Group Implicit Concurrent Algorithms in Nonlinear Structural Dynamics

by

M. Ortiz and E. D. Sotelino
Division of Engineering
Brown University
Providence, RI 02912

NAG1-634

NASA Contract: NAG-1-634

Final Report

October 1989

Table of Contents

CHAPTER I

Introduction

During the 70's and early 80's, considerable effort was devoted to developing efficient and reliable time-stepping procedures for transient *structural* analysis. Mathematically, the equations governing this type of problems are generally *stiff*, i. e., they exhibit a wide spectrum in the linear range. For instance, in thermal analysis problems, the presence of materials with widely differing conductivities, such as steel and concrete, may contribute to the spread in eigenvalues. As a further example in the area of structural vibrations, the bending stiffness of beams is typically much smaller than their axial stiffness, which again tends to give widely varying eigenfrequencies. Another key characteristic of structural problems is that, in most areas of application, the response lies in the lower part of the spectrum. Typical examples are: earthquake engineering, fluid/structure interaction problems in reservoirs, and others.

The algorithms best suited to this type of applications are those which accurately integrate the low frequency content of the response without necessitating the resolution of the high frequency modes. This inevitably means that the algorithm must be unconditionally stable, which in turn rules out explicit integration. Thus, the early work in the area was primarily geared to developing unconditionally stable time-stepping algorithms for linear and nonlinear applications. The most prominent example of that class of algorithms, and one which played a central role in all subsequent developments, is Newmark's method.

Within its range of unconditional stability, Newmark's method is *implicit.* In typical large scale applications involving nonlinear structures, the cost of Newmark's algorithm is dominated by the equation solving phase. More recent research has endeavored to alleviate this source of computational cost while retaining the requisite stability of the algorithm. Examples of contributions in this direction are implicit/explicit partition methods [1], staggered procedures for coupled problems [2], the method of alternating directions [3], and semi-implicit procedures such as Trujillo's algorithm [4] and element-by-element methods [5,6].

However, by far the most exciting possibility in the algorithm development area in recent years has been the advent of parallel computers with multiprocessing capabilities. Considerable research is presently underway to replace the traditional algorithms devised for sequential machines by others well suited to parallel computing. In this work, we are mainly concerned with the developement of parallel algorithms in the area of structural dynamics. Thus, a primary objective is to devise unconditionally stable and accurate time-stepping procedures which lend themselves to an efficient implementation in concurrent machines. Following a succinct summary in Chapter II of some features of the new computer architectures which bear on subsequent discussions, and a brief overview in Chapter III of current research efforts in the area, a new class of concurrent procedures, or Group Implicit (GI) algorithms, is introduced and analyzed in Chapter IV. Our numerical simulations show that GI algorithms hold considerable promise for application in coarse-grain as well as medium-grain parallel computers. Examples of such computers are the Alliant series, the iPSC Hypercube computer, the ETA machine and the Cray series.

References

1. T. Belytschko, and T. J. R. Hughes, 'Mesh Partitions of Explicit-Implicit Time Integration,' *Formulations and Computational Algorithms in Finite Element Analysis*, K. Bathe, J. Oden, and W. Wunderlich, eds., MIT Press, Cambridge, Mass., 1976, pp. 673-690.

2. C. A. Felippa, and K. C. Park, 'Staggered Solution Transient Analysis Procedures for Coupled Mechanical Systems: Formulation,' *Computer Methods in Applied Mechanics and Engineering*, Vol. 24, 1980, pp. 61-111.

3. T. J. R. Hughes, J. Winget, I. Levi, and T. E. Tezduyar, 'New Alternating Direction Procedures in Finite Element Analysis Based Upon EBE Approximate Factorizations,' *Recent Developments in Computer Methods for Nonlinear Solid and Structural Mechanics*, N. Perrone, S. Atluri, eds., ASME, New York, June 1983, pp. 75-109.

4. D. H. Trujillo, 'An Unconditionally Stable Explicit Algorithm for Structural Dynamics,' *Int. Journal for Numerical Methods in Engineering*, Vol. 11, 1977, pp. 1579-1592.

5. M. Ortiz, P. M. Pinsky, and R. L. Taylor, 'Unconditionally Stable Element-by-Element Algorithms for Dynamic Problems,' *Computer Methods in Applied Mechanics and Engineering*, Vol. 36, No. 2, 1983, pp. 223-239.

6. T. J. R. Hughes, I. Levi, and J. Winget, 'An Element-by-Element Solution Algorithm for Problems of Structural and Solid Mechanics,' *Computer Methods in Applied Mechanics and Engineering*, Vol. 36, No. 2, 1983, pp. 241-254.

CHAPTER II

Survey of Present Parallel Architectures

This chapter summarizes the state of the art in the area of parallel architectures. Firstly, a brief history of parallel processing is given, followed by a survey of recent advances in parallel computers. It bears emphasis that parallel processing is a rapidly evolving field and the focus of intensive research worldwide. Over one hundred projects on parallel architectures are under way in the United States universities and industry [1].

2.1. Brief History of Parallel Processing

In the last 40 years *sequential (serial)* architectures have dominated the computer architecture world. During this period several improvements, in terms of computing speed, have been achieved, mostly due to increasingly faster electronic components. However, this avenue for progress is limited by a simple fact of Physics: "No signal can travel faster than the speed of light in the vacuum". Thus, while the eletronic components themselves may become increasingly faster, the computer itself is not. Parallel Processing is widely viewed as a solution to this problem. By performing several subtasks concurrently, the total time required to perform the combined task is reduced.

Even though Parallel Processing is a phenomenon of the 80's, the idea of using parallelism can be traced back to three computers developed in the late 60's [2], namely: ILLIAC IV, CDC Star100 and Texas Instrument ASC. Nonetheless, these computers are not based on the same type of architecture. The ILLIAC IV is an *array processor*, while the other two are *vector processors.*

The *array processors* [3] are very useful in handling operations with matrices. They are constituted of a control processor and arithmetic processors. Its operation is initiated by the control processor fetching an instruction and determining if it is a matrix operation or not. If it is not then it performs it, otherwise it passes the instruction onto the arithmetic processors, with each processor holding the part of the matrix being operated inside its own local memory. Since all processors receive instructions from the control processor simultaneouly, the entire matrix operation proceeds in parallel. The FPS-164 Scientific Computer [1] is an attached processor with an architecture based on array processor technology [4]. In order to offload the interactive parts of the design and analysis in engineering projects, Swanson Analysis System Inc., added the FPS-164 attached processor to its DEC Vax-11/780 superminicomputer [5], making of it a more efficient machine.

The *vector processors* are also know as *pipeline processors* [3]. In these computers a particular vector instruction operates in a sequence of operands (a vector) rather than on single operands. Vector processing involves a technique known as pipelining, that can be understood simply with reference to an assembly line. Each stage in the pipeline always performs the same subtask to different operands that go through it. This technique is designed to exploit the bandwidth data movement outside the pipe in order to keep the pipeline saturated. One of the earliest vector processors is the CDC Star100 [6] [2]. Its central processor has a pipeline arithmetic unit, which segments arithmetic computations into a sequence of basic operations.

[1] *Floating Point System, Inc.*

[2] *Control Data Corporation*

The arithmetic unit can perform basic operations simultaneously on independent pair of data elements to stream through the pipeline. The Texas Instruments Advanced Scientific Computer (ASC) [3] is another example of this first generation of vector processing supercomputers [7]. The next generation of supercomputers that exploits vector processing is exemplified by: the CRAY-1 [4], which contains 13 independent pipelines referred to as functional units (to carry out a specific task: multiplication, addition and logical operations), and the CDC Cyber 205 [5], which consists of up to 4 pipelines, each of which performs a variety of operations.

Both array processors and vector processors are SIMD machines (section 2.2.5). For these machines, the same instruction is executed simultaneously by all processors on different sets of data.

In a wider sense, some degree of parallelism can be found in other early computers, in so much as different components could operate concurrently. For instance, while the central processing unit is busy performing computations, the input could be read in or the output printed out, by the appropriate devices.

Aside from these limited uses of parallelism, true multiprocessing computers only began to become widely available in the market and to be the subject of extensive research over the past decade. Multiprocessing is achieved in MIMD machines (section 2.2.6). For these, all processors execute simultaneously different intructions on different sets of data. One of the first computers in this category was built by linking together two SIMD processors [3], by Cray Research Inc. The CRAY-XMP consists of two redesigned versions of CRAY-1 supercomputers placed back-to-back, which can communicate via a cluster of very faster registers.

[3] *Texas Instrument, Inc.*
[4] *Cray Research, Inc.*
[5] *Control Data, Inc.*

A fundamental characteristic of a Parallel Processing system is its *granularity*, [8]. The granularity of a system is the size of the units of work allocated to each processor. Coarse-grain parallelism involves computational processes at the outermost level of program control and implies a small number of large and complex processors. On the other hand, in fine-grain parallelism the unit of work is the execution of a statement and it implies a large number of small and simple processors. The medium-grain parallelism consist of the cases between the other two extremes.

