to each group is smaller since the group time step must be below the critical

time step of all elements in the group. With larger groups, the number of elements integrated using an unnecessarily small time step increases. Therefore, the benefits of subcycling are reduced.

The mesh for the automobile impact problem is shown in Figure 3.4 with material properties given in Table 3.10. Adjusted element time steps ranged from 4.9E-7 to 7.84E-6 sec., giving an maximum integer time step ratio of 16. For this problem, however, most of the element blocks required 2 or 4 subcycles per master time step.

The run times and efficiencies due to parallelization for the automobile frontal crash problem are shown in Table 4.5 for block sizes of 32 and 64 elements. Trends described in the discussion of the containment vessel results are consistent with the results of this problem. The first trend is that the efficiency due to parallelization decreases as the number of processors and the element block size increases. This conclusion has been supported by the results of all problems studied in this report.

The second trend is that the speed-up due to subcycling also decreases with the number of processors. For the auto problem, the speed-up due to subcycling is 2.43 and 2.15 for 1 processor and 4 processors, respectively. Unlike, the containment vessel problem, the block size has little effect on the speed-up due to subcycling when 4 or less processors were used. This is due to the fact that as the problem size increases, the problem of processor idleness diminishes. However, for 8 processors, the speed-up due to subcycling is 2.04 for an element block size of 32 and 1.76 for an element block size of 64. The reduction in speed-up due to the increase in block size is primarily due to processor idleness.

By comparing the run times of the original version of the code with the vectorized-concurrent version of the code, a total speed-up of 47.7 is realized. A speed-up of almost 20 is achieved by the improvements due to vectorization modifications, monitors and subcycling over the compiler optimized version.

4.4 An Efficient Allocation Algorithm for Concurrency and Subcycling

Consider a discretization of the 120 degree cylindrical panel shown in Figure 3.3 in which 1000 elements have a critical time step of 1.0 x 10^{-4} sec and 20 elements have a critical time step of 1.0 x 10^{-5} sec. Elements with the small time step will require 10 subcycles for every cycle of the larger elements. Assume the elements are divided into groups of twenty, giving 1 group of elements with a time step of Δt and 50 groups of elements with a time step of $10\,\Delta t$. Table 4.6 shows how the processors are allocated during each subcycle. Groups are indicated in the table by their time step. During the first subcycle, all 51 groups are updated. However, during the remaining nine subcycles, only one group is updated, leaving seven processors idle. This inefficient use of multiple processors reduces the benefit achieved by subcycling.

In order to minimize the effects of processor idleness, the algorithm was modified by determining which element groups could be updated early if a processor was expected to remain idle during a subcycle. The modification is as follows. Within each subcycle, the nodal clocks are compared with the current time. If the nodal clock

is behind the current time, a new acceleration is computed and displacements and velocities are updated. Before accelerations can be computed for a node, the internal force vectors for all elements containing that node must be updated. However, it is not necessary for the internal force vector to be updated during the same cycle that the node is updated. It is only necessary that the elements be updated within the nodal time step. For example, suppose a node has a nodal time step of $4\,\Delta t$, where Δt corresponds to the smallest element group time step, and an element containing the node also has a time step equal to $4\,\Delta t$. The internal force vector for the element can be updated in any of the four subcycles occurring before the nodal update.

Figure 4.7 gives a graphical representation of the algorithm for a problem consisting of four elements with critical time steps of $\Delta t, 2\,\Delta t, 4\,\Delta t$ and $8\,\Delta t$, respectively. Due to the enforcement of the stability condition, Eqn. 4.7, the time steps of the elements are modified to $\Delta t, \Delta t, 2\,\Delta t$ and $4\,\Delta t$, respectively. A vector signifies a nodal update and a σ signifies an element update. The subcycle number in which the node or element can be updated is printed next to the symbol. During a given subcycle, certain elements *must* be updated in order to correctly update the nodes whose nodal clock fall behind the time. The other elements *can* be updated, however their contribution is not needed for nodal updates in the current subcycle. For example, elements 1 and 2 must be updated during each subcycle. However, because element 3 has a time step of $2\,\Delta t$, it can be updated during subcycle 1 or 2. Element 3 *can* be updated in subcycle 1 but if it is not updated early, it *must* be updated in subcycle 2 in order to update node 3 at time $= t + 2\,\Delta t$. Element 4 can be updated during

any of the first four subcycles as long as the update has been completed before node 4 is updated at time = $t + 4\Delta t$. The following table illustrates the element groups which *must* be updated and those which *can* be updated for the example in Figure 4.7.

Subcycle	Groups Which Must Go	Groups Which Can GO
1	1,2	3,4
2	1,2,3	4
3	1,2	3,4
4	1,2,3,4	-
5	1,2	3,4
6	1,2,3	4
7	1,2	3,4
8	1,2,3,4	-

In order to incorporate this algorithm in the parallel code, a flag is assigned to each element group at the beginning of each subcycle. The flag indicates whether the group must be updated during that subcycle or if not, whether the group can be updated early. The flag also indicates whether the group was updated during an earlier subcycle so that multiple updates do not occur. Once the number of element groups which must be updated during the subcycle is known, the number of

processors which will remain idle during the element calculations can be determined. Element groups which have been flagged for early update are then assigned to the idle processors. The possible values for the flag, NFLAG, are listed below.

NFLAG

- 1 - Group must be updated: This flag is reserved for groups using a time step Δt. These groups must be updated every time step.
- 2 - Group must be updated or is being updated early due to an available processor.
- 0 - Group was updated early and cannot be updated until after the appropriate nodes have been updated.
- -2 - Group can be updated early if there is an available processor.

The flags are assigned to the element groups immediately after the group clocks have been updated. The number of element groups which must be updated during the subcycle, NMGO, and the number of idle processors, NCANGO, are then computed. The following code determines which groups will be updated early based on the number of processors (NPROCS).

```
      IF (MOD(NMGO,NPROCS) .EQ. 0) GO TO 100
      NCANGO = NPROCS - MOD(NMGO,NPROCS)
C
      DO 90 J = 1,NBLOCKS
        IF (NFLAG(J) .GE. 0) GO TO 90
        IF (NFLAG(J) .EQ. -2) NFLAG(J) = 2
        NCANGO = NCANGO - 1
        IF (NCANGO .EQ. 0) GO TO 100
   90 CONTINUE
  100 CONTINUE
```

When processors are available to update element groups early, it is most efficient to choose the group with the smallest relative time step. Because the elements were initially sorted in terms of increasing time step before being subdivided into groups, the element groups are naturally ordered in terms of increasing time step. Therefore in the above loop, the elements groups with the smallest time step are given priority.

Table 4.7 shows the allocation of processors for the modified cylindrical panel problem using the new algorithm. The amount of time lost due to idle processors has been reduced by 70% when eight processors are used. Note that the algorithm has no effect on the nodal calculations. Table 4.8 compares the execution times and speed-up due to subcycling for the problem without subcycling, with subcycling and with subcycling using the new allocation algorithm. The theoretical speed-up due to subcycling for this problem, as calculated by Equation 4.2 is 8.47.

The increase in speed-up attributed to the allocation algorithm for 4 and 8 processors is 24% and 30%, respectively. As can been seen, the effectiveness of this algorithm improves as the number of processors increases. Note, however, that the

speed-up due to subcycling is greater using four processors than using eight processors. For problems of this type, where only a few element groups are subcycling, using fewer processors will also minimize processor idleness, thus improving speed-up.

Table 4.2: Material Properties and Loading for Axially Loaded Beam

Thickness	t	=	0.1 in
Young's Modulus	E	=	1.0×10^6 psi
Poisson's ratio	ν	=	0.1
Yield stress	σ_y	=	1.0 psi
Plastic modulus	E_p	=	1.0×10^4 psi
Load for Elastic	P_e	=	0.005 lbs
Load for Plastic	P_p	=	0.05 lbs

Table 4.3: Number of Subcycles per Element Group for Three Group Sizes

Number of Subcycles Req'd	Group Time Step	12 elements per Group	24 elements per Group	32 elements per Group
8	0.3800E-5	8	4	3
4	0.7600E-5	8	4	3
2	1.5200E-5	4	2	2
1	3.0400E-5	9	5	3
		29 Total	15 Total	11 Total

Table 4.4: Run Times (Efficiency) for Containment Vessel with Subcycling

		SINGLE Δt	MIXED	TIME	INTEGRATION
Program Version	Number of Processors	24 elements per group	12 elements per group	24 elements per group	32 elements per group
WHAM0	1	3768			
WHAM_OPT	8	1291			
WHAM0_VECPAR	1	959	394	322	308
	4	309(78%)	122(81%)	111(73%)	105(73%)
	8	201(60%)	81(61%)	85(47%)	91(42%)

Table 4.5: Run Times (Efficiency) for Automobile Impact with Subcycling (in CPU hrs)

Program Version	Number of Processors	SINGLE Δt 32 elements per group	MIXED 32 elements per group	TIME 64 elements per group
WHAM0	1	103.1		
WHAM_OPT	8	41.3		
WHAM0_VECPAR	1	24.9	10.2	10.1
	4	6.8(92%)	3.13(81%)	3.36(75%)
	8	4.4(71%)	2.16(59%)	2.50(51%)

Table 4.6: Allocation of Processors for Modified Cylindrical Panel Problem

Subcycle	P1	P2	P3	P4	P5	P6	P7	P8
1	Δt	$10\Delta t$	$10\Delta t$	$10\Delta t$	$10\Delta t$	$10\Delta t$	$10\Delta t$	$10\Delta t$
	$10\Delta t$	$10\Delta t$	$10\Delta t$	$10\Delta t$	$10\Delta t$	$10\Delta t$	$10\Delta t$	$10\Delta t$
	$10\Delta t$	$10\Delta t$	$10\Delta t$	$10\Delta t$	$10\Delta t$	$10\Delta t$	$10\Delta t$	$10\Delta t$
	$10\Delta t$	$10\Delta t$	$10\Delta t$	$10\Delta t$	$10\Delta t$	$10\Delta t$	$10\Delta t$	$10\Delta t$
	$10\Delta t$	$10\Delta t$	$10\Delta t$	$10\Delta t$	$10\Delta t$	$10\Delta t$	$10\Delta t$	$10\Delta t$
	$10\Delta t$	$10\Delta t$	$10\Delta t$	$10\Delta t$	$10\Delta t$	$10\Delta t$	$10\Delta t$	$10\Delta t$
	$10\Delta t$	$10\Delta t$	$10\Delta t$	idle	idle	idle	idle	idle
2	Δt	idle	idle	idle	idle	idle	idle	idle
3	Δt	idle	idle	idle	idle	idle	idle	idle
4	Δt	idle	idle	idle	idle	idle	idle	idle
5	Δt	idle	idle	idle	idle	idle	idle	idle
6	Δt	idle	idle	idle	idle	idle	idle	idle
7	Δt	idle	idle	idle	idle	idle	idle	idle
8	Δt	idle	idle	idle	idle	idle	idle	idle
9	Δt	idle	idle	idle	idle	idle	idle	idle
10	Δt	idle	idle	idle	idle	idle	idle	idle

Table 4.7: Allocation of Processors using New Allocation Algorithm

Subcycle	P1	P2	P3	P4	P5	P6	P7	P8
1	Δt	$10\Delta t$	$10\Delta t$	$10\Delta t$	$10\Delta t$	$10\Delta t$	$10\Delta t$	$10\Delta t$
2	Δt	$10\Delta t$	$10\Delta t$	$10\Delta t$	$10\Delta t$	$10\Delta t$	$10\Delta t$	$10\Delta t$
3	Δt	$10\Delta t$	$10\Delta t$	$10\Delta t$	$10\Delta t$	$10\Delta t$	$10\Delta t$	$10\Delta t$
4	Δt	$10\Delta t$	$10\Delta t$	$10\Delta t$	$10\Delta t$	$10\Delta t$	$10\Delta t$	$10\Delta t$
5	Δt	$10\Delta t$	$10\Delta t$	$10\Delta t$	$10\Delta t$	$10\Delta t$	$10\Delta t$	$10\Delta t$
6	Δt	$10\Delta t$	$10\Delta t$	$10\Delta t$	$10\Delta t$	$10\Delta t$	$10\Delta t$	$10\Delta t$
7	Δt	$10\Delta t$	$10\Delta t$	$10\Delta t$	$10\Delta t$	$10\Delta t$	$10\Delta t$	$10\Delta t$
8	Δt	$10\Delta t$	idle	idle	idle	idle	idle	idle
9	Δt	idle	idle	idle	idle	idle	idle	idle
10	Δt	idle	idle	idle	idle	idle	idle	idle

Table 4.8: Timings for Modified Cylindrical Panel Problem in sec CPU

Number of Processors	No Subcycling	Subcycling (Speed-up)	Subcycling with Processor Allocation Algorithm (Speed-up)
4	192	37.4 (5.1)	30.5 (6.3)
8	122	30.4 (4.0)	23.6 (5.2)