Lincoln [9] points out that a major choice confronting computer architectures is the degree of which they can be considered *general purpose* or *special purpose*. In the *general purpose* category several different parallel architectures have appeared ranging from super computers with only a few powerful processors, i. e. coarse grained systems (CRAY XMP, CRAY 2, and ETA-10) to massively parallel computers with thousands of processors, i. e. fine-grained systems (Connection Machine of Thinking Machine Inc., with up to 65,536 processors). Other architectures (Alliant FX8, Intel iPSC hypercube) fall between these two extremes, i. e., while not qualifying as supercomputers, their processors are much more powerful than those in the massively parallel machines (medium-grained systems). Even though they are classified as general purpose machines, they are suited for solving specifically scientific problems, due to the programming languages available in them. *Special purpose* machines are designed to solve a particular problem, or class of problems. Norrie [3] divide them into two categories:

1) The architecture is modeled to reflect the physical structure of the problem to be solved. An example would be the Finite Element Machine which is a research computer built at Nasa's Langley Research Center [10]. It consists of a minicomputer front end, called controller, attached to a MIMD array of

asynchronous microcomputers, referred to as the array. There is no shared memory in the system and each processor runs its own program on its own data. An additional circuitry provides a rich interconnection environment for communication and cooperative computation. The basic idea is that each processor is assigned to each node of the finite element grid. Each processor is connected to its immediate eight neighbors [11], and all the processors in the systems are conneted through a global bus. Very fast results can be achieved in this machine, but difficulties arise when the physical problem structure is altered.

2) The architecture is designed to reflect the general solution method for that class of problems. An example is the Parfem, a parallel finite element machine developed at University of Calgary, Canada [3]. The generator of element stiffness matrices, or Gen, is an array-type processor. Programmed in the controller are the algorithms to be used for generating the stiffness matrices and the algorithm for determining the order in which the stiffness matrices are to be generated. The system-matrix assembler, or Ass, is a vector processor, while the actual architecture of the Solut, equation solver, has not yet been established.

Since special purpose machines perform specific tasks, a general approach for this kind of technology is rare. Nonetheless, Kung [12] provides a general guideline by introducing the concept of systolic architecture, which is a methodology for mapping high-level computations into hardware structures. Law [13] defines systolic architectures as: "devices attached to a conventional computer to perform a special purpose function with extreme high speed". In these machines the single processing element is replaced by an array of processors with built in hardware instruction. Several systolic algorithms have been developed, especially systolic matrix algorithms, which are useful in finite element computations.

Despite recent progress, there remains a need for more efficient processors and inter-connection networks. For example, Adams and Crocket in [10] show that floating-point arithmetic, communication and synchronization times represent a significant share of the total execution time of a parallel algorithm. On the other hand, the interface between user and parallel machine is a challenge that software specialists will have to deal with promptly. Furthermore, because of the wide variety of parallel architectures, the issue of portability is a major concern to code developers. The task of porting parallel codes from one computer to another still involves a great deal of restructuring of the algorithm in order to make it work well. Some attemps have been made to help users of parallel computers. In [14] a system is presented that helps users "fine-tune" the output of an automatic system. Another approach to portability [15] is to develop and implement an abstraction (called *monitor*) that is independent of the architecture or parallel processing primitive on any particular machine. As a final example, in [16] the design of user oriented software to support the solution of large problems by engineers and scientists using a 64-bit array processor, which shares memory with a 32-bit minicomputer is developed. It provides the user with tools to help in the creation and manipulation of large matrices using the hypermatrix scheme.

2.2. Models of Computation

Computers operate on a stream of data through a stream of instructions, i. e., a computer program is a sequence of instructions which modifies a set of data. According to the nature of these streams, computers can be classified into four categories:

- Single Instruction stream, Single Data stream (SISD)

- Multiple Instruction stream, Single Data stream (MISD)

- Single Instruction stream, Multiple Data stream (SIMD)
- Multiple Instruction stream, Multiple Data stream (MIMD)

A basic issue in parallel processing is that of the organization of the system's memory. The SIMD and MIMD models of computation, prevalent among parallel computers, can be further classified, according to how their processors communicate, into

- Tightly coupled systems or Shared Memory Computers
- Loosely coupled systems or Interconnection Network Computers

A discussion of these two forms of communication is given in sections 2.2.1 and 2.2.2. In sections 2.2.3 to 2.2.6 the models of computation are described.

2.2.1. Shared Memory Computers

Shared Memory computers are also known as Parallel Random Access Machines (PRAM). In this kind of computers communication between processors occurs through the common memory, via variable sharing. In other words, if processor A wants to communicate the value of a variable x to processor B, two steps must be performed. First processor A writes the value of x in its address in memory. Then processor B reads it from the same location.

While a program is being executed, all processors are allowed to access the common memory. Depending on whether more than one processor is permitted to simultaneously read from or write into memory, an additional classification can be established:

- Exclusive_Read, Exclusive_Write (EREW). Only one processor can access (read from or write into) a specific memory location at a time.

- Concurrent_Read, Exclusive_Write (CREW). Several processors are allowed to read from the same memory address simultaneously, but only one processor can write into a specific memory location at a time.

- Exclusive_Read, Concurrent_Write (ERCW). Only one processor can read from a specific memory location at a time, but several of them can write on the same address simultaneously.

- Concurrent_Read, Concurrent_Write (CRCW). Several processors are allowed to read from or write into the same location in memory at the same time.

Writing simultaneously on memory may give rise to contention problems if several processors attempt to store a different value in the same address. In this case, a decision needs to be made to select which value is to be stored. Usually, priorities can be set so that only one of the values is stored. By contrast, simultaneously reading from the same memory location does not cause contention problems, since the contents of the location is not changed as a result of the operation.

A pressing issue regarding Shared Memory computers is that, for a large number of processors, they may be either expensive to built or simply unfeasable. Akl [17] discusses this issue:

> "When one processor needs to gain access to a datum in memory, some circuitry is needed to create a path from that processor to the location in memory holding the datum. The cost of such circuitry is usually expressed as the number of logical gates required to decode the address provided by the processor. If the memory consists of M locations, then the cost of the decoding circuitry may be expressed as $f(M)$ for some cost function f. If N processors share that memory, then the cost of the decoding

circuitry climbs to $N \times f(M)$. For large N and M this may lead to prohibitively large and expensive decoding circuitry between processors and the memory".

While costs can be reduced in several ways, such as, dividing the memory into R blocks, say of M/R locations each, in practice shared memory computers have only a few processors. Some examples of such computers are the following: Denelcor Heterogeneous Element Processor (HEP) [6] with 8 processors, Alliant FX8 [7] with up to 8 processors, and Sequent Balance [8] with up to 30 processors.

2.2.2. Interconnection Network Computers

The Interconnection Network computers are an assembly of loosely coupled processors. The communication between processors is entirely done via a communication network. Depending on the nature of this network, several different architectures can be achieved. The ideal network is that in which each processor is connected to all others. In this case, the communication is immediate between any two pairs of processors. However for a large number of processors the ideal network is unfeasible, since the total number of lines to interconnect N processors is $N(N-1)/2$ ($N-1$ lines leave each processor). In addition, the physical size of each of the processors limits the number of connections that can be made to it. Several communication networks based on direct communication between sets of processors have been proposed. Some of them are described next.

1) *Linear Array.* Here each processor P_i is connected to its neighbors P_{i-1} and P_{i+1} through a two-way communication line. The processors on the extremes,

[6] *Denelcor, Inc.*
[7] *Alliant Computer Systems Corporation*
[8] *Sequent Computer Systems, Inc.*

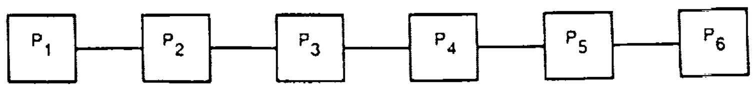


Figure 1. Linear array connection, [17].

i. e., P_0 and P_N have only one neighbor and so only one line is connected to each one of them. Figure 1 exemplifies this network for $N = 6$

2) *Two-Dimensional Array.* Here the processors are arranged in a 2-D array of $N^{1/2}$ by $N^{1/2}$ elements. The processor in row i and column j, called $P_{i,j}$, is connected to its neighbors: $P_{i-1,j}, P_{i+1,j}, P_{i,j-1}$ and $P_{i,j+1}$. The processors located on the extreme rows and/or columns will have only two or three neighbors. Figure 2 exemplifies this network for $N = 3$.

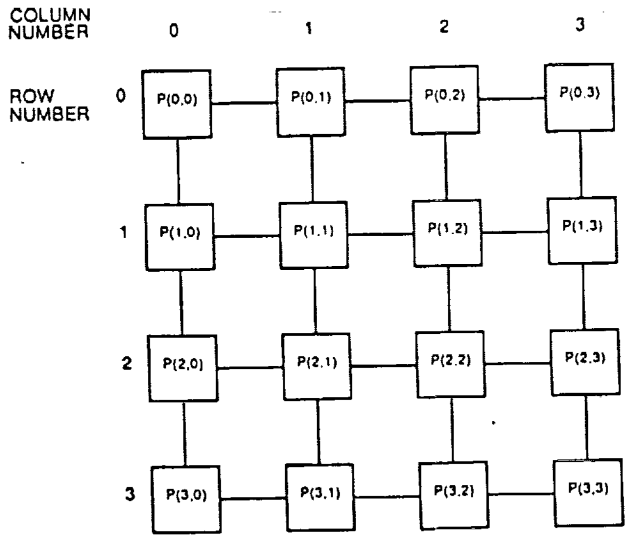


Figure 2. Two-dimensional array connection, [17].

3) *Cube Connection.* Here the total number of processors is $N = 2^q$, where q

is an integer greater or equal to one. Each processor is connected to q others so as to form a q-dimensional cube or hypercube. The neighbors of P_i, say P_j, are obtained as follows: the binary representation of j with q bits is obtained from the binary representation, also with q bits, of i differing only in one single bit. Figure 3 exemplifies the hypercube network for $q = 0, 1, 2$, and 3. For the processors that are not connected directly, communication is done via its neighbors. In this case it will take at most q steps for a processor to communicate with another.

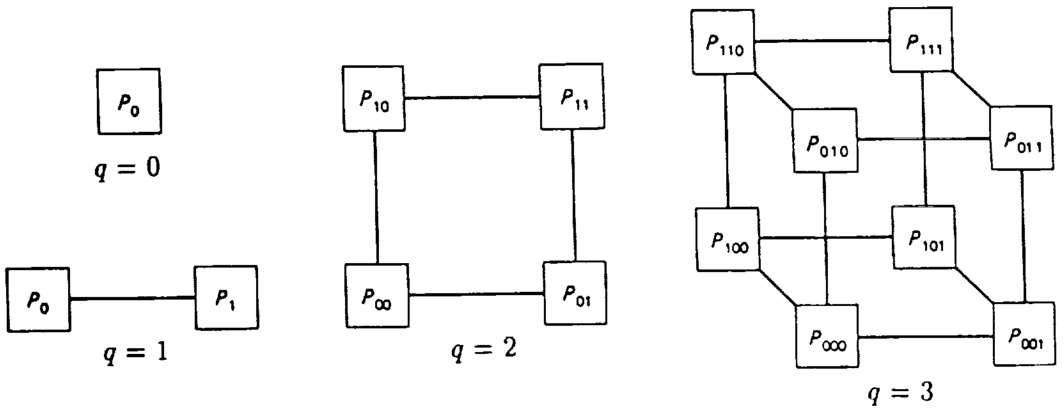


Figure 3. Cube connection: $q = 0, 1, 2$, and 3, [8].

4) *Tree Connection.* Here the processors are arranged as the nodes of a complete binary tree. Thus, if the tree has d levels then the number of nodes is $N = 2^d - 1$. Each node in the tree is a processor. Each processor in level i is connected to its parent at level $i + 1$ and to its two children at level $i - 1$. Evidently, the processor in the root is connected only to its children and the processors in the leaves are connected only to their parents. Figure 4 exemplifies this network for $d = 4$.

There is a wide variety of possible interconnection networks. The above are

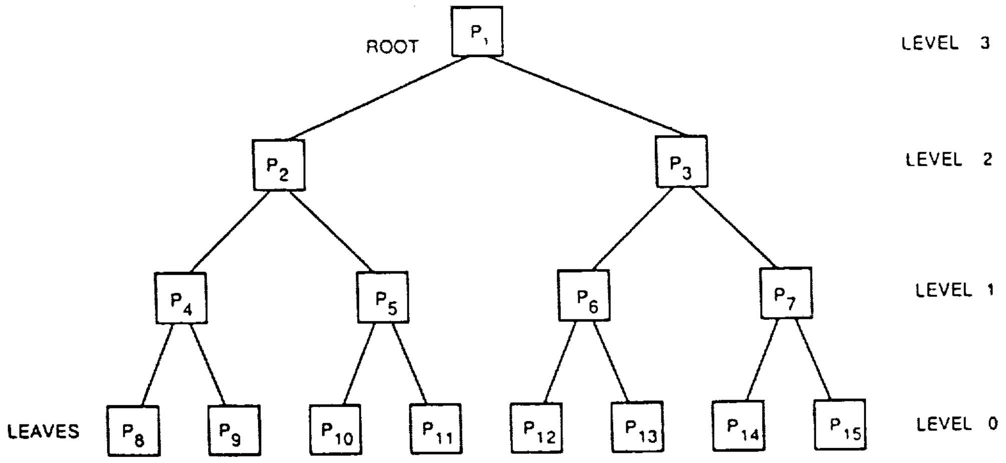


Figure 4. Tree connection, [17].

just a sample. The choice of which one of them to use depends on the application, the number of available processors, the computations themselves and the desired speed-up.

The number of processors in Interconnection Networks computers is typically much higher than in Shared Memory computers. Some examples of the former are: Caltech Hypercube(cube connected with $q = 6$, i. e., 64 processors), Intel iPSC [9] (cube connected with $q = 5, 6$ or 7, i. e., 32, 64 or 128 processors, and Connection Machine [10] (cube connected, containing 65,536 processors).

2.2.3. SISD Model

This model consists of one single processor that receives a single stream of operations which modify a single stream of data, Figure 5.

The control sends an instruction to be executed, i. e., an arithmetic operation,

[9] *Intel Corporation*
[10] *Thinking Machine, Inc.*

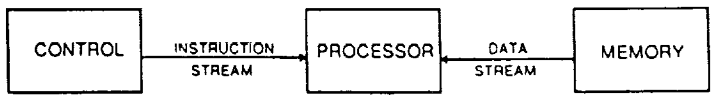


Figure 5. SISD computer, [17].

on a specific datum that is stored in memory. No parallelism is possible in this model, since it contains one processor only. Most conventional computers fall into this category.

2.2.4. MISD Model

In this model, N processors perform different streams of instructions on the same stream of data, Figure 6.

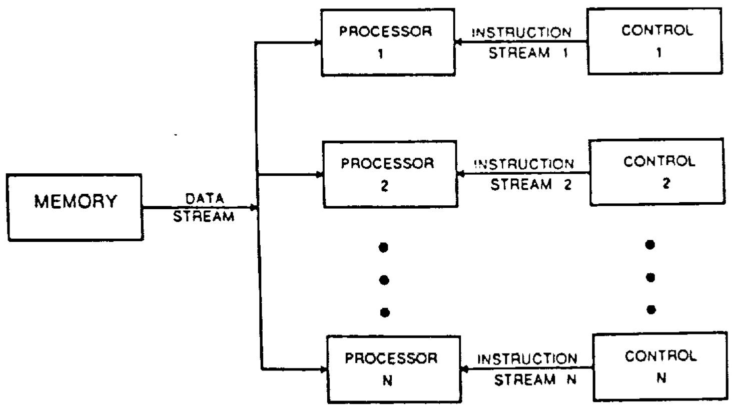


Figure 6. MISD computer, [17].

This computer architecture is useful when the same data is to be used for several different computations. An example of the kind of problem that is amenable to efficient solution in a MISD computer is that of determining if a number is prime [17]. In this case, each processor divides the number by a possible divisor (any number between 1 and the number itself), issuing a flag in case it succeeds and thereby stopping the process. The class of problems that can be solved efficiently in MISD computers is very limited. The main disadvantage of these machines is the fact that the data cannot be modified by the processors. This is a very stringent limitation in many fields of application.

2.2.5. SIMD Model

The computers classified under the SIMD model contain N processors. Each processor contains its own local memory, where programs and data are stored. All processors operate under the same control unit, which issues the same instruction to all of them to be performed on a different data set. In this model all processors operate synchronously, Figure 7.