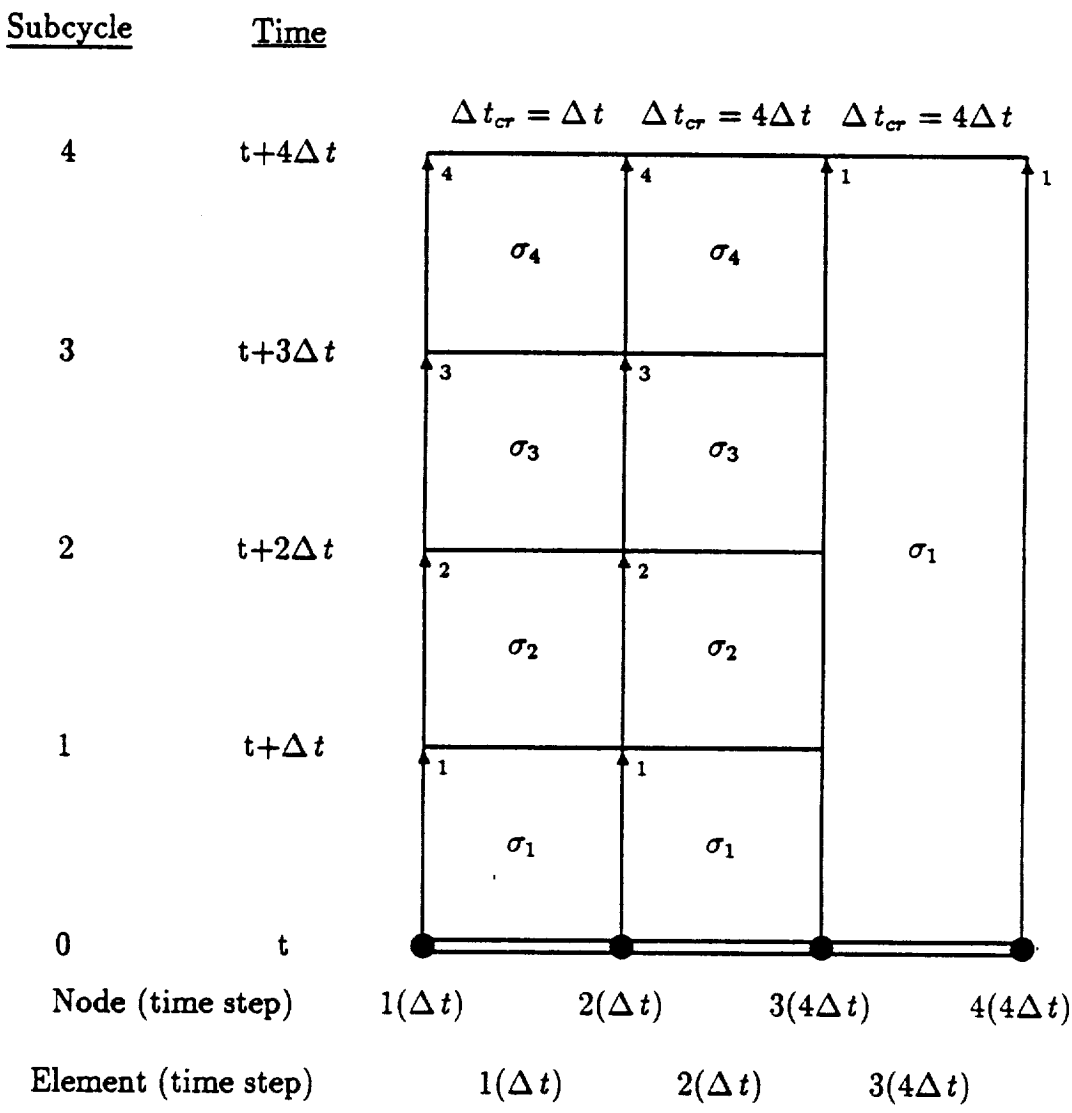
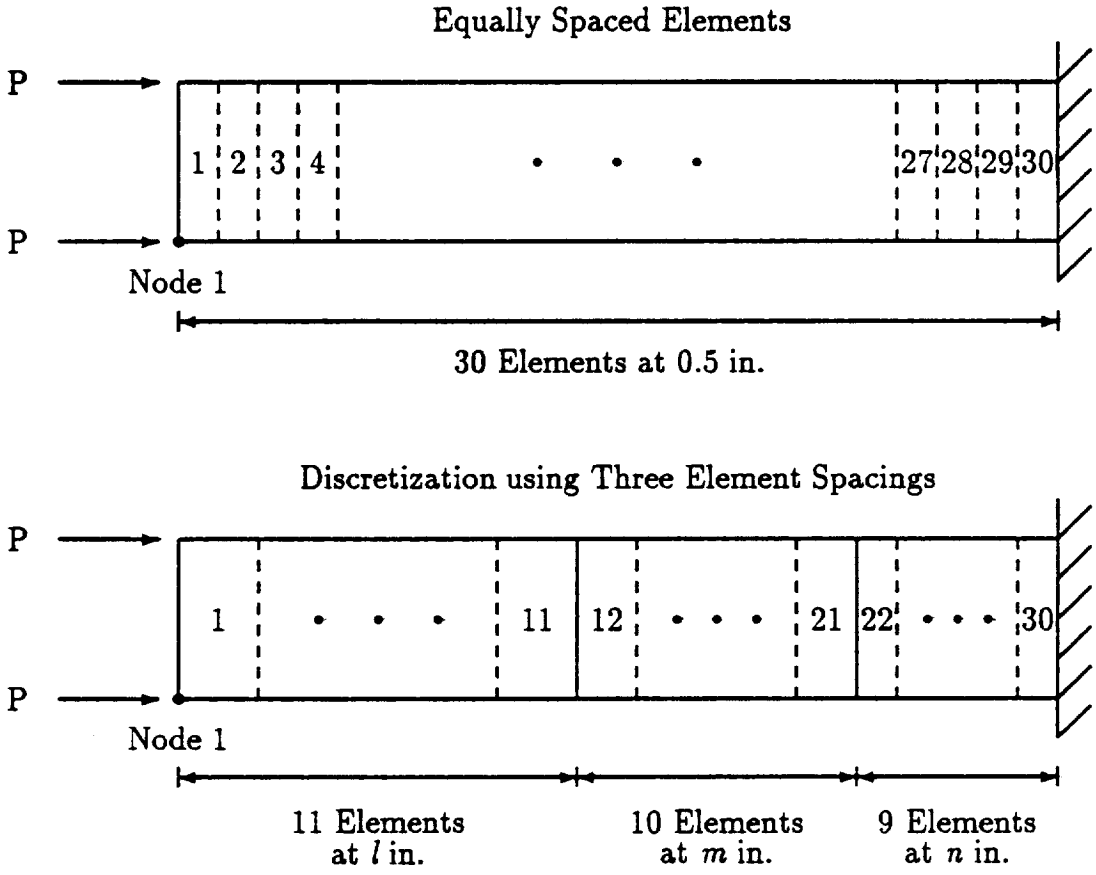


Figure 4.1: Graphical Representation of Subcycling



Case	l	m	n
1	1.075	0.269	0.054
2	1.100	0.257	0.037
3	1.242	0.124	0.010

Figure 4.2: Axially Loaded Beam for Subcycling Stability Study

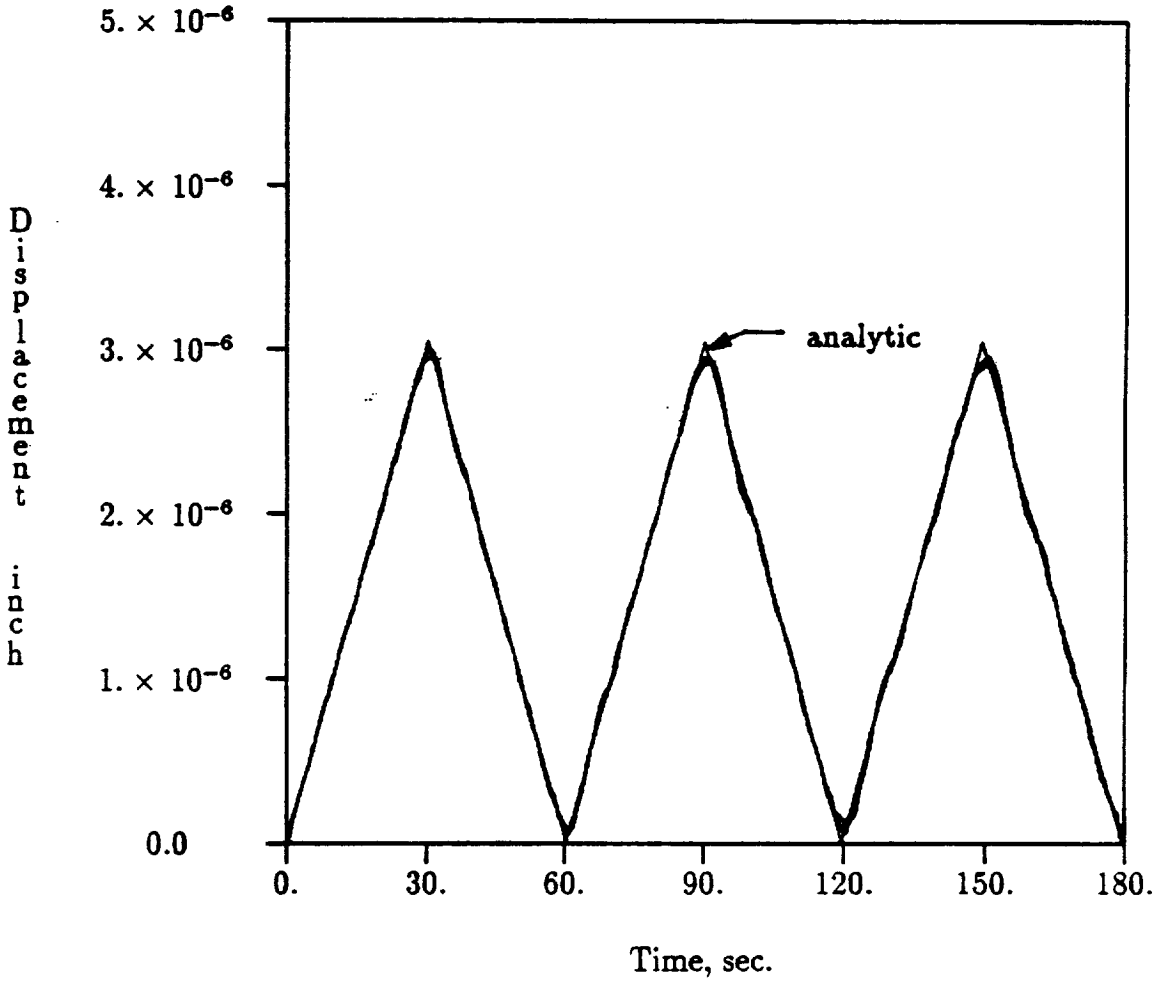


Figure 4.3: Displacement vs. Time : Elastic Rod at x = 0.0 in.

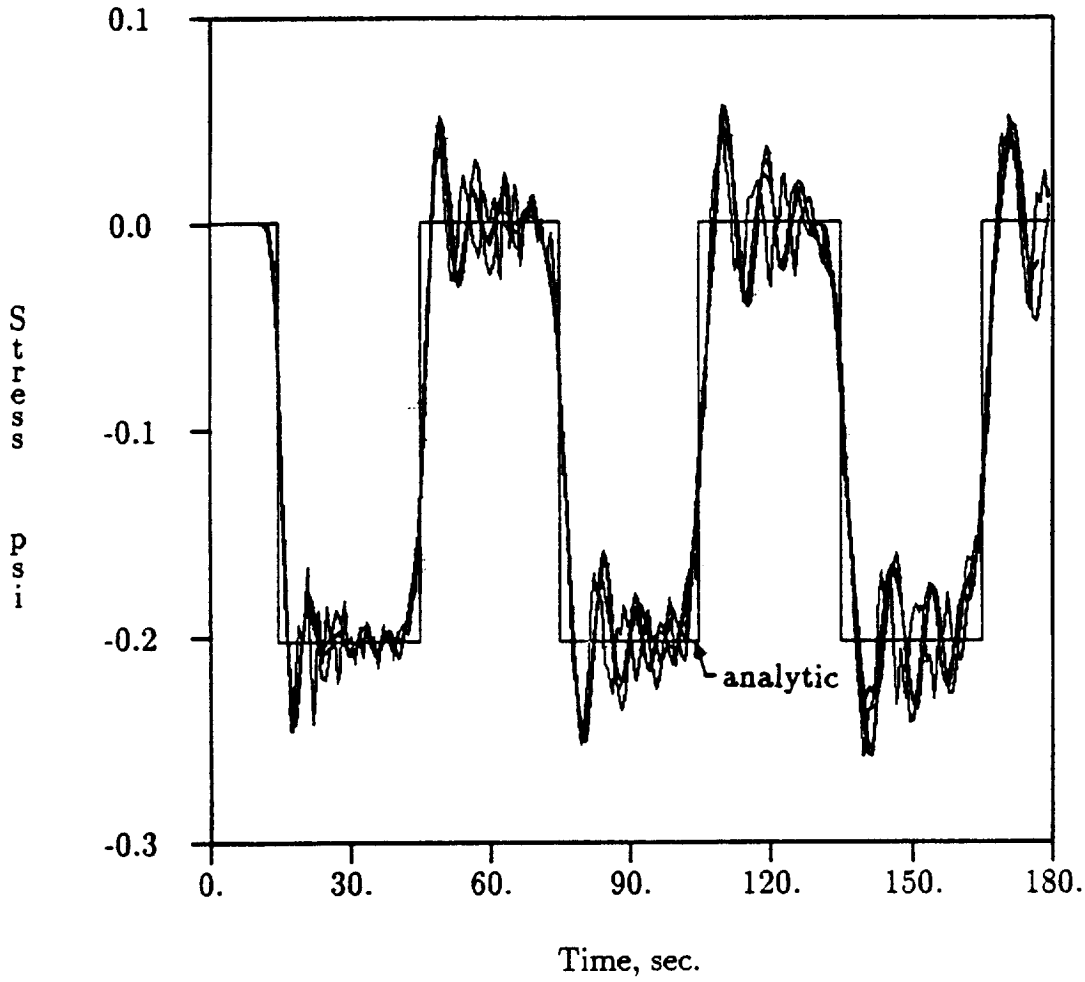


Figure 4.4: Stress vs. Time : Elastic Rod at x = 15 in.

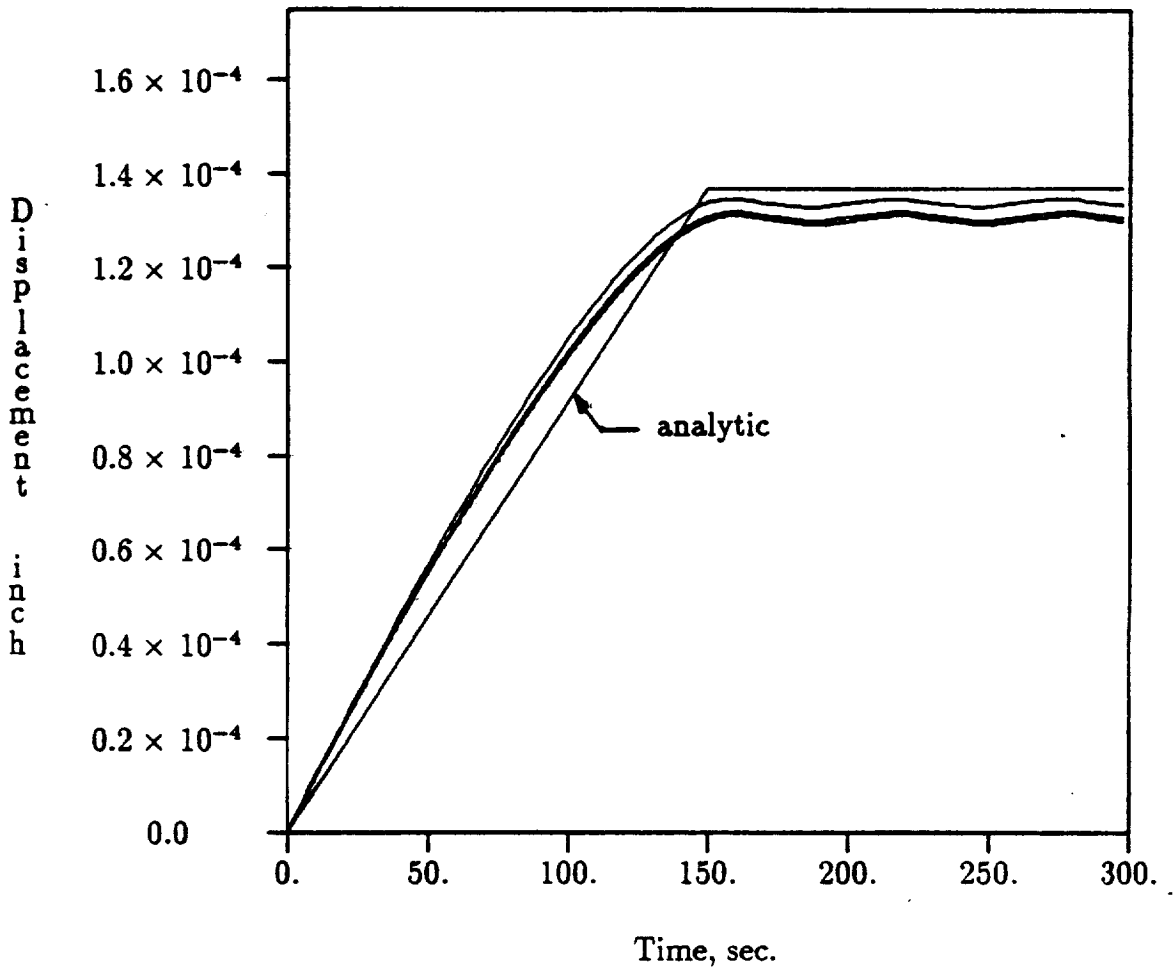


Figure 4.5: Displacement vs. Time : Elastic-Plastic Rod at x = 0.0 in.

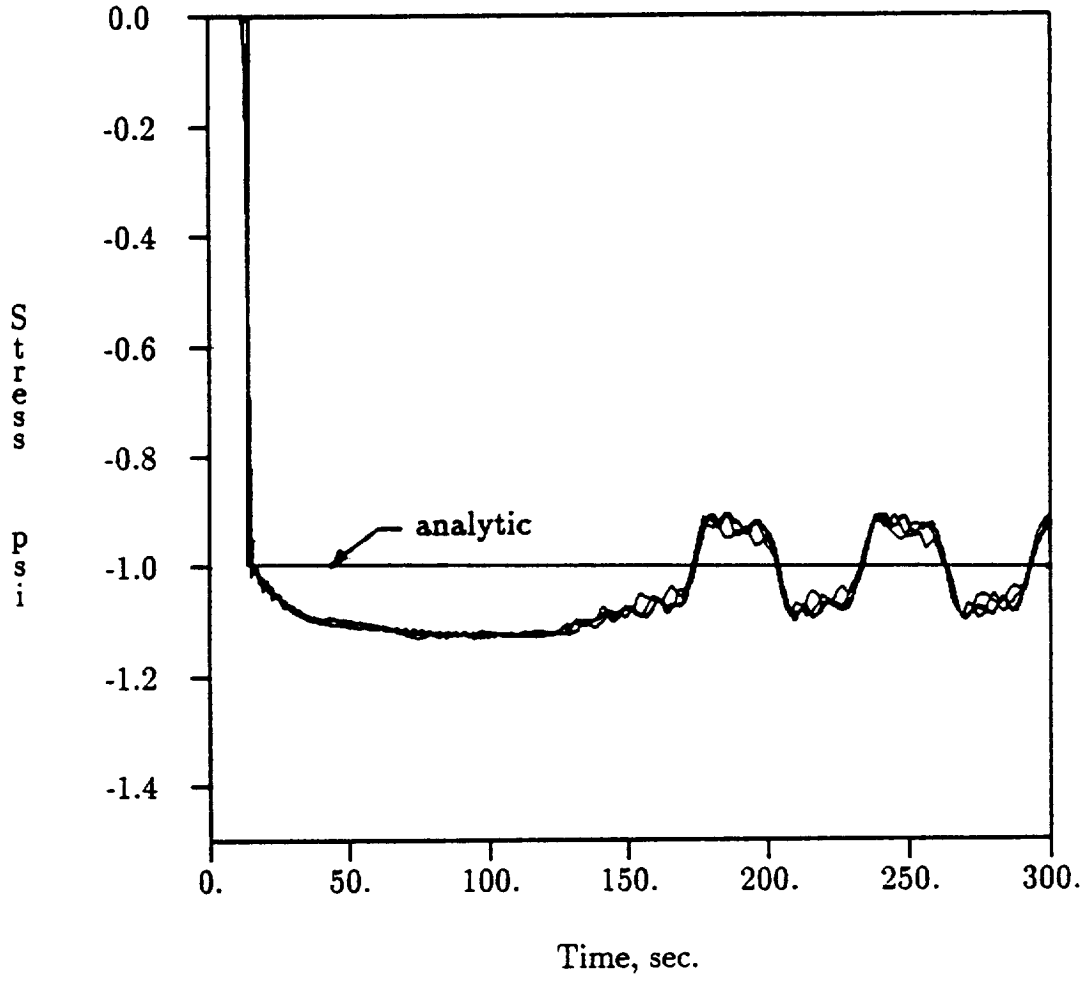


Figure 4.6: Stress vs. Time : Elastic-Plastic Rod at x = 15 in.

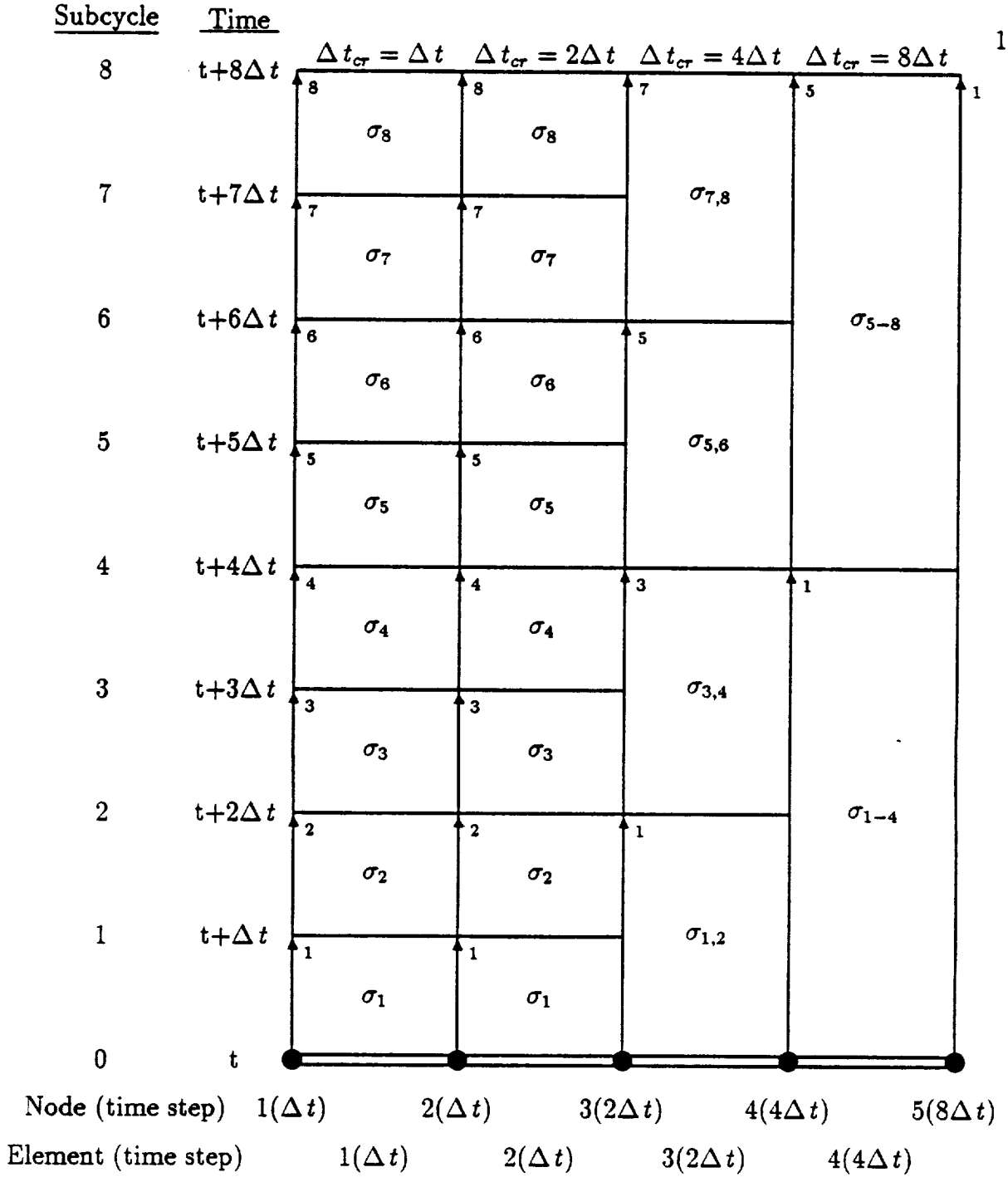


Figure 4.7: Graphical Representation of Processor Allocation Algorithm

Chapter 5

Conjugate Gradient Method

5.1 Introduction

Conjugate gradient methods for solving nonlinear algebraic equations can be readily adapted to a parallel environment. Because these methods are iterative in nature, their effectiveness for nonlinear finite element analysis relies heavily on the ability to calculate the internal force vector in a highly efficient manner. As shown in Table 3.12, the parallel computation of the internal force vector is performed at an efficiency (speed-up/no. processors) of approximately 90 and 76% for four and eight processors, respectively. (Note that the efficiency of the parallel operations varies for each problem, however, in general, the efficiency improves as the problem size increases.) Therefore, conjugate gradient methods should achieve a high degree of success for analyses performed on multiple processors.

In addition to being easily parallelized, explicit-iterative methods have other advantages over solution methods such as the modified or unmodified Newton-Raphson

method. Explicit methods do not require the expensive computation and assembly of the global stiffness matrix. This advantage is especially important in nonlinear analysis because the tangential stiffness matrix changes with the evolution of the response. In Newton methods, the stiffness must be frequently recomputed and triangularized. Finally, because no stiffness matrix need be stored in iterative methods, less memory is required.

The conjugate gradient method is often applied iteratively to a series of linear problems

$$\boldsymbol{K}_{\nu}^{tang}\,\Delta \boldsymbol{d}_{\nu} = \boldsymbol{r}_{\nu} \tag{5.1}$$

where $\boldsymbol{K}_{\nu}^{tang}$ is the linear tangent matrix in step ν [18]. In elastic-plastic problems, $\boldsymbol{K}_{\nu}$ may be based on the current state of the element (or quadrature points). When convergence of the conjugate gradient algorithm is achieved, this only provides a solution to the linearized equations, so the procedure must be repeated with a new tangent stiffness until the residual $\boldsymbol{r}$ meets the error tolerance. An advantage of this approach is that the convergence of the conjugate gradient method for problem (5.1) is guaranteed. Therefore, it can be viewed as a full Newton method with an iterative solution of the Newton equations. In this chapter, a direct approach to the nonlinear equations will be considered where, in effect, $\boldsymbol{K}^{tang}$ varies during the iterative procedure.

A disadvantage of nonlinear solution methods is the potentially slow convergence

for some types of nonlinear behavior. Using conjugate gradient methods, slow convergence can occur when the structure unloads. For example, assume a member of a structure has yielded and then unloads. A new displaced shape for the structure is assumed and an elastic stress is computed for the member. If the assumed solution overshoots the equilibrium position, the updated displacements may force the member into the plastic region. As the iterations continue, the state of the member may oscillate between elastic and plastic behavior. This oscillation can significantly slow convergence or prevent convergence entirely. Research involving iterative solution methods includes developing techniques to improve convergence in nonlinear problems. Techniques currently used for accelerating convergence will be discussed and compared to a new method developed to improve the convergence for nonlinear problems.