The level of complexity of the data, as well as the instructions to be executed in a SIMD computer may vary widely, from a single number to a list of strings and from an arthimetic operation to a complete program. In many applications, partial results obtained during the execution by the different processors may need to be exchanged among them. In this model, the communication among processors is achieved in one of two ways: through a **shared memory** (section 2.2.1) or through an **interconnection network** (section 2.2.2). Even though the SIMD model can be applied to a broader range of problems than the preceding models it still is limited to those which can be subdivided into identical subproblems. An example of the SIMD concept can be found on the vector unit of a CRAY, in which the same operation is to be performed on all components of a vector concurrently. Another example of a SIMD model is the GF-11 of IBM, a special purpose computer which has 576 processors (including 64 backup processors). Communications are done

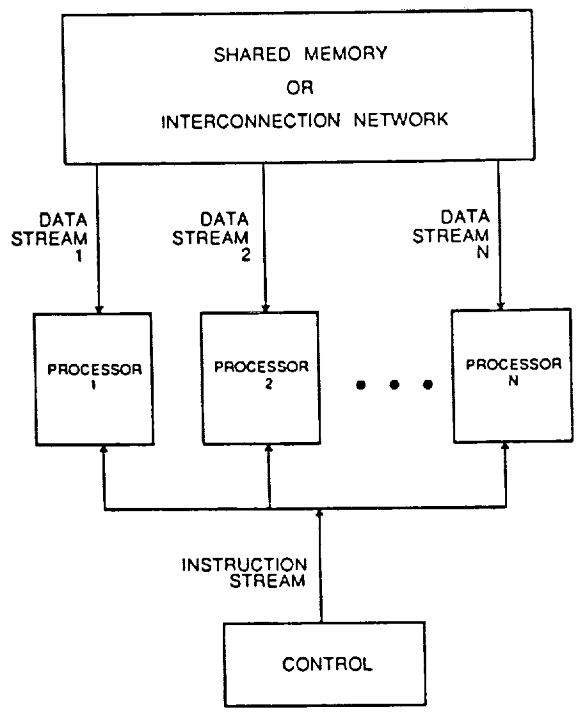


Figure 7. SIMD computer, [17].

via an interconnect network. Some additional flexibility is achieved by using local registers to control the behavior of each processor.

2.2.6. MIMD Model

The MIMD model concerns N processors, N streams of instruction and N streams of data. This architecture enables all problems to be solved in parallel, as long as parallelism exists in the application. Thus, it is a *general purpose architecture*. Figure 8 shows a schematic of a MIMD computer.

Each processor has its own control, arithmetic and logic units, as well as a local memory. Each control unit issues its own stream of instructions to its respective processor. All processors can execute independent programs concurrently. As in

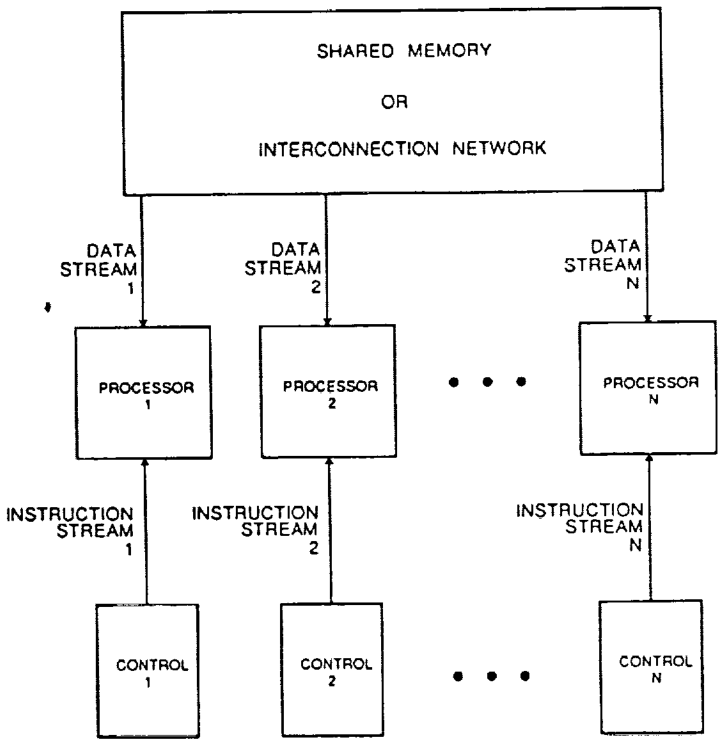


Figure 8. MIMD computer, [17].

the case of the SIMD model, the communication between processors is achieved through a **shared memory** (section 2.2.1) as well as through an **interconnection network** (section 2.2.2).

The MIMD class of computers represent the most general and powerful model of parallel computers. Here the problems to be solved are in general asynchronous, which means that all processors are executing independent tasks simultaneously. Initially, all processors are free and the parallel algorithm starts to be executed by an arbitrarily chosen processor, which creates the tasks to be performed. Once a task is created it is assigned to a *free* processor (a processor is freed when it completes the execution of a task). In case no free processor is available, the process stands by until a free processor becomes available. It is important to note that the idle time of processors depends very much on the way the problem is implemented, since as long as there is a free processor and a task to be performed no time is "wasted".

Most parallel computers currently in the market are MIMD models. Some examples of MIMD machines are: Alliant FX8 (with up to 8 processors), Denelcor HEP (with up to 50 processors) and Intel iPSC (with up to 128 processors).

2.3. Parallel Computers

In this section some recently developed parallel computers are discussed. Special attention is devoted to the description of the Alliant FX8 computer, where most of the simulations discussed in the sequel were conducted.

IBM/NYU Ultracomputer [8,18]

IBM/NYU Ultracomputer is a research project at New York University. The design of the Ultracomputer approximates a paracomputer (a multi processor in which multiple accesses to the same memory location are served in the same time required for a single access) by using message-switching network connected to a central shared memory. It is an example of a parallel computer whose memory is reconfigurable between global and local. The initial configuration, the RP3, has 512 processors, with the peak processing power of about 500 Mflops.

Connection Machine CM-2 [18]

The Connection Machine is designed by Thinking Machine Inc. It is a massively parallel architecture. It has 65,536 processors. It is an example of a SIMD machine and it has been extensively used for Artificial Intelligence applications.

It consists of two parts: a front end machine and a hypercube of 64k processors. Single data instructions are executed by the front end, while the CM executes large data items.

Intel iPSC [19]

The Intel iPSC is an example of MIMD machine. It is based on the CalTech's Cosmic Cubic Design. It was the first commercial computer using an interconnection network of the hypercube type. In present models, it offers up to 128 processors. The individual processors have up to 512Kb of memory, and the connections are provided by a high speed Ethernet. It also has an intermediate host machine (Intel 310), which serves as both the control processor and the user interface running UNIX.

Intel iPSC/2 [8]

The iPSC/2 is the current version of the iPSC, which represents significant advances over the original iPSC. Each node of the iPSC/2 is a functionally complete computer with its own processor, memory and communication facilities. The node processor on the iPSC/2 is a four-MIPS Intel 80386 processor. Each individual node can have up to 8 MBytes of memory, and the communication is done via a Direct-Connect routing module (DCM) on each node. The problem of passing message to distant processors, is solved efficiently by the DCM.

Sequent Balance [20]

The Balance system is a MIMD *multiprocessor* (another name for shared memory machines). The Balance CPUs are identical general purpose, 32-bit microprocessors. All processors share a single pool of memory. Also, all processors, memory modules, and I/O controllers plug into a single high-speed bus. The scheduling for the processors is done automatically by themselves, to ensure that all processors are kept busy as long as there are executable processes available. The Balance systems are available in two models, the Balance 8000, with 2 to 12 processors, and the Balance 21000, with 4 to 30 processors. Both Balance models can be configured with 4 to 28 Mbytes of memory and provide 16 Mbytes of virtual address space per processor.

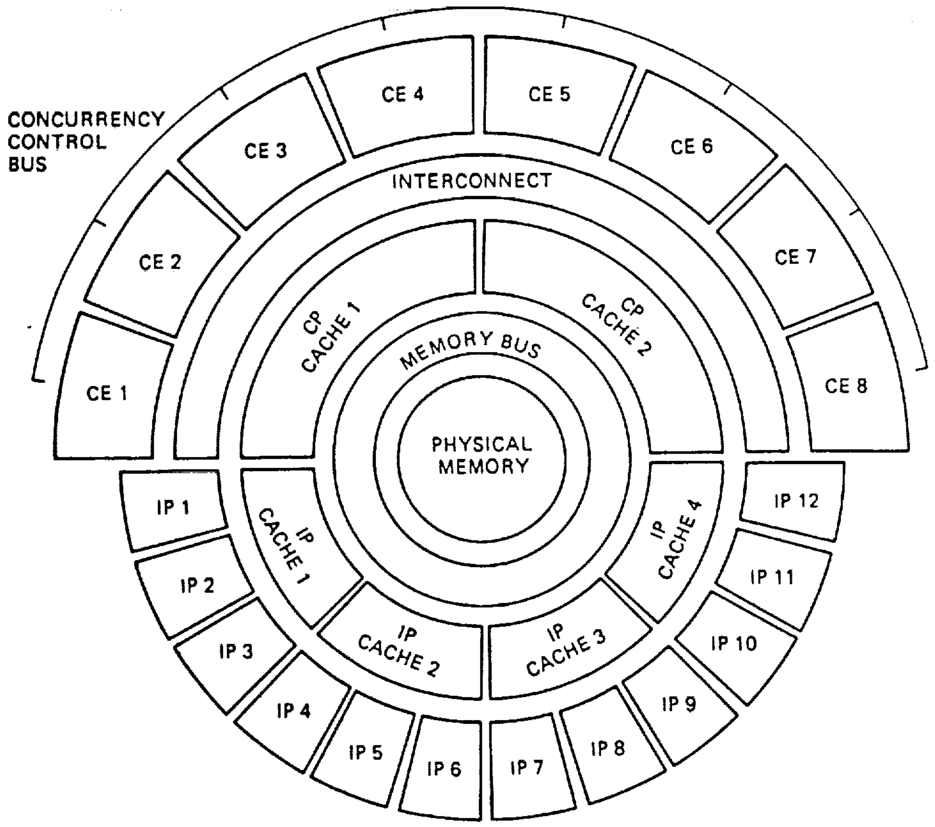
Alliant FX8 [8,21,22]

The Alliant FX8 is an MIMD machine with shared memory. Its basic approach to parallel processing is to use hardware for the scheduling and synchronization of the multiple processors and to develop compilers that automatically break up programs into those parts that can be vectorized and those that must run in scalar form. The Alliant architecture is based on two distinct, but interconnected, resource classes:

- The *interactive processors (IP's)* comprise an expandable pool of computers that execute interactive user jobs and the operating system in parallel with each other and with the computational complex (the second resource class).

- The *computational complex* introduces the Alliant concurrency, which groups up to 8 processors, called *computational elements (CEs)*, in a complex. Each CE is a 4450 - KWhetstone (32-bit) general purpose microprogrammed computer with an integrated vector instruction set. Each CE delivers 11.8 MFLOPs peak performance (32-bit).

Concurrency initiation, synchronization, and suspension are accomplished by the *Concurrency Control Unit* (CCU) in each CE and an interconnecting *Concurrency Control Bus.* The Concurrency Control Bus provides a high-speed communication path between CCUs that is independent of program data and instructions paths. The (CCU) is an 8000-gate CMOS gate array that connects the CEs of a complex. The CCU controls Alliant concurrency and assures high parallel processing efficiency. The CCU interfaces with the instruction unit of a CE and up to seven other CCUs to control up to eight CEs running concurrently. Because it is the hardware that performs the scheduling and synchronization of multiple CEs in the computational complex, the performance speed-up delivered to a single application approaches the number of CEs installed.



(with permission of Alliant Computer Systems)

Figure 9. Alliant architecture, [8].

The Concentrix operating system is an implementation of the Berkeley 4.2

UNIX operating system. Concentrix supports parallel processing without programmer or operator intervention. The system manages two types of processes and dynamically schedules jobs on available processors as long as work remains. Compute-intensive jobs take priority on the computational complex; interactive user jobs, input/output, and other operating system activities are scheduled for any available IP or otherwise idle computational complex. Figure 9 shows a sketch of the Alliant architecture.

The software optimization is done at compilation time by turning on/off the optimization options. The optimizations available are: concurrency, vectorization and global optimization (done by the machine to avoid "unecessary" operations that might exist in the code).

References

1. J. J. Dongarra, and E. L. Lusk, 'Advanced Computing Research Facility and Algorithm Design for Different Computers,' *Parallel Computations and their Impact on Mechanics*, ed. by A. K. Noor, ASME, New York, 1987.

2. A. K. Noor, 'Preface', *Parallel Computations and their Impact on Mechanics*, ed. by A. K. Noor, ASME, New York, 1987.

3. C. Norrie, 'Supercomputers for Superproblems: An Architectural Introduction,' *Computer*, Vol. 17, No. 3, 1984, pp. 62-79.

4. D. D. Wilthmarth, and R. C. Young, 'Structural Analysis on the FPS-164,' *Computers and Structures*, Vol. 20, No. 1-3, 1985, pp. 77-83.

5. J. A. Swanson, G. R. Cameron, and J. C. Haberland, 'Adapting the Ansys Finite-Element Analysis Program to an Attached Processor,' *Computer*, Vol. 16, No. 6, 1983, pp. 85-91.

6. A. Noor, J. Lambiotte, 'Finite Element Dynamic Analysis on the CDC STAR-100 Computer,' *Computer and Structures*, Vol. 10, No. 1-2, 1979, pp. 7-19.

7. A. K. Noor, and J. M. Peters, 'Element Stiffness Computation on CDC Cyber 205 Computer,' *Communications in Applied Numerical Methods*, Vol. 2, No. 3, 1986, pp. 317-328.

8. A. L. DeCegama, 'Parallel Processing Architectures and VLSI Hardware,' Volume I, *Prentice Hall*, 1989.

9. N. R. Lincoln, 'Supercomputers = Colossal Computations + Enormous Expectations + Renowned Risk,' *Computer*, Vol. 16, No. 5, 1983, pp. 38-47.

10. L. M. Adams, and T. W. Crockett, 'Modelling Algorithm Execution Time on Processor Arrays,' *Computer*, Vol. 17, No. 7, 1984, pp. 38-43.

11. R. G. Melhen, 'On the Design of a Pipelined/Systolic Finite Element System,' *Computers and Structures*, Vol. 20, No. 1-3, 1985a, pp. 67-75.

12. H. T. Kung, 'Why Systolic Architectures?,' *Computer*, Vol. 15, No. 1, 1982, pp. 37-46.

13. K. H. Law, 'Systolic Arrays for Finite Element Analysis,' *Computers and Structures*, Vol. 20, No. 1-3, 1985, pp. 55-65.

14. D. Gannon, D. Atapattu, H. L. Mann, and B. Shei, 'A Software Tool for Building Supercomputer Application,' *Parallel Computations and their Impact on Mechanics*, ed. by A. K. Noor, ASME, New York, 1987.

15. J. Boyle, R. Butler, T. Disz, B. Glickfeld, B., E. Lusk, R. Overbeek, J. Patterson, and R. Stevens, 'Portable Programs for Parallel Processors,' *Holt, Rinehart & Wiston*, 1987.

16. N. Sarigul, M. Jin, R. Kolar, and H. A. Kamel, 'Design of Array Processor Software for Nonlinear Structural Analysis,' *Computers and Structures*, Vol. 20, No. 6, 1985, pp. 963-974.

17. S. G. Akl, 'The Design and Analysis of Parallel Algorithms', *Wadsworth Pub. Co.*, 1989.

18. O. A. McBryan, 'State-of-art in Highly Parallel Computer Systems,' *Parallel Computations and their Impact on Mechanics*, ed. by A. K. Noor, ASME, New York, 1987.

19. iPSC System Overview, *Intel Corporation*, Jan 1986.

20. BALANCE Guide to Parallel Programming, *Sequent Company*, June 1986.

21. ALLIANT FX/Series, Product Summary, *Alliant Computer Systems Corporation*, June 1985

22. FX/Fortran Programmer's manual, *Alliant Computer System Corporation*, May 1985.

CHAPTER III

Survey of Parallel Algorithms

Akl [1] defines a parallel algorithm as a solution method for a given problem destined to be performed on a parallel computer. In [2] Noor defines *vector computations* as simultaneous processing of several independent data streams on a single processor, and *parallel computations* as simultaneous processing of independent streams of data on multiple processors. The main difference between these two modes of computations is their CPU performance compared with the scalar mode. While vector processing can significantly reduce CPU time, parallel processing increases CPU time (due to overhead), but reduces the wall-clock time.

In computational mechanics a prominent role is played by the finite element method. Several avenues for parallelizing this method have been proposed. In [3] an array of systolic processors for doing finite element calculations is presented. Systolic arrays are a network of very simple processors, which operate in parallel and are usually designed as special purpose systems (see Section 2.1). In this systolic array there is one processor allocated for each node of the finite element mesh. Each processor maintains one row of the coefficient matrix either in element or global form. Connectivity and data flow between processors is dictated by the connectivity of the nodes in the finite element mesh and can be generated as the element connectivity is defined.

In a more abstract sense, different approaches to the parallelization of the finite element method have been considered [4]:

- *Subdomain splitting.* It is based in the "divide and conquer" technique. Here, the task to be performed is divided into subtasks that are either independent or loosely coupled (to reduce the extent of communication among processors). The idea is one of domain (or spatial) decomposition, i. e., the domain is divided into regions (that can even overlap). The problem is then decomposed into the solution of boundary value problems in the subdomains. Since the data on the interface of the subdomains is not known an iterative solution procedure is necessary. In [5] Rodrigue considers two methods of decomposition: one in which the decomposition is made without regard to the partial differential equations being solved and the another in which the decomposition is made according to the heuristics of the solution of the partial differential equation.

- *Substructuring.* This concept is closely related to that of subdomain splitting. It can also be identified at the algebraic level with partitioning of the system matrices. To achieve a perfectly balanced workload distribution is generally an intractable combinatorial optimization problem. In [6] Flower et al. propose a satisfactory approximate solution for this problem by means of an analogy to the phenomenon of annealing in solids.