5.2 Nonlinear Conjugate Gradient Algorithm

Conjugate gradient methods provide an iterative solution for the minimization of a function f($\boldsymbol{x}$) which can be approximated as a quadratic function:

$$f(\boldsymbol{x}) = c - \boldsymbol{b} \cdot \boldsymbol{x} + \frac{1}{2}\boldsymbol{x} \cdot \boldsymbol{A} \cdot \boldsymbol{x} \tag{5.2}$$

and whose gradient is easily calculated as

$$g(\boldsymbol{x}) = \boldsymbol{A} \cdot \boldsymbol{x} - \boldsymbol{b} \tag{5.3}$$

The matrix $\boldsymbol{A}$ is the Hessian matrix of the function, i.e., its components are the second partial derivatives of f($\boldsymbol{x}$). As the gradient approaches zero, $\boldsymbol{x}$ approaches the solution which minimizes f($\boldsymbol{x}$).

In structural mechanics, the Hessian matrix corresponds to the structural stiffness matrix, $\boldsymbol{x}$ refers to the nodal displacements and $\boldsymbol{b}$ is the external force vector. Therefore, Eqn. 5.3 expresses the error or residual in the equilibrium equations for the displaced shape $\boldsymbol{x}$ and can be written as

$$g(\boldsymbol{x}) = \boldsymbol{K}\,\boldsymbol{d} - \boldsymbol{f}^{ext} \tag{5.4}$$

or

$$g(\boldsymbol{x}) = \boldsymbol{f}^{int} - \boldsymbol{f}^{ext} \tag{5.5}$$

The solution technique systematically minimizes the function f($\boldsymbol{x}$) along a number of "non-interfering" directions called conjugate directions [25]. Two directions $\boldsymbol{u}$ and $\boldsymbol{v}$ are conjugate if they satisfy the following relationship:

$$\boldsymbol{u} \cdot \boldsymbol{A} \cdot \boldsymbol{v} = 0 \tag{5.6}$$

Physically, this relationship can be described as follows. If the function is minimized along some direction $\boldsymbol{u}$, the computed gradient of the function at the minimum will be perpendicular to $\boldsymbol{u}$ because the component of g($\boldsymbol{x}$) along $\boldsymbol{u}$ must equal zero at the minimum. The function can then be minimized along a new direction $\boldsymbol{v}$ without

altering the previous minimization by requiring that the gradient remains perpendicular to $\boldsymbol{u}$ during the minimization along $\boldsymbol{v}$. In other words, the change in the gradient must remain perpendicular to $\boldsymbol{u}$. The change in the gradient is given by

$$\delta \boldsymbol{g} = \boldsymbol{A} \cdot \delta \boldsymbol{x} \tag{5.7}$$

or

$$\delta \boldsymbol{g} = \boldsymbol{A} \cdot \boldsymbol{v} \tag{5.8}$$

as f($\boldsymbol{x}$) is minimized along $\boldsymbol{v}$. Therefore, Eqn. 5.8 expresses the condition for conjugate directions. Because each new direction must satisfy this condition, the result is a set of linearly independent, mutually conjugate directions.

Conjugate gradient methods are based on the following theorem presented in [25]: Let $\boldsymbol{A}$ be a symmetric, positive-definite, $n \times n$ matrix. Let $\boldsymbol{g}_0$ be an arbitrary vector and $\boldsymbol{p}_0 = \boldsymbol{g}_0$. For $i = 0, 1, 2, ...$, define two sequences of vectors

$$\boldsymbol{g}_{i+1} = \boldsymbol{g}_i - \alpha_i \boldsymbol{A} \cdot \boldsymbol{p}_i \tag{5.9}$$

and,

$$\boldsymbol{p}_{i+1} = \boldsymbol{g}_{i+1} + \beta_i \boldsymbol{p}_i \tag{5.10}$$

where α_i and β_i are chosen such that at successive iterations the gradients $\boldsymbol{g}$ remain

orthogonal and the directions $\boldsymbol{p}$ remain conjugate, i.e.,

$$\boldsymbol{g}_{i+1} \cdot \boldsymbol{g}_i = 0$$

and

$$\boldsymbol{p}_{i+1} \cdot \boldsymbol{A} \cdot \boldsymbol{p}_i = 0.$$

Then, for all $i \neq j$,

$$\begin{aligned} \boldsymbol{g}_i \cdot \boldsymbol{g}_j &= 0 \\ \boldsymbol{p}_i \cdot \boldsymbol{A} \cdot \boldsymbol{p}_j &= 0 \end{aligned} \tag{5.11}$$

Therefore, the conjugate gradient methods produce a sequence of mutually orthogonal gradients as well as a set of mutually conjugate directions.

Note that the update equation for the gradient in Eqn. 5.9 contains the Hessian matrix. Instead of computing the Hessian to calculate the new gradient, the local minimum of f($\boldsymbol{x}$) along the direction $\boldsymbol{p}_i$ can be found using a line search technique. Therefore, the local minimum is computed as $\boldsymbol{x}_{i+1}$ and Eqn. 5.9 can be replaced by:

$$\boldsymbol{g}_{i+1} = g(\boldsymbol{x}_{i+1}) = \boldsymbol{f}^{int}(\boldsymbol{x}_{i+1}) - \boldsymbol{f}^{ext} \tag{5.12}$$

Line search techniques will be discussed later.

The following flow chart illustrates the implementation of the basic conjugate gradient algorithm.

Flow Chart for Conjugate Gradient Algorithm

1. Assuming an initial $\boldsymbol{x}_1$, compute the gradient $\boldsymbol{g}_1$ and assign the gradient direction: $\boldsymbol{p}_1 = -\boldsymbol{g}_1$

2. Loop over conjugate directions

   (a) Perform line search for local minimum along direction $\boldsymbol{p}_i$

      i. Compute the step size alpha

      ii. Update displacement vector: $\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + \alpha_i \ \boldsymbol{p}_i$

      iii. Compute internal force vector $\boldsymbol{f}^{int}$

      iv. Compute new gradient: $\boldsymbol{g}_{i+1} = \boldsymbol{f}^{int} - \boldsymbol{f}^{ext}$

      v. Test for convergence. If yes, go to 3

   (b) Compute β_i

   (c) Compute new gradient direction: $\boldsymbol{p}_{i+1} = \boldsymbol{g}_{i+1} + \beta_i \ \boldsymbol{p}_i$

   (d) Go to 2

3. Output.

The equilibrium convergence criterion used in Step 2.a.v is given by:

$$\left(\frac{||\boldsymbol{g}||}{||\boldsymbol{f}^{ext}||} \right)^{\frac{1}{2}} \leq \epsilon \tag{5.13}$$

where $||\cdot||$ is the L^2 norm of the vector and $\boldsymbol{f}^{ext}$ is the vector of applied forces.

The differences between versions of the conjugate gradient method lie primarily in the computation of the coefficients α and β. The parameter α is chosen to ensure that each computation of the gradient $\boldsymbol{g}_{i+1}$ is orthogonal to its immediate predecessor, and β ensures that the direction $\boldsymbol{p}_{i+1}$ is conjugate to its immediate predecessor. The most common expressions for β are:

$$\beta_i = \frac{\boldsymbol{g}_{i+1} \cdot \boldsymbol{g}_{i+1}}{\boldsymbol{g}_i \cdot \boldsymbol{g}_i}, \quad \text{Fletcher-Reeves [14]} \tag{5.14}$$

and,

$$\beta_i = \frac{\boldsymbol{g}_{i+1} \cdot (\boldsymbol{g}_{i+1} - \boldsymbol{g}_i)}{\boldsymbol{g}_i \cdot \boldsymbol{g}_i}, \quad \text{Polak-Ribière [24]} \tag{5.15}$$

Note that these expressions for β are identical for an exact quadratic function. However, $f(\boldsymbol{x})$ is rarely an exact quadratic so the second term in the numerator of the Polak-Ribière expression acts as an accelerator for convergence [9].

As mentioned previously, the computation of α is performed using a line search technique to find the local minimum of $f(\boldsymbol{x})$ along some direction $\boldsymbol{p}_i$. The classical approach for the line search consists of first bracketing the solution along the search direction and then iterating to the optimum α. Possible iterative schemes include a golden section search, a quadratic or cubic curve fitting technique, and the method of false position. Another line search technique used by Klessig and Polak [20] in the implementation of the Polak-Ribière conjugate gradient algorithm uses a stability test which insures convergence instead of a minimization routine to determine α. Papadrakakis concludes [23] that the classical approach with the method of false

position-bisection technique is the most effective of the methods discussed above. Therefore, this technique was used for the work presented in the chapter.

5.3 Convergence Enhancements

Two methods were used to study the convergence properties of the conjugate gradient method. Diagonal scaling is frequently used in practice and significantly improves convergence properties for elastic-plastic solutions. A new method called the zeta-parameter method is also presented as an alternative to diagonal scaling.

5.3.1 Diagonal Scaling

The condition number of the matrix $\boldsymbol{A}$ is an important factor in determining the effectiveness of iterative solutions. The condition number is a numerical measure of the ill-conditioning which exists in the matrix $\boldsymbol{A}$ and is defined by:

$$C(\boldsymbol{A}) = \frac{\lambda_{max}}{\lambda_{min}} \tag{5.16}$$

where λ_{max} and λ_{min} are the maximum and minimum eigenvalues of $\boldsymbol{A}$ [11]. A frequent cause of ill-conditioning, and thus a large condition number, in a structural stiffness matrix is a large disparity in element stiffness properties. By scaling the equations, the condition number of $\boldsymbol{A}$ can be minimized and the convergence properties of the solution can be improved.

Diagonal scaling is a commonly used technique which has been proven effective in improving the convergence of conjugate gradient methods. The elements of the diagonal scaling matrix $\boldsymbol{W}$ are equal to the diagonal elements of the global stiffness matrix $\boldsymbol{A}$. The scaling matrix is computed once prior to the first iteration. The computation of the matrix is performed efficiently by calculating the diagonal terms of the element stiffness matrices and assembling them in vector form. The element stiffness matrix $\boldsymbol{k}$ is given by

$$\boldsymbol{k} = \int_V \boldsymbol{B}^T \boldsymbol{C}\, \boldsymbol{B}\, dV \tag{5.17}$$

where $\boldsymbol{B}$ is the gradient matrix and $\boldsymbol{C}$ is the matrix of material constants. For a four-node plane isoparametric element, the stiffness is an 8×8 matrix computed by

$$\boldsymbol{k} = \int_{-1}^{1} \int_{-1}^{1} \boldsymbol{B}^T \boldsymbol{C}\, \boldsymbol{B}\, t\, J\, d\xi\, d\eta \tag{5.18}$$

where t is the element thickness, J is the determinant of the Jacobian matrix and ξ and η are the natural coordinates. Because only the diagonal terms of the element stiffness matrix contribute to the diagonal of the global stiffness matrix, these terms can be computed explicitly.

The gradient matrix $\boldsymbol{B}$ for a 4-node constant strain quadrilateral element with

two degrees of freedom per node is given by

$$\boldsymbol{B} = \frac{1}{J}\begin{bmatrix} \frac{\delta N_1}{\delta \xi} & 0 & \frac{\delta N_2}{\delta \xi} & 0 & \frac{\delta N_3}{\delta \xi} & 0 & \frac{\delta N_4}{\delta \xi} & 0 \\ 0 & \frac{\delta N_1}{\delta \eta} & 0 & \frac{\delta N_2}{\delta \eta} & 0 & \frac{\delta N_3}{\delta \eta} & 0 & \frac{\delta N_4}{\delta \eta} \\ \frac{\delta N_1}{\delta \eta} & \frac{\delta N_1}{\delta \xi} & \frac{\delta N_2}{\delta \eta} & \frac{\delta N_2}{\delta \xi} & \frac{\delta N_3}{\delta \eta} & \frac{\delta N_3}{\delta \xi} & \frac{\delta N_4}{\delta \eta} & \frac{\delta N_4}{\delta \xi} \end{bmatrix} \tag{5.19}$$

corresponding to the unknowns

$$\boldsymbol{u}^T = [u_1\ v_1\ u_2\ v_2\ u_3\ v_3\ u_4\ v_4].$$

N_I is the shape function of node I and J is the determinant of the Jacobian given by

$$J = \det \boldsymbol{J} = x_{,\xi}\, y_{,\eta} - x_{,\eta}\, y_{,\xi} \tag{5.20}$$

For one-point quadrature, the gradient matrix and the Jacobian are evaluated at the point $(\xi, \eta) = (0, 0)$, giving

$$\det \boldsymbol{J} = \frac{A}{4} \tag{5.21}$$

and,

$$\boldsymbol{B} = \frac{1}{2A}\begin{bmatrix} y_{24} & 0 & y_{31} & 0 & y_{42} & 0 & y_{13} & 0 \\ 0 & x_{42} & 0 & x_{13} & 0 & x_{24} & 0 & x_{31} \\ x_{42} & y_{24} & x_{13} & y_{31} & x_{24} & y_{42} & x_{31} & y_{13} \end{bmatrix} \tag{5.22}$$

where,

$$x_{IJ} = x_I - x_J$$

and,

$$y_{IJ} = y_I - y_J$$

A is the area of the element and x_I and y_I are the coordinates of node I. Using a plane stress material matrix

$$C = \frac{E}{1-\nu^2}\begin{bmatrix} 1 & \nu & 0 \\ \nu & 1 & 0 \\ 0 & 0 & \frac{1-\nu}{2} \end{bmatrix} \tag{5.23}$$

and assuming a unit thickness, the diagonal terms of the element stiffness matrix can be computed by Eqn. 5.18 as

$$\begin{aligned} k_{11} &= k_{55} = \tfrac{1}{A}\left[\tfrac{E}{1-\nu^2}\, y_{24}^2 + \tfrac{E}{2(1+\nu)}\, x_{42}^2\right] \\ k_{22} &= k_{66} = \tfrac{1}{A}\left[\tfrac{E}{1-\nu^2}\, y_{31}^2 + \tfrac{E}{2(1+\nu)}\, x_{13}^2\right] \\ k_{33} &= k_{77} = \tfrac{1}{A}\left[\tfrac{E}{2(1+\nu)}\, y_{24}^2 + \tfrac{E}{1-\nu^2}\, x_{42}^2\right] \\ k_{44} &= k_{88} = \tfrac{1}{A}\left[\tfrac{E}{2(1+\nu)}\, y_{31}^2 + \tfrac{E}{1-\nu^2}\, x_{13}^2\right] \end{aligned} \tag{5.24}$$

Assembly is performed by adding the stiffness contributions due to each element

to a global vector according to its nodal degree of freedom. This method was used for the assembly of all nodal arrays. Storage is also minimized using this algorithm. Usually the elastic material constants are used in computing the scaling matrix. The gradient vector is then scaled by multiplying by the inverse of the scaling matrix. The flow chart from the previous section is then modified as follows:

Flow Chart for Conjugate Gradient Algorithm with Scaling

1. Compute the diagonal scaling matrix $\boldsymbol{W}$
2. Assuming an initial $\boldsymbol{x}_1$, compute the gradient $\boldsymbol{g}_1$
3. Scale the gradient vector: $\boldsymbol{z}_1 = \boldsymbol{W}^{-1}\,\boldsymbol{g}_1$
4. Assign the gradient direction: $\boldsymbol{p}_1 = -\,\boldsymbol{z}_1$.
5. Loop over conjugate directions
   (a) Perform line search for local minimum along direction $\boldsymbol{p}_i$
      i. Compute the step size alpha
      ii. Update displacement vector: $\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + \alpha_i\ \boldsymbol{p}_i$
      iii. Compute internal force vector $\boldsymbol{f}^{int}$
      iv. Compute new gradient: $\boldsymbol{g}_{i+1} = \boldsymbol{f}^{int} - \boldsymbol{f}^{ext}$
      v. Test for convergence. If yes, go to 3
   (b) Compute β_i
      i. Scale the new gradient vector: $\boldsymbol{z}_{i+1} = \boldsymbol{W}^{-1}\,\boldsymbol{g}_{i+1}$

ii. Compute β

$$\beta = \frac{z_{i+1} W z_{i+1}}{z_i W z_i} = \frac{z_{i+1} g_{i+1}}{z_i g_i}, \quad \text{Fletcher-Reeves}$$

or,

$$\beta = \frac{z_{i+1} g_{i+1} - z_i g_i}{z_i g_i} \quad \text{Polak-Ribière}$$

(c) Compute new gradient direction: $p_{i+1} = z_{i+1} + \beta_i \; p_i$

(d) Go to 2

6. Output.

5.3.2 Zeta-Parameter Method

A new method developed to improve the convergence properties of the conjugate gradient method is called the zeta-parameter method. When an element changes state, i.e., goes from elastic behavior to plastic behavior or visa versa, an iterative method will frequently cause the stiffness of the element to oscillate from elastic to plastic as the solution is approached. This oscillation in the element stiffness can significantly slow down the convergence of the solution. The zeta-parameter method attempts to decrease the amount of oscillation in the element stiffness, thus converging to a solution in a more direct manner.

Zeta is a scalar of the form:

$$\zeta = 1.0 - \kappa * \left(\frac{\boldsymbol{g}_i^T \boldsymbol{g}_i}{\boldsymbol{g}_1^T \boldsymbol{g}_1} \right)^m \tag{5.25}$$

where κ is a constant between 0.5 and 0.9 and m is a constant between 0.0 and 1.0. Typical values for κ and m are 0.8 and 0.5, respectively. The subscript on the gradient vector $\boldsymbol{g}$ refers to the iteration number. Zeta is used to compute a fictitious element stress state which is a linear combination of the elastic and nonlinear stresses given by

$$\boldsymbol{\sigma} = (1 - \zeta) * \boldsymbol{\sigma}_{elastic} + \zeta * \boldsymbol{\sigma}_{plastic} \tag{5.26}$$

Zeta is only applied to elements which change state during the load step. When an element changes from elastic behavior to plastic behavior, both the elastic and plastic stresses are calculated. During the first iteration, ζ is small, i.e., using the typical value for κ, $\zeta = 0.2$. Therefore, the computed stress is close to the calculated elastic stress. As the number of iterations or directions increases, the value of the gradient at the current iteration decreases with respect to the gradient of the first iteration and zeta increases. The computed stress slowly approaches the calculated plastic stress. As convergence is achieved, the value of ζ goes to 1.0 and the computed stress tends to the calculated plastic stress. If the element unloads, the expression for the stress state is given by:

$$\boldsymbol{\sigma} = \zeta * \boldsymbol{\sigma}_{elastic} + (1 - \zeta) * \boldsymbol{\sigma}_{plastic} \tag{5.27}$$

and the internal force of the element is computed from a stress state which slowly changes from plastic to elastic.

The modified flow chart for the zeta-parameter method is given below.

Flow Chart for Conjugate Gradient Algorithm with Zeta-Parameter

1. Assuming an initial $\boldsymbol{x}_1$, compute the gradient $\boldsymbol{g}_1$ and assign the gradient direction: $\boldsymbol{p}_1 = -\boldsymbol{g}_1$

2. Loop over conjugate directions

   (a) Perform line search for local minimum along direction $\boldsymbol{p}_i$ - Note: the line search is based on the gradient calculated using the actual stresses, not the stresses modified by ζ

      i. Compute the step size alpha

      ii. Update displacement vector: $\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + \alpha_i \ \boldsymbol{p}_i$

      iii. Compute elastic and plastic element stresses

      iv. Apply ζ parameter to elements which change state

      v. Compute internal force vector based on actual stresses for convergence criteria

      vi. Compute internal force vector using stresses modified by ζ via Eqns. 5.26 and 5.27 for elements which change state

      vii. Compute new gradients using internal forces in (v) and (vi): $\boldsymbol{g}_{i+1} = \boldsymbol{f}^{int} - \boldsymbol{f}^{ext}$

viii. Test for convergence using (v). If yes, go to 3

(b) Compute β_i

(c) Compute ζ_i

(d) Compute new gradient direction: $\boldsymbol{p}_{i+1} = \boldsymbol{g}_{i+1} + \beta_i \ \boldsymbol{p}_i$

(e) Go to 2

3. Output.

5.4 Numerical Studies

Two problems are used to study the behavior of diagonal scaling and the zeta-parameter technique.

1. A statically loaded rigid bar supported by three springs,

2. A dynamically loaded cantilever beam.

The first problem is a rigid bar suspended by three equally spaced springs as shown in Figure 5.1. The bar is loaded statically by an applied force located halfway between springs 2 and 3. The yield stress for spring 2 is twice as large as the yield stresses for the other two springs. Using a load increment of 0.1, the behavior of the structure is as follows:

1. Stage 1 : $0.0 < \mathrm{P} < 0.8$

   All springs load in tension

2. Stage 2 : $0.8 < P < 0.9$

   Spring 3 yields in tension; spring 1 unloads; spring 2 loads in tension

3. Stage 3 : $0.9 < P < 1.3$

   Spring 1 loads in compression; springs 2 and 3 load in tension

4. Stage 4 : $1.3 < P < 1.4$

   Spring 2 yields in tension; spring 1 unloads; spring 3 loads in tension

The conjugate gradient method without convergence enhancements requires 19 direction iterations for stage 2 and 53 direction iterations for stage 4. These stages correspond to the unloading of spring 1 caused by the yielding of the other springs. In order to visualize the path traversed during each iteration, contours of the residual or gradient were plotted as a function of the change in displacement at the point of load application and the change in rotation of the rigid bar. Figures 5.2 and 5.6 show the three-dimensional plots of these contours for the load steps in Stages 2 and 4, respectively. The contour plot for Stage 2 forms a shallow valley, while the contours for Stage 4 form a very steep valley with plateaus on each edge. The plateaus correspond to configuration states in which all three springs have yielded. The plateau to the left of the valley corresponds to springs 2 and 3 yielding in tension and spring 1 yielding in compression while the plateau to the right of the valley corresponds to all springs yielding in tension. The left slope of the valley wall corresponds to spring 1 loading in compression and the right slope of the valley wall corresponds to spring 1 loading in tension. Figures 5.3 and 5.7 show the path

formed by plotting the intermediate solution for each direction iteration for the basic conjugate gradient method superimposed on a two-dimensional plot of the contours for Stages 2 and 4. The resulting paths show a slow zigzag traverse across the valley until the solution is achieved. Note that the stress in spring 1 oscillates from elastic to plastic as the path traverses the valley. The purpose of the convergence enhancement techniques is to modify the directions to arrive at the solution while minimizing the zigzag motion.

The diagonal scaling and zeta-parameter techniques were implemented into the conjugate gradient program and the number of iterations were monitored. The expressions for β developed by Fletcher-Reeves and Polak-Ribière were used to determine the effect on convergence properties.

The diagonal scaling technique required 8 iterations for the Stage 2 load step and 30 iterations for the Stage 4 load step. For this problem, the expression used for β made no difference in the number of iterations required. The stiffness term used for the scaling factor was also modified to determine the effect on the number of iterations. The following parameters were combined to determine the effect of modifying the scaling factor:

1. Diagonal scaling matrix equal to the diagonal terms of the stiffness matrix,

2. Diagonal scaling matrix equal to the square root of the diagonal terms of the stiffness matrix,

3. Scaling factor using elastic constants for all springs,

4. Scaling factor using elastic and plastic constants corresponding to the current state of the element.

Again, no change in the number of iterations required for convergence was found due to these modifications. Figures 5.4 and 5.8 show the solution paths for the conjugate gradient method with diagonal scaling for the load step corresponding to Stage 2 and 4, respectively. As can been seen, the amount of zigzag motion has been decreased.

The zeta-parameter method with constants $\kappa = 0.8$ and $m = 0.5$ required 9 iterations for the Stage 2 load step and 7 iterations for the Stage 4 load step using the Fletcher-Reeves β expression. The number of iterations varied slightly when the Polak-Ribière expression was used. Variations in the constants used in the expression for zeta (Eqn. 5.25) also affected the number of iterations required for convergence. For example, by changing κ to 0.7, the method required 13 iterations for Stage 2 and 14 iterations for Stage 4. Using $\kappa = 0.5$, the method required 9 iterations for Stage 2 and 41 iterations for Stage 4.

Figures 5.5 and 5.9 show the solution paths for Stage 2 and 4, respectively. The zeta-parameter method traverses the valley in a more direct manner and converges to the solution quickly. Therefore, by selecting the correct constants in the expression for zeta, the zeta-parameter method significantly improves the convergence properties of the conjugate gradient method.

If the proper constants are not selected, the zeta-parameter method tends to overshoot the solution and must reverse directions for convergence. This frequently

requires many iterations and thus makes the method less effective. Another disadvantage is that the method tends to iterate about the wrong solution. In some cases, a large number of iterations are made before the method finds the correct direction along which to minimize. This problem is reduced by restarting the solution method when the difference in displacements between successive iterations is negligible. Restarting is performed by simply setting β to zero, thus assigning the next direction equal to the negative of gradient.

The second problem studied is the two-dimensional cantilever beam showed in Figure 5.10. An oscillating concentrated load is applied at the free end of the beam. The load versus time curve for the concentrated load is shown in Figure 5.11. The x-component of the displacement of all nodes along the bottom of the beam is constrained. The purpose of constraining the bottom nodes is to increase the amount of yielding in the beam.

The problem was first run with the three versions of the conjugate gradient program using the Fletcher-Reeves β expression. None of the codes converged during the load step corresponding to the reversal of the applied load, i.e., convergence was not achieved within 350 iterations. Using the Polak-Ribière expression for β, all versions of the code converged within the allowable number of iterations.

Table 5.1 lists the number of iterations required for convergence for each time step using the Polak-Ribière β expression. Diagonal scaling significantly improved the convergence properties of the solution. The number of iterations required for convergence was reduced by as much as 2. The larger reductions in iterations occurred

in the load steps in which the conjugate gradient method had particular difficulty in converging due to element yielding and unloading. Diagonal scaling had little effect during load steps in which the elements remain elastic.

The zeta-parameter method showed little improvement over the basic conjugate gradient method in the number of iterations required for convergence. Slight improvement is seen in some load steps; however, this improvement was balanced by a slight increase in the number of iterations required for other load steps. The constants used in the expression for ζ were modified with no significant effect on the number of iterations. By including diagonal scaling with the zeta-parameter method, the number of iterations required for convergence was significantly reduced. The amount of improvement was similar to the improvement found by using diagonal scaling alone.

It appears that the effectiveness of the zeta-parameter method is diminished in more complex problems. In the cantilever beam problem, several elements are yielding or unloading during a given load step. In addition, the change of state in an element may occur at different times during the iteration process. However, ζ will only be effective in modifying the stress state of elements which change state at the beginning of the iterations.

5.5 Conclusions

The zeta-parameter method has shown some success in improving the convergence properties of the conjugate gradient method. In the spring problem, a significant

reduction in the number of iterations is seen. However, this success is not consistently achieved for large problems. The method has difficultly in selecting minimization directions which consistently approach the correct solution. At times the method overshoots the solution and requires a number of iterations to reverse direction. At other times, the method selects an unsuitable direction and the minimization routine fails to advance the solution. Further improvements are needed in the zeta-parameter method to achieve a technique which will consistently accelerate the convergence of the conjugate gradient method.