- *Operator splitting.* Splitting provides a generalization of substructuring. Splitting strategies can be developed in a variety ways. One example is the method of alternating directions [7], whereby a multidimensional problem is reduced into a series of one-dimensional problems. Another example is provided by the method of fractional steps [8].

- *Element-by-element strategies.* In finite element calculations, global arrays are assembled from element contributions. This modular characteristic of the method can be taken as a basis for the formulation of splitting schemes in which the elements in the mesh are treated sequentially [9]. Although the method considerably reduces the storage requirements with respect to implicit algorithms, its inherently sequential nature renders it of limited value for parallel computing.

In large scale nonlinear analyses, the most costly phase of the computations is the repeated solution of systems linear algebraic equations. Considerable research is presently being devoted to the development of parallel equation solvers. In Section 3.1 some direct and iterative techniques that have been exploited are presented. For transient problems, several techniques have been proposed for the integration of the equations of evolution which broadly fall into two categories: explicit and implicit methods. In Section 3.2 the tradeoff between explicit or implicit schemes is discussed.

3.1. Equation Solvers

In many finite element applications, the solution phase is responsible for a large fraction of the execution times. Whereas the element computations can be easily performed in parallel, since they constitute independent operations, equation solvers are not trivially parallelizable. The systems of equations arising in the displacement method are of the form

$$\mathbf{K}\mathbf{u} = \mathbf{f} \tag{3.1}$$

where $\mathbf{K}$ is the stiffness matrix, $\mathbf{u}$ is the vector of nodal displacements, and $\mathbf{f}$ is the vector of effective nodal forces. In nonlinear applications solved by means of the Newton-Raphson method, $\mathbf{K}$ is the tangent stiffness matrix, $\mathbf{u}$ is the vector of increment of nodal displacements, and $\mathbf{f}$ is the vector of residual forces.

Methods to solve (3.1) based on the direct factorization of the matrix $\mathbf{K}$ are called direct methods. Iterative solvers constitute the other main solution strategy and have been applied since the early 60s [10]. A typical iterative method involves the inital selection of an approximation $\mathbf{u}^{(0)}$ to $\mathbf{u}$, and the determination of a sequence $\mathbf{u}^{(1)}, \mathbf{u}^{(2)}, \mathbf{u}^{(3)}, \ldots$, such that the $\lim_{i \to \infty} \mathbf{u}^{(i)} = \mathbf{u}$. Iterative solvers typically require considerable less storage than direct solvers. In terms of performance, direct methods of solution are generally faster than iterative methods and have been preferred for use on sequential machines. However, because of the parallelism inherent in iterative solvers, since the advent of parallel and vector computers methods like preconditioned conjugate gradients, successive overrelaxation (SOR), Gauss-Seidel, and point Jacobi's have elicited renewed attention.

3.1.1. Iterative Methods

Consider the following system of linear equations

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{3.2}$$

where $\mathbf{A}$ is an nxn coefficient matrix, $\mathbf{x}$ is the solution vector with n components and $\mathbf{b}$ is a given column vector also with n components. Widely used iterative methods [11] to solve a system like (3.2) are: Point Jacobi, Gauss Seidel and Successive Overrelaxation methods. The solution vector exists and is unique if and only if $\mathbf{A}$ is nonsingular, i. e., $\mathbf{A}^{-1}$ exists, since

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b} \tag{3.3}$$

From here on it is assumed that the matrix $\mathbf{A}$ is nonsingular and furthermore that its diagonal terms, i. e., a_{ii} are all nonzero. This matrix can be decomposed as

$$\mathbf{A} = \mathbf{D} - \mathbf{L} - \mathbf{U} \tag{3.4}$$

where, $\mathbf{D}, \mathbf{L}$ and $\mathbf{U}$ are respectively diagonal, lower triangular and upper triangular matrices. Their respective elements are:

$$\begin{aligned} &d_{ii} = a_{ii}, \;\; d_{ij} = 0 \text{ for } i \neq j \\ &l_{ij} = -a_{ij} \text{ for } i > j, \text{ and } l_{ij} = 0 \text{ for } i \leq j \\ &u_{ij} = -a_{ij} \text{ for } i < j, \text{ and } u_{ij} = 0 \text{ for } i \geq j \end{aligned} \tag{3.5}$$

Using (3.4), equation (3.2) can be rewritten as

$$\mathbf{Dx} = (\mathbf{L} + \mathbf{U})\mathbf{x} + \mathbf{b}. \tag{3.6}$$

The **point Jacobi method** is defined by the recurrence relation

$$\mathbf{D}\mathbf{x}^{(m+1)} = (\mathbf{L} + \mathbf{U})\mathbf{x}^{(m)} + \mathbf{b}, \;\; m \geq 0. \tag{3.7}$$

Since the elements in the diagonal of $\mathbf{A}$ are nonzero, $\mathbf{D}$ is nonsingular, and (3.7) can be rewritten as

$$\mathbf{x}^{(m+1)} = \mathbf{D}^{-1}(\mathbf{L} + \mathbf{U})\mathbf{x}^{(m)} + \mathbf{D}^{-1}\mathbf{b}, \;\; m \geq 0. \tag{3.8}$$

The matrix $\mathbf{J} = \mathbf{D}^{-1}(\mathbf{L} + \mathbf{U})$ is called point Jacobi matrix associated with the matrix $\mathbf{A}$.

One of the disadvantages of this method is that all the components of $\mathbf{x}^{(m)}$ need to be saved while computing $\mathbf{x}^{(m+1)}$. A way of avoiding this shortcoming is by taking advantage of how the matrices $\mathbf{D}, \mathbf{L}$ and $\mathbf{U}$ are formed, i. e,

$$a_{ii}x^{(m+1)} = -\sum_{j=1}^{i-1} a_{ij}x_j^{(m+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(m)} + b_i, \;\; 1 \leq i \leq n, \;\; m \geq 0. \tag{3.9}$$

In matrix notation (3.9) can be translated into,

$$(\mathbf{D} - \mathbf{L})\mathbf{x}^{(m+1)} = \mathbf{U}\mathbf{x}^{(m)} + \mathbf{b}, \tag{3.10}$$

Since the lower triangular matrix $(\mathbf{D}-\mathbf{L})$ is nonsigular (3.10) can finally be rewritten as

$$\mathbf{x}^{(m+1)} = (\mathbf{D}-\mathbf{L})^{-1}\mathbf{U}\mathbf{x}^{(m)} + (\mathbf{D}-\mathbf{L})^{-1}\mathbf{b}, \tag{3.11}$$

This iterative method is the point **Gauss-Seidel method** and the point Gauss-Seidel matrix associated with matrix $\mathbf{A}$ is defined as $\mathbf{G} = (\mathbf{D}-\mathbf{L})^{-1}\mathbf{U}$.

An alternative iterative method is the **SOR (Successive Overrelaxation method)**. To derive it, define first the auxiliary vector iterates $\tilde{\mathbf{x}}^{(m)}$

$$\mathbf{D}\tilde{\mathbf{x}}^{(m+1)} = -\mathbf{L}\mathbf{x}^{(m+1)} - \mathbf{U}\mathbf{x}^{(m)} + \mathbf{b}. \tag{3.12}$$

Then, the method is defined as

$$\mathbf{x}^{(m+1)} = \mathbf{x}^{(m)} + w[\tilde{\mathbf{x}}^{(m+1)} - \mathbf{x}^{(m)}] = (1-w)\mathbf{x}^{(m)} + w\tilde{\mathbf{x}}^{(m+1)}, \tag{3.13}$$

where w is the relaxation factor. From (3.13) one can verify that $\mathbf{x}^{(m+1)}$ is a weighted mean of $\mathbf{x}^{(m)}$ and $\tilde{\mathbf{x}}^{(m+1)}$. When $w > 1$ the weight is an overrelaxation weight, otherwise it is an underrelaxation weight. Putting (3.12) and (3.13) together the following relation is derived:

$$(\mathbf{D}-w\mathbf{L})\mathbf{x}^{(m+1)} = [(1-w)\mathbf{D} + w\mathbf{U}]\mathbf{x}^{(m)} + w\mathbf{b}. \tag{3.14}$$

Note that $\mathbf{D}-w\mathbf{L}$ is nonsingular for any w, and thus, the final form of the SOR method can be written as

$$\mathbf{x}^{(m+1)} = (\mathbf{D}-w\mathbf{L})^{-1}[(1-w)\mathbf{D} + w\mathbf{U}]\mathbf{x}^{(m)} + w(\mathbf{D}-w\mathbf{L})^{-1}\mathbf{b}. \tag{3.15}$$

The matrix $\mathbf{R} = (\mathbf{D}-w\mathbf{L})^{-1}[(1-w)\mathbf{D} + w\mathbf{U}]$ is called the point successive relaxation matrix. Equation (3.15) can be rewritten in the form

$$\mathbf{x}^{(m+1)} = \mathbf{x}^{(m)} + w\mathbf{D}^{-1}(\mathbf{b} - \mathbf{L}\mathbf{x}^{(m+1)} - \mathbf{U}\mathbf{x}^{(m)} - \mathbf{D}\mathbf{x}^{(m)}) \tag{3.16}$$

A more elaborate iterative procedure is the **preconditioned conjugate gradient (PCG)** method. The PCG is an extension of the conjugate gradient method, which is an extension of the method of the steepest descent. The latter is based in the following [12]:

Let f have a continuous first partial derivative. The gradient vector of f is $\mathbf{g}(\mathbf{x}) = \nabla \mathbf{f}(\mathbf{x})^{\mathbf{T}}$, or simply $\mathbf{g}_k$. The method of steepest descent, for minimizing a function, is defined by the iterative algorithm

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k \mathbf{g}_k, \tag{3.17}$$

where α_k is a nonegative scalar minimizing $f(\mathbf{x}_k - \alpha \mathbf{g}_k)$. That is, from the point $\mathbf{x}_k$ it searches along the direction of the negative gradient $-\mathbf{g}_k$ to a minimum point on this line; this minimum point is taken to be $\mathbf{x}_{k+1}$.