Table 5.1: Number of Iterations per Load Step for Cantilever Beam

Load Step	Time (sec.)	Basic C.G.	C.G. with Diag. Scaling	C.G. with Zeta-Param.	C.G. with Scaling & Zeta
1	0.036	26	30	26	30
2	0.072	28	28	28	28
3	0.108	31	25	31	25
4	0.144	31	23	31	23
5	0.180	33	23	33	23
6	0.216	34	23	33	26
7	0.252	34	27	34	27
8	0.288	44	25	44	26
9	0.324	45	31	45	31
10	0.360	45	36	47	36
11	0.396	50	34	49	34
12	0.432	54	37	52	39
13	0.468	59	40	56	40
14	0.504	67	46	68	46
15	0.540	74	54	72	54
16	0.576	100	60	104	60
17	0.612	103	61	105	60
18	0.648	153	70	163	70
19	0.684	137	75	136	72
20	0.720	232	113	237	113
21	0.756	230	150	224	149
22	0.792	183	132	181	130
23	0.828	173	125	188	123
24	0.864	146	109	146	111
25	0.900	156	107	146	106

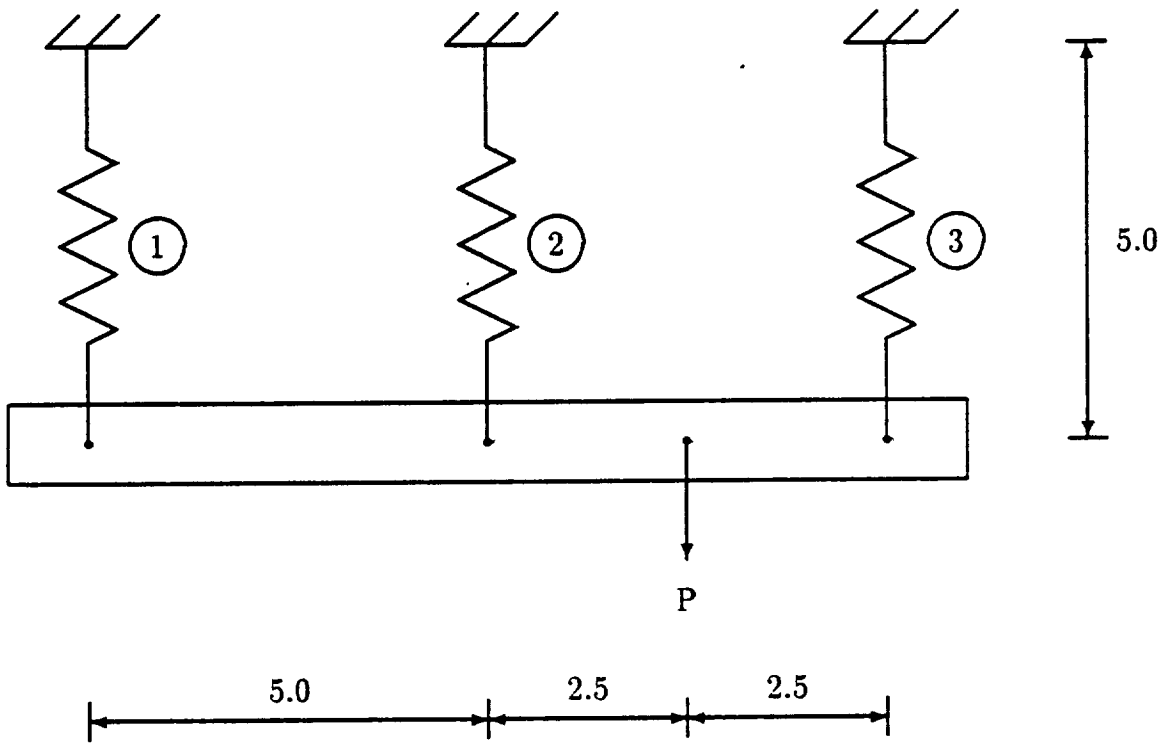


Figure 5.1: Configuration of Spring-Supported Bar Problem

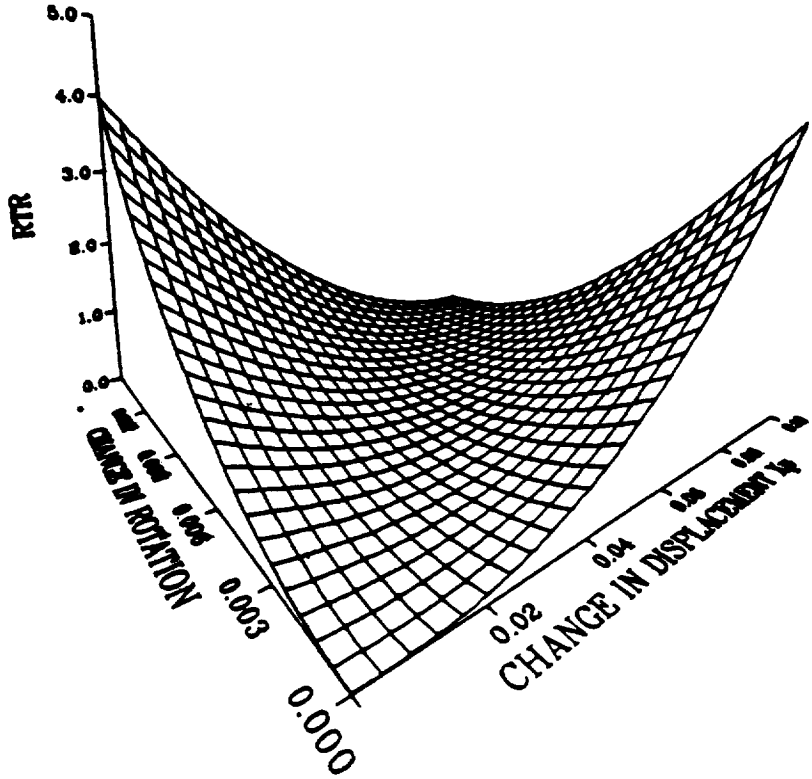


Figure 5.2: 3D Contours of Residual in Stage 2 of Spring Problem

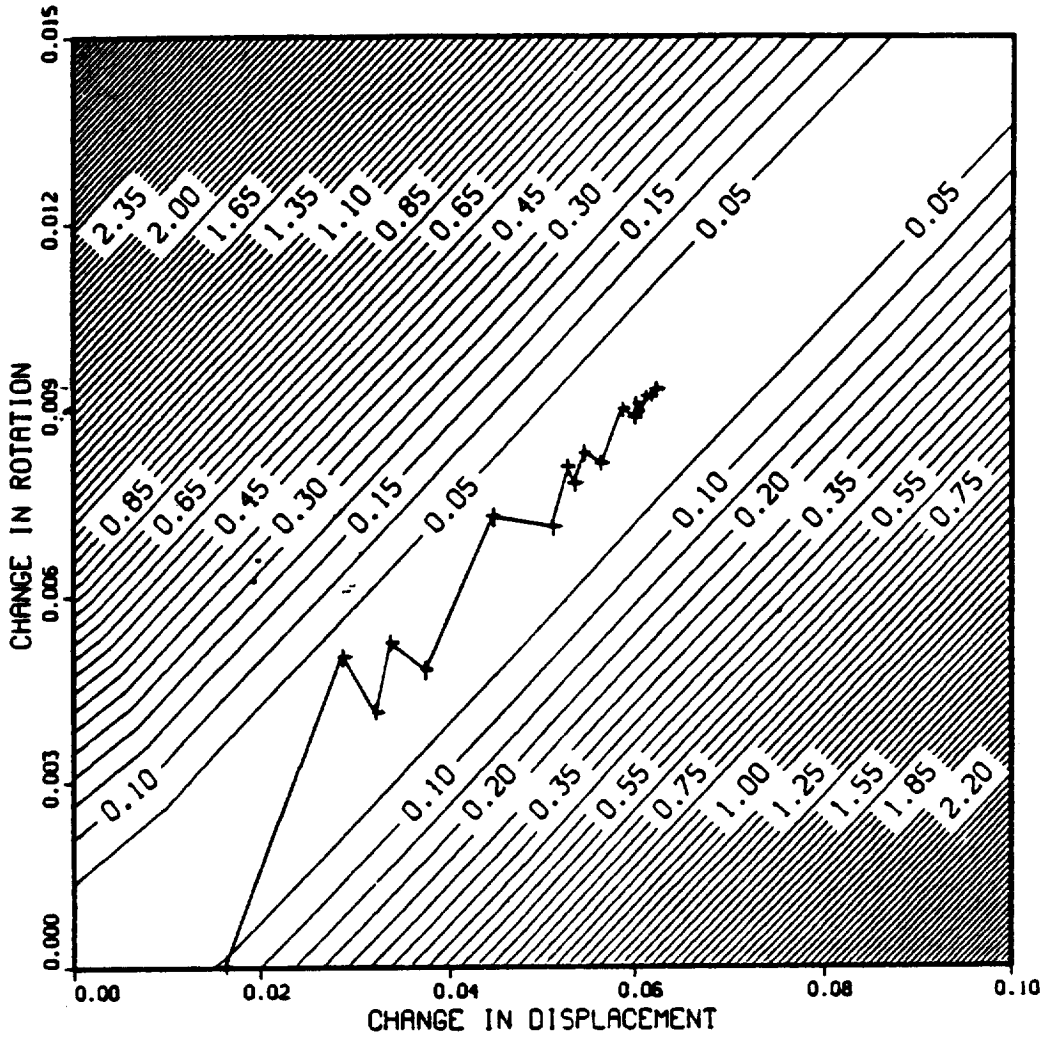


Figure 5.3: 2D Contours for Stage 2 Using Basic Conjugate Gradient

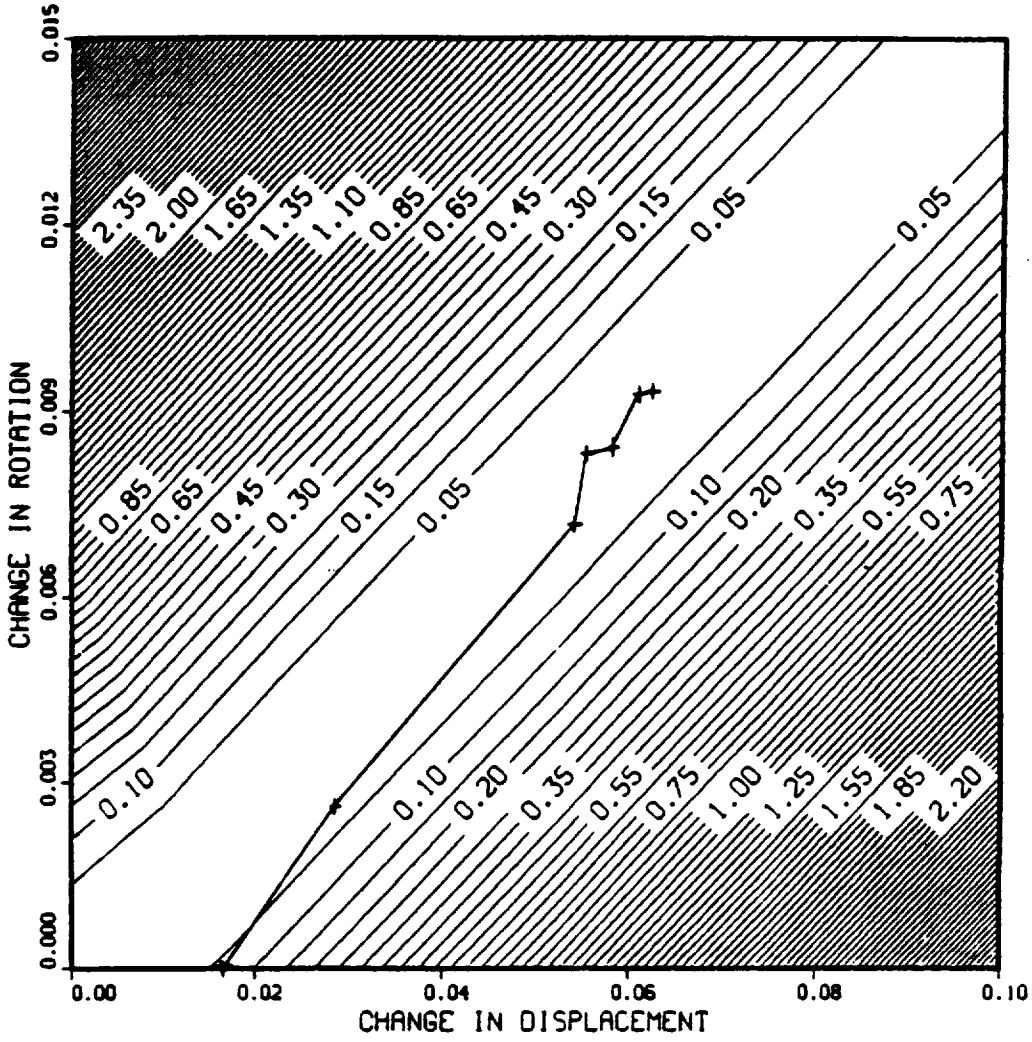


Figure 5.4: 2D Contours for Stage 2 Using Diagonal Scaling

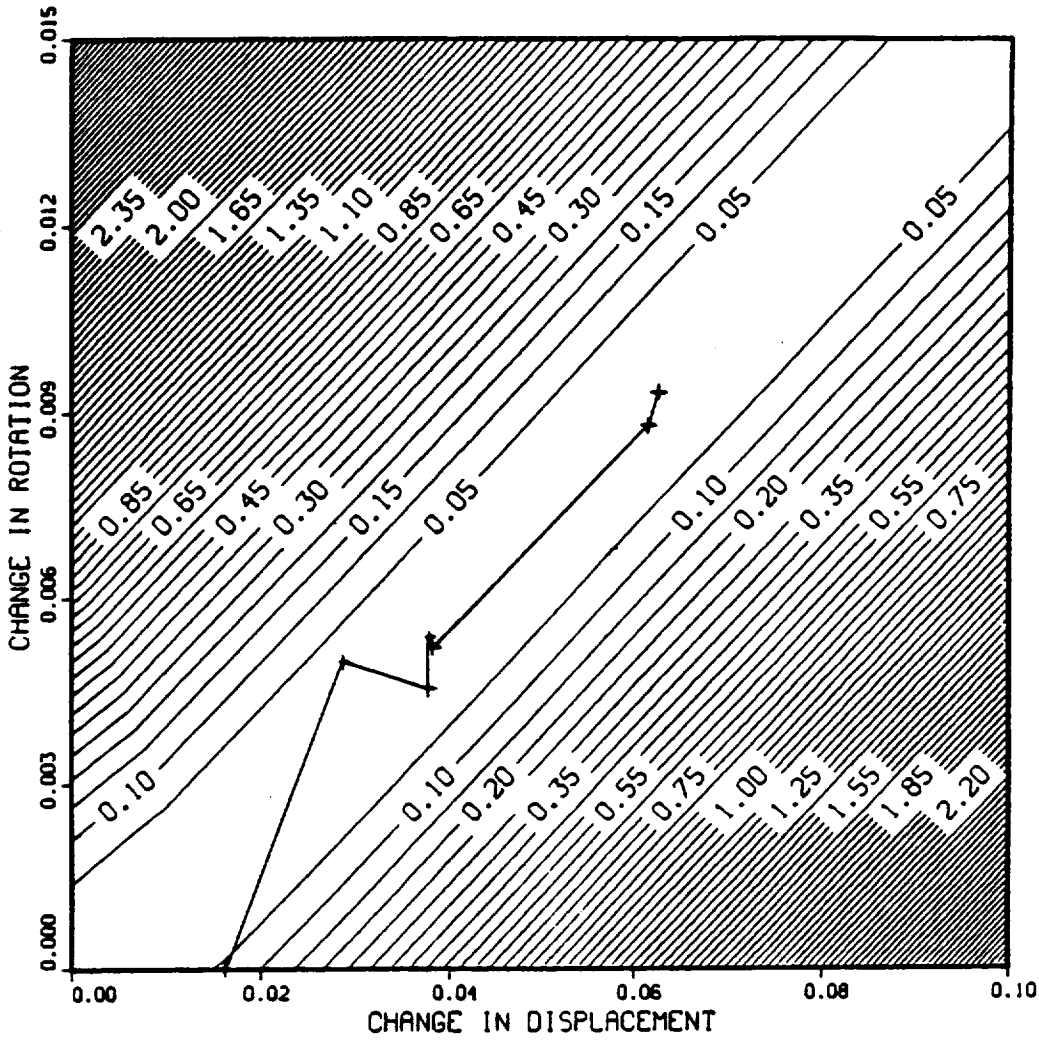


Figure 5.5: 2D Contours for Stage 2 Using Zeta-Parameter Method

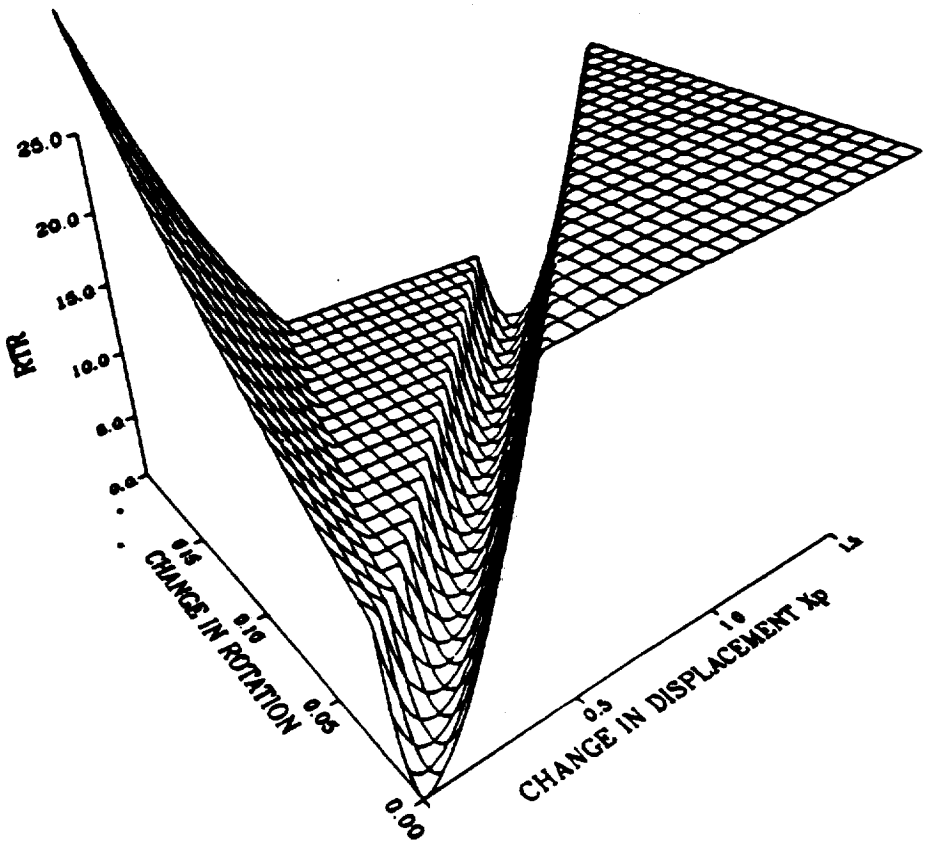


Figure 5.6: 3D Contours of Residual in Stage 4 of Spring Problem

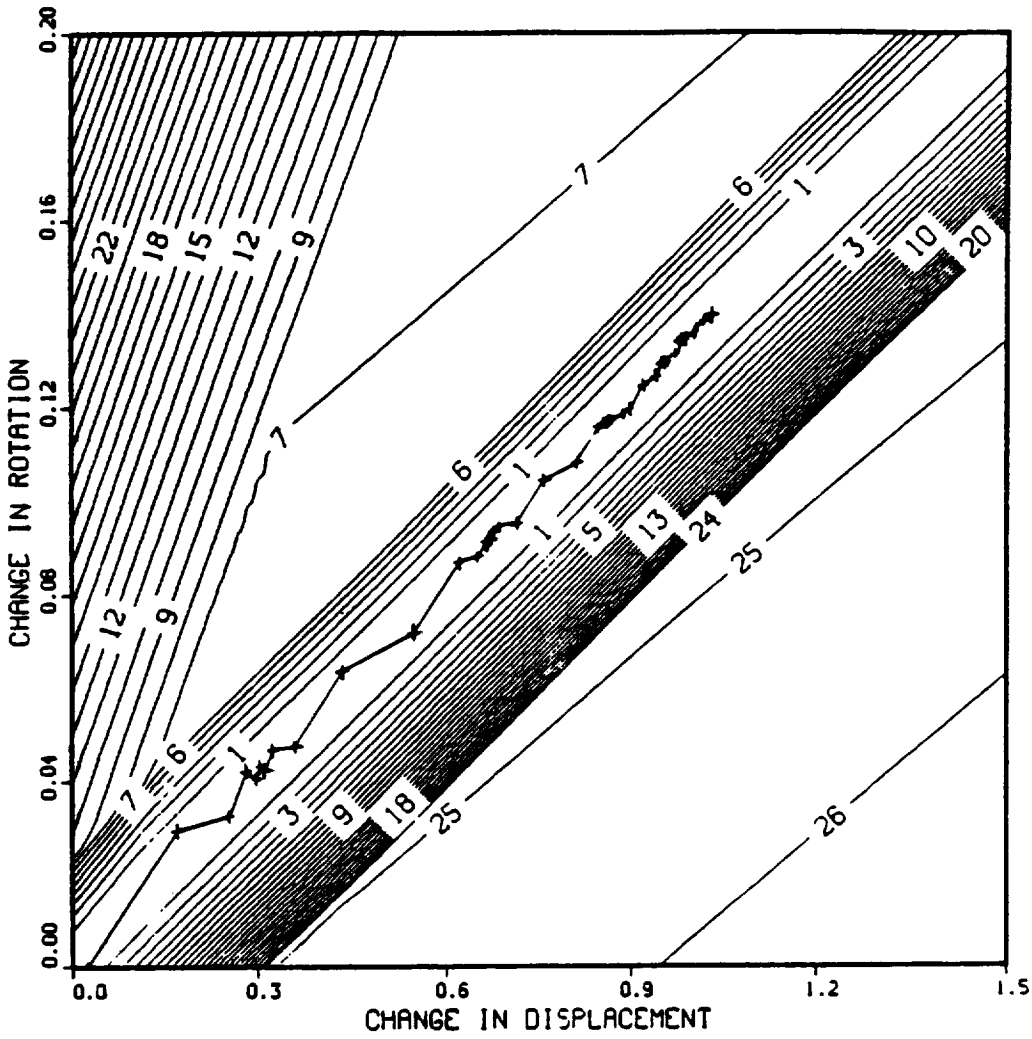


Figure 5.7: 2D Contours for Stage 4 Using Basic Conjugate Gradient

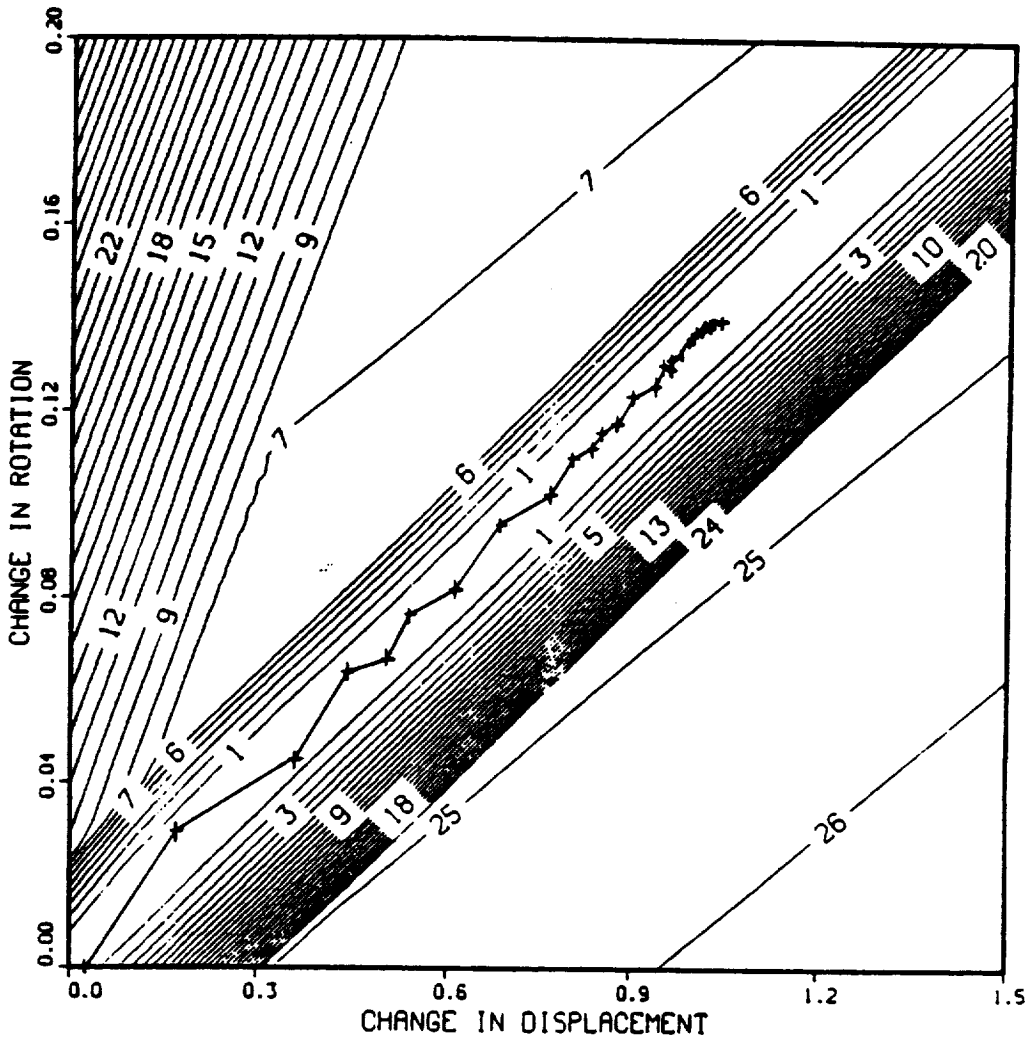


Figure 5.8: 2D Contours for Stage 4 Using Diagonal Scaling

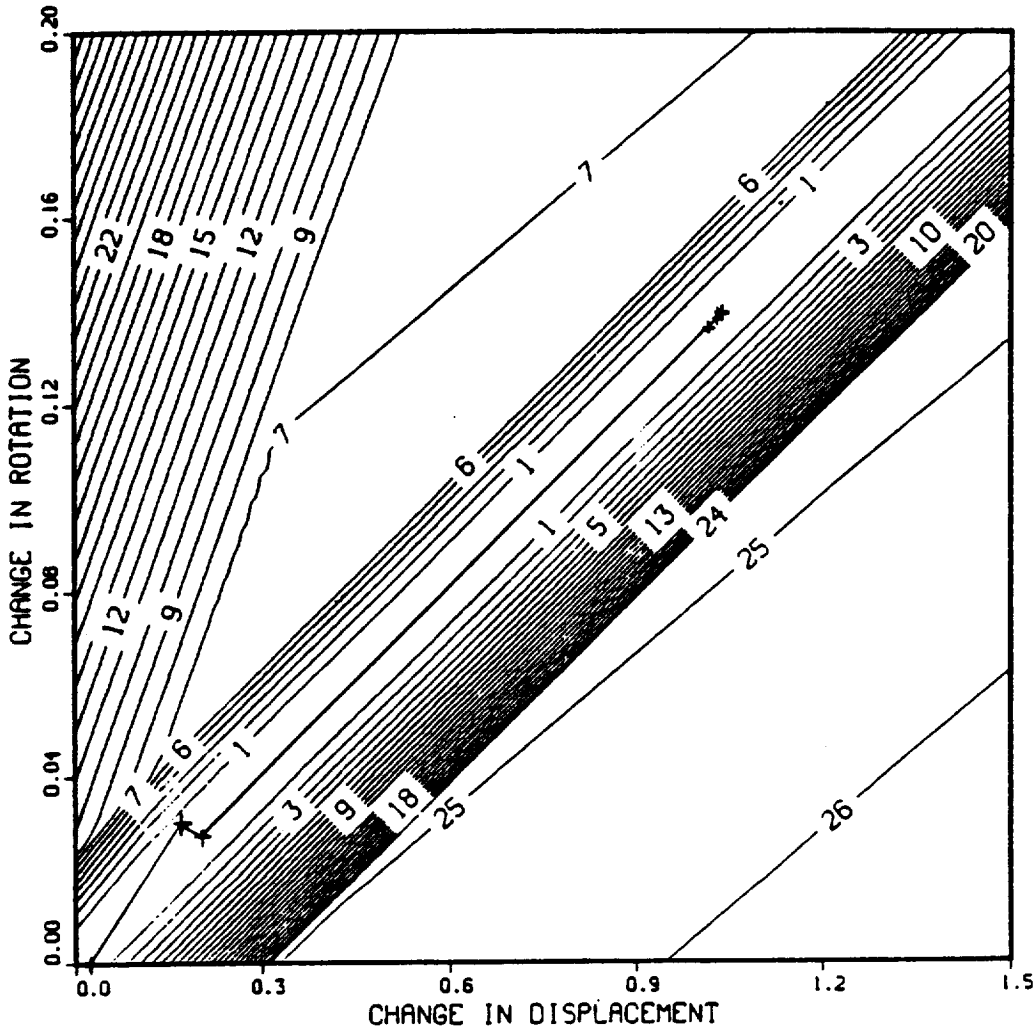


Figure 5.9: 2D Contours for Stage 4 Using Zeta-Parameter Method

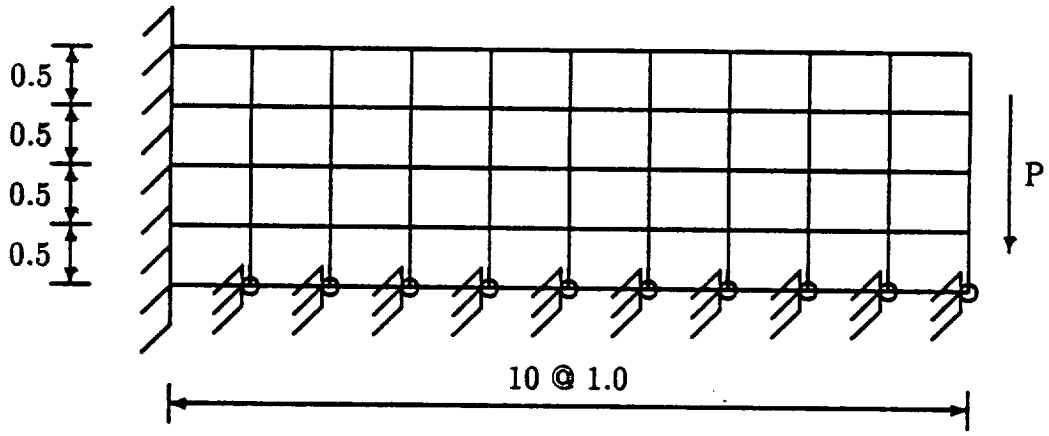


Figure 5.10: Configuration of Confined Cantilever Beam

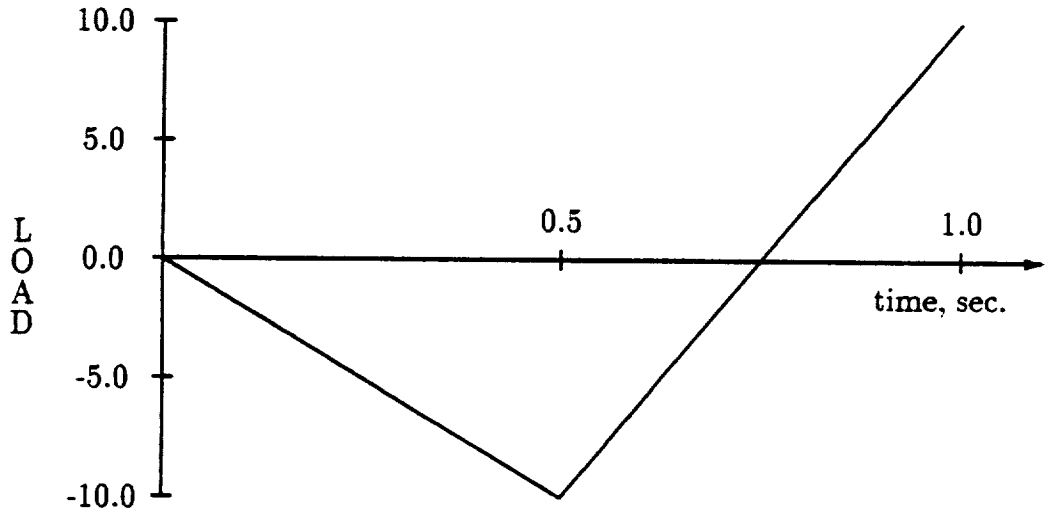


Figure 5.11: Load-Time Curve for Cantilever Beam Problem

Chapter 6

Summary and Conclusions

Vectorization and concurrency significantly speed up the execution of explicit finite element programs. A comparison of run times between the original version of the code using explicit time integration and the fully vectorized, concurrent version of the code shows speed-ups of over 25. The implementation of subcycling increases the speed-up by an average factor of 2. Note the speed-up due to subcycling is problem dependent and will vary according to the size of the problem and the range and distribution of element sizes and types.

One of the most significant factors in reducing the effectiveness of parallel processing is the problem of processor idleness. Because vectorization and concurrency tend to be competing processes, an algorithm was developed to optimized the block size based on the size of the problem, the number of processors and the number of different element types in the mesh. This minimizes the amount of time in which a processor may remain idle and also reduces the possibility of an element block containing a small number of elements.

Another algorithm was developed to minimize processor idleness caused by a small number of blocks subcycling in a mixed integration solution. The algorithm determined the element blocks which could be updated early if a processor was expected to remain idle during a subcycle. The increase in speed-up due to the implementation of this algorithm was 30 % for eight processors.

An additional factor which decreases the efficiency of parallel execution is memory contention. Memory contention problems occur when more than one processor attempts to access a shared memory location simultaneously. Locks which prevent simultaneous access may create a slowdown if substantial interference exists.

Finally, the zeta-parameter method was effective in improving convergence properties of the conjugate gradient method for the one-dimensional spring problem. However, the technique does not consistently produce directions which accelerate convergence for a wide range of problem types.

Bibliography

[1] Belytschko, T., "Partitioned and Adaptive Algorithms for Explicit Time Integration," *Nonlinear Finite Element Analysis in Structural Mechanics,* ed. by W. Wunderlich, E. Stein, and K.-J. Bathe, Springer-Verlag, Berlin, 1980, pp. 572-584.

[2] Belytschko, T., "Explicit Time Integration of Structural-Mechanical Systems," *Advanced Structural Dynamics,* Donea, J., ed., Applied Science Publishers, Essex, England, 1980, pp. 97-122.

[3] Belytschko, Ted, and Lin, Jerry, I., "Eigenvalues and Stable Time Steps for the Bilinear Mindlin Plate Element," *International Journal for Numerical Methods in Engineering,* Vol. 21, 1985, pp. 1729-1745.

[4] Belytschko, Ted, Lin, Jerry I., and Tsay, Chen-Shyh, "Explicit Algorithms for the Nonlinear Dynamics of Shells," *Computer Methods in Applied Mechanics and Engineering,* Vol. 42, 1984, pp. 225-251.

[5] Belytschko, T. and Liu, W. K., "Time Integration with Explicit/Explicit Partitions in EPIC-2", Report to Ballistics Research Laboratory, July 1982.

[6] Belytschko, Ted, Smolinski, Patrick and Liu, Wing Kam, "Stability of Multi-Time Step Partitioned Integrators for First-Order Finite Element Systems," *Computer Methods in Applied Mechanics and Engineering,* Vol. 49, 1985, pp. 281-297.

[7] Belytschko, T. and Tsay, C. S., "WHAMSE: A Program for Three-Dimensional Nonlinear Structural Dynamics," EPRI Report NP-2250, Palo Alto, CA, February 1982.

[8] Belytschko, T., Yen, H. J., and Mullen, R., "Mixed Methods for Time Integration," *Computer Methods in Applied Mechanics and Engineering,* Vol. 97, 1986, pp. 1-24.

[9] Biffle, J.H., "JAC - A Two-Dimensional Finite Element Computer Program for the Non-Linear Quasistatic Response of Solids with the Conjugate Gradient Method," Sandia Report SAND81-0998, Sandia National Laboratories, Albuquerque, NM, April 1984.

[10] Boyle, James, Ralph Butler, Terrence Disz, Barnett Glickfeld, Ewing Lusk, Ross Overbeek, James Patterson, Rick Stevens, *Portable Programs for Parallel Processors,* Holt, Rinehart and Winston, Inc., 1987.

[11] Cook, Robert D., *Concepts and Applications of Finite Element Analysis,* John Wiley & Sons, Inc., 1974.

[12] Flanagan, D. P., and Belytschko, T., "Eigenvalues and Stable Time Steps for the Uniform Strain Hexahedron and Quadrilateral," *Journal of Applied Mechanics,* Vol. 51, March 1984, pp. 35-40.

[13] Flanagan, Dennis. P., and Taylor, Lee. M., "Structuring Data for Concurrent Vectorized Processing in a Transient Dynamics Finite Element Program," *Parallel Computations and Their Impact on Mechanics,* ed. Ahmed K. Noor, The American Society of Mechanical Engineers, New York, NY, 1987, pp. 291-299.