The first step in the conjugate gradient method is identical to a step descent method; each succeeding step moves in a direction that is linear combination of the current gradient and the preceding direction vector [12]:

Given an approximation solution $\mathbf{x}_0$ to $\mathbf{x}$, define

$$\mathbf{d}_0 = -\mathbf{g}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0, \tag{3.18}$$

and at each iteration k define the method as

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{d}_k \tag{3.19}$$

$$\alpha_k = \frac{-\mathbf{g}_k^T \mathbf{d}_k}{\mathbf{d}_k^T \mathbf{A} \mathbf{d}_k} \tag{3.20}$$

$$\mathbf{d}_{k+1} = -\mathbf{g}_{k+1} + \beta_k \mathbf{d}_k \tag{3.21}$$

$$\beta_k = \frac{\mathbf{g}_{k+1}^T \mathbf{A} \mathbf{d}_k}{\mathbf{d}_k^T \mathbf{A} \mathbf{d}_k} \tag{3.22}$$

The conjugate gradient method can be improved by way of preconditioning. This technique consists of splitting the coefficient matrix in a form: $\mathbf{A} = \mathbf{M} - \mathbf{N}$, such that the system: $\mathbf{Mx} = \mathbf{b}$ is computationally inexpensive compared to the original system. When $\mathbf{M}^{-1} = \mathbf{A}^{-1}$ the iteration converges in one step. The closer $\mathbf{M}^{-1}$ is to $\mathbf{A}^{-1}$, the faster the convergence of the method. A discussion of preconditioning strategies may be found in [13].

The implementation of the aforementioned iterative methods in parallel computers has received much attention. Adams in [14] shows how to reorder the computations in the SOR algorithm to maintain the same asymptotic rate of convergence as the row-wise ordering and to obtain parallelism at different levels. Two major problems are introduced when this reordering is performed: it is unlikely that an ordering can be developed that is best for every new parallel machine, and also reordering computations can change the mathematical properties of the algorithm.

Discretizing an elliptical partial differential equation on a regular domain of 9-point stencil as in Figure 1 gives rise to a system of linear equations of the type (3.2).

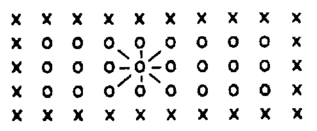


Figure 1. Discretized domain, [14]

For simplicity, one unknown per node is considered here. An "ordering" means that the nodes must be updated sequentially. The first step of this method is to order the unknowns at the nodes to indicate which nodes must be updated before the others. Using the multi-color SOR method [15] so that the nodes of same color are updated simultaneously the desired parallelism is created. There are several possible multi-color orderings, with probably differing rates of convergence. To aid the choice of an ordering, one imposes that the new ordering must have the same rate of convergence as the row-wise ordering of the domain. The row-wise ordering is shown in Figure 2, which indicates that a node may not be updated at iteration $k+1$ until all the nodes in the stencil to the left and below are updated.

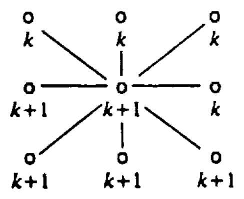


Figure 2. Stencil rule for row-wise ordering, [14]

A systematic procedure to find the 4-color ordering for this stencil with the same rate of convergence as that of the row-wise ordering is given in [16]. The basic idea is to apply the stencil rule given in Figure 2 to the grid in Figure 1, but allowing the nodes to be updated on subsequent iterations as soon as the appropriate data is available. Figure 3 shows the sequence of update times for each node. The three sets are each in a different iteration of the SOR method. In a parallel computer the update is performed by color, i. e., first all R nodes are updated, followed by the B nodes, the G nodes and finally the O nodes.

R 5,9	B 6,10	G 7,11	O 8,12	R 9	B 10	G 11	O 12
G 3,7,11	O 4,8,12	R 5,9	B 6,10	G 7,11	O 8,12	R 9	B 10
R 1,5,9	B 2,6,10	G 3,7,11	O 4,8,12	R 5,9	B 6,10	G 7,11	O 8,12

Figure 3. R/B/G/O coloring and ordering, [14]

In [17] Doi et al propose parallel processing and pre-processing algorithms for the solution of partial differential equations by the finite element method. In the solution phase a parallel SOR method is proposed, given by

$$x_i^{(m+1)} = x_i^{(m)} + wD_{ii}^{-1}[\sum_{j=i-1}^{i+1} L_{ij}^{(m+1)} + U_{ij}x_j^{(m)} + D_{ij}x_j^{(m-1)}], \tag{3.23}$$

where $i = 1, ..., n$ and $k = 1, 2, ...$ The parallel processing system considered possesses the following attributes:

(1) It consists of n slave processors SP_i ($i = 1, ...n$) with identical performances and a master processor MP, which controls the slave processors.

(2) The n SP's constitute an one dimensional array, i. e., each processor SP_i can access the shared memory SM_i and SM_{i-1}, which are the shared memories of processor SP_i and SP_{i-1} respectively.

(3) Each SP_i has a local memory LM_i that allows it to run its own program.

(4) MP can access any of the SM's.

In the calculation of (3.23), SP_i uses x_{i-1}, x_i and x_{i+1}. In order for SP_i to be able to take in all the data needed in the calculation directly from SM_{i-1}, SM_i and LM_i, without any data transfer, x_i is assigned to both SM_{i-1} and SM_i. The convergence decision for iteration m is made by MP while the SP's perform iteration $m+1$.

An algorithm for solving (3.2) using iterative methods that requires one single matrix-vector multiplication per iteration is presented by Melhem in [18]. This product is performed using the unassembled elemental arrays, therefore eliminating the need for the irregular assembly stage. More specifically, the product of the matrix **A** with any vector **p** can be done as

$$\mathbf{A}\ \mathbf{p} = \sum_{e=1}^{n} \mathbf{M}^{eT}\ \bar{\mathbf{A}}^{e}\ \mathbf{M}^{e}\ \mathbf{p} = \sum_{e=1}^{n} \mathbf{M}^{eT}\ \bar{\mathbf{A}}^{e}\ \mathbf{p}^{e}, \qquad (3.24)$$

where $\mathbf{M}^e$ is a Boolean matrix for each element, such that $\mathbf{M}^e_{ij} = 1$ if the global numbering of node i in element e is equal to j. The partial products $\bar{\mathbf{A}}^e\ \mathbf{p}^e$ for $e = 1, 2, ..., n$ may be pipelined at the same rate at which the arrays $\bar{\mathbf{A}}^e$ are generated.

Seager in [13] studies a standard PCG algorithm for the solution of symmetric linear systems in the context of multi-processing. The expensive operation, as noted above, is the matrix-vector product. This computation may be decomposed into concurrent tasks, each responsible for calculating a different part of the vectors **x**, **d** and **g**. In this way each processor performs part of the matrix-vector multiply. These vectors are partitioned so that vectorization is not detrimentally affected.