[14] Fletcher, R., and Reeves, C.M., "Function Minimization by Conjugate Gradients," *Computer Journal,* Vol. 7, 1964, pp. 149-154.

[15] Kennedy, J. M., Belytschko, T., and Lin, J. I., "Recent Developments in Explicit Finite Element Techniques and Their Application to Reactor Structures," *Nuclear Engineering and Design,* Vol. 97, 1986, pp. 1-24.

[16] Hughes, Thomas J.R., "Numerical Implementation of Constitutive Models: Rate-Independent Deviatoric Plasticity," *Proceedings of the Workshop on the Theoretical Foundation for Large-Scale Computations of Nonlinear Behavior,* Siavouche Nemat-Nasser (ed.), Martinus Nijhoff Publishers, 1984, pp. 29-63.

[17] Hughes, T. J. R., Cohen, M., and Haroun, M., "Reduced and Selective Integration Techniques in Finite Element Analysis of Plates," *Nuclear Engineering and Design,* Vol. 46, 1978, pp. 203-222.

[18] Hughes, Thomas J. R., Ferencz, Robert M., and Hallquist, John O., "Large-Scale Vectorized Implicit Calculations in Solid Mechanics on a CRAY X-MP/48 Utilizing EBE Preconditioned Conjugate Gradients," *Computer Methods in Applied Mechanics and Engineering,* Vol. 61, 1987, pp. 215-248.

[19] Hwang, Kai and Briggs, Fayé A., *Computer Architecture and Parallel Processing,* Mc-Graw-Hill, Inc., 1984.

[20] Klessig, R., and Polak, E., *Efficient Implementations of the Polak-Ribière Conjugate Gradient Algorithm,* SIAM J. Control, Vol. 10, 1972, pp. 524-549.

[21] Lusk, Ewing L., and Overbeek, Ross A.,"Implementation of Monitors with Macros: A Programming Aid for the HEP and Other Parallel Processors," Technical Report ANL-83-97, Argonne National Laboratory, Argonne, Illinois, December 1983.

[22] Lusk, E.L. and Overbeek, R.A., "Use of Monitors in FORTRAN: a Tutorial on the Barrier, Self-Scheduling Do-Loop, and Askfor Monitors," *Parallel MIMD Computation: The HEP Supercomputer and its Applications,* ed. J. S. Kowalik, The MIT Press, 1985.

[23] Papadrakakis, Manolis, and Ghionis, Panaghiotis, "Conjugate Gradient Algorithms in Nonlinear Structural Analysis Problems," *Computer Methods in Applied Mechanics and Engineering,* Vol. 59, 1986, pp. 11-27.

[24] Polak, E., and Ribière, "Note sur la convergence de methodes de directions conjugées," *Revue Française Inf. Rech. Oper.*, 16 RI 1969, pp. 35-43.

[25] Press, William H., Flannery, Brian P., Teukolsky, Saul A., and Vetterling, William T., *Numerical Recipes,* Cambridge University Press, 1986.

[26] Schendel, U., *Introduction to Numerical Methods for Parallel Computers,* Ellis Horwood Limited, 1984.

[27] NCSA Summer Institute, Training Information on Vectorization, University of Illinois, Urbana, IL, January 1987.

[28] Smolinski, Patrick, Belytschko, Ted, and Liu, Wing Kam, "Stability of Multi-Time Step Partitioned Transient Analysis for First-Order Systems of Equations," *Computer Methods in Applied Mechanics and Engineering,* Vol. 65, 1987, pp. 115-125.

[29] *Using the Alliant FX/8*, Mathematics and Computer Science Division, Technical Memorandum No. 69, ANL/MCS-TM-69,Rev. 1, Argonne National Laboratory, September 1986.

[30] Yamada, Y., Yoshimura, N., and Sakurai, T., "Plastic Stress-Strain Matrix and its Applications for the Solution of Elastic-Plastic Problems by the Finite Element Method," *International Journal of Mechanical Sciences,* Vol. 10, 1968, pp. 561-578.