In [19] Nour-Omid et al. propose a method based on partitioning the mesh into substructures. The nodes of each substructure are subdivided into interior nodes and interface nodes. The latter are the nodes shared by more than one substructure. While the interior nodes are eliminated by means of direct factorization, (section 3.1.2), the interface nodes are solved for by means of a preconditioned conjugate gradient iteration. The choice of a direct method for solving the interior nodes can be justified by the fact that, for small systems, direct methods are more efficient than iterative methods. The resulting system of equations resulting after the elimination of the interior nodes is of the form

$$\Big[\sum_{i=1}^{p} \mathbf{L}^{(i)T}\, \bar{\mathbf{K}}_s^{(i)}\, \mathbf{L}^{(i)}\Big]\, \mathbf{u}_s = \sum_{i=1}^{p} \mathbf{L}^{(i)T}\, \bar{\mathbf{f}}_s^{(i)}, \tag{3.25}$$

where, $\mathbf{L}^{(i)}$ is the localization operator that maps the nodal displacements within a substructure ($\mathbf{u}_s^{(i)}$) to the global nodal displacement ($\mathbf{u}_s$), i.e., $\mathbf{u}_s^{(i)} = \mathbf{L}^{(i)}\, \mathbf{u}_s$, p is the number of substructures, $\bar{\mathbf{K}}_s^{(i)}$ is the reduced stiffness matrix and finally $\bar{\mathbf{f}}_s^{(i)}$ is the reduced force vector. A PCG method is used to solve (3.25). At each iteration of the PCG algorithm the product of the matrix in (3.25) and a vector $\mathbf{d}$ is evaluated (equations (3.20) and (3.22)). Using the definition of the coefficient matrix such product can be written as

$$\mathbf{h}_k = \sum_{i=1}^{p} \mathbf{L}^{(i)T}\, \bar{\mathbf{K}}_s^{(i)}\, \mathbf{L}^{(i)}\, \mathbf{d}_k \tag{3.26}$$

The cost of computing this product dominates the total cost of the PCG method. Concurrency can be achieved here by computing each term in this sum separately. First, $\mathbf{d}$ is localized to each processor by means of the localization operator ($\mathbf{L}^{(i)}$). Then, the product of $\bar{\mathbf{K}}_s^{(i)}$ and the localization of $\mathbf{d}$ to the ith substructure is computed. Although computing the product by this means may be two or three times slower than a direct calculation, the gains afforded by concurrency tend to dominate provided that the number of processors is high enough.

In [10] Biffle adapts the nonlinear conjugate gradient algorithm to concurrent vector processing computers, while striving to preserve the vector processing speed of the algorithm. The efficiency of the conjugate gradient iteration depends critically on the cost of calculating the residual. The method used to perform the residual calculation is highly vectorizable. To accomplish multitasking while preserving vectorization, the following data structure is used. The size of each block of elements is increased to 1024 elements. When a processor becomes available it is given 64 elements for which to calculate their residual forces. Giving a processor 64 elements allows the processor to run in vector speed. If another processor becomes available, then the next available block of 64 elements is processed. When all the residual forces are calculated for the 1024 elements, then one of the processors performs the accumulation of the element residual forces into a residual force vector.

3.1.2. Direct Methods

The most basic direct method to solve symmetric systems of n linear equations is the Gaussian elimination followed by backsubstitution. The Gaussian elimination method consists in reducing (factorizing) the given system of equation to one in which the coefficients matrix is an upper triangular matrix. Once this simplified system of equations is obtained the solution can be found very straightforwardly. First the nth component of the solution vector is computer, followed by the $(n-1)$th component and so on, until all of them are computed.

Another widely known and used method for factorizing the coefficient matrix is the Cholesky method, which is a symmetric variant of the Gaussian elimination tailored to symmetric positive definite matrices [20]. Considering the system of equations given in (3.2). Applying Cholesky's method to $\mathbf{A}$ yields the triangular factorization

$$\mathbf{A} = \mathbf{L}\mathbf{L}^T, \tag{3.27}$$

where $\mathbf{L}$ is a lower triangular matrix with positive diagonal terms. (3.2) can then be rewritten as

$$\mathbf{L}\mathbf{L}^T\mathbf{x} = \mathbf{b}, \tag{3.28}$$

and substituting $\mathbf{y} = \mathbf{L}^T\mathbf{x}$, one can obtain $\mathbf{x}$, by solving

$$\mathbf{L}\mathbf{y} = \mathbf{b} \quad \text{and} \quad \mathbf{L}^T\mathbf{x} = \mathbf{y}. \tag{3.29}$$

It should be pointed out that to solve the first system of equations in (3.29), one should use forward substitution, i. e., the 1st component of $\mathbf{y}$ is computed first, followed by the second and so on until all the n components are known.

Another way of factorizing matrix $\mathbf{A}$ is

$$\mathbf{A} = \mathbf{L}\mathbf{D}\mathbf{L}^T \tag{3.30}$$

where $\mathbf{L}$ is a lower triangular matrix and $\mathbf{D}$ is a diagonal matrix. In this case, substituting $\mathbf{y} = \mathbf{D}\mathbf{L}^T\mathbf{x}$, the solution vector $\mathbf{x}$ is obtained by soving

$$\mathbf{L}\mathbf{y} = \mathbf{b} \quad \text{and} \quad \mathbf{D}\mathbf{L}^T\mathbf{x} = \mathbf{y}. \tag{3.31}$$

A very used factorization for the direct solution of systems of equations is the $\mathbf{LU}$ decomposition, i. e.,

$$\mathbf{A} = \mathbf{L}\mathbf{U}, \tag{3.32}$$

where $\mathbf{L}$ and $\mathbf{U}$ are lower and upper triangular matrices, respectively. Following the same idea as the methods above, one substitutes $\mathbf{y} = \mathbf{U}\mathbf{x}$, and obtain the solution by

$$\mathbf{L}\mathbf{y} = \mathbf{b} \quad \text{and} \quad \mathbf{U}\mathbf{x} = \mathbf{y}. \tag{3.33}$$

In [18] Melhem presents a parallel direct solver for the system of equations (3.2). The factorization is done using $\mathbf{LU}$ decomposition. A frontal technique to allow both the asssembly and factorization stages to be performed in parallel and also to minimize the storage requirement in the assembly phase is developed. In most of the parallel schemes for direct solution of (3.2), the rows of $\mathbf{A}$ have to be processed in sequential order. This restriction is satisfied by assigning appropriate global labels to the nodes. The interaction between the assembly and the factorization phases allows automatically the knowledge of when a row is ready to be passed to the factorization phase, eliminating the preprocessing step. It is also shown that the bandwidth of the resulting matrix (after the numbering process) is comparable with the best known algorithm for minimizing the bandwidth.

Among the features presented by Law [21] to parallelize the finite element method, is that of performing the solution phase of the method concurrently. There, a systolic array (see chapter II) is developed for performing the **LU** decomposition.

Farhat [22] develops a computer program architecture for the solution of finite element systems using concurrent processing. The basic approach involves the automatic creation of substructures. The algorithm, then, consists of solving each substructure problem independently using $\mathbf{L}^T\mathbf{DL}$ factorization and the solution for the equations corresponding to the interface nodes (nodes that belong to more than one substructure) is obtained by means of the Gaussian elimination method, which possesses inherent parallelism.

In [23] Johnsson presents three classes of concurrent elimination algorithms for the solution of banded systems of equations. One class exploits the independence of a single variable from the system of equations, another the independence of the data sets for the elimination of different variables and the third is a combination of the other two.

The tradeoffs between parallelization and vectorization are assessed in [24]. For the **LU** factorization a method is presented that takes advantage of both parallel and vector capabilities of the parallel computer Alliant FX8 (see chapter II for details). The classical **LU** decomposition consists mainly of dot products. The proposed algorithm decomposes the coefficients matrix **A** into its lower and upper triangular components, which can be written as

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix} = \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{L}_{21} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{U}_{11} & \mathbf{U}_{12} \\ \mathbf{0} & \mathbf{B} \end{bmatrix}, \tag{3.34}$$

where all the submatrices are kxk matrices. The first step of the algorithm is

$$\mathbf{A}_{11} \leftarrow \mathbf{A}_{11}^{-1}, \quad \mathbf{L}_{21} = \mathbf{A}_{21}\mathbf{A}_{11}, \quad \mathbf{B} = \mathbf{A}_{22} - \mathbf{L}_{21}\mathbf{A}_{12}. \tag{3.35}$$

The above operations are then performed recursively on the smaller matrix **B**. This block **LU** algorithm consists mainly of matrix-matrix operations. To invert the kxk blocks the classical **LU** factorization of $\mathbf{A}_{11}$ is used. The computation of the inverse of the original matrix is thus avoided. Unfortunately, this block **LU** is more expensive by a factor of $(1 + 2k^2/n^2)$ than the classical **LU** factorization.

3.2. Time Stepping Algorithms

The equations of motion governing linear structural systems are of the form

$$\mathbf{M}\ddot{\mathbf{d}} + \mathbf{C}\dot{\mathbf{d}} + \mathbf{K}\mathbf{d} = \mathbf{f}, \tag{3.36}$$

where **K** is the stiffness matrix, **M** is the mass matrix, **C** is the damping matrix, **f** is the vector of applied discretized loads, and $\ddot{\mathbf{d}}$ and $\mathbf{d}$ are the vectors of nodal accelerations and displacements, respectively. A superimposed dot represents differentiation with respect to time. If an initial condition is given, equation (3.36) can be integrated to produce the time history of the response of the structure. Integration methods are usually categorized into two groups: *explicit* and *implicit*. Explicit schemes use initial data only to update the solution and do not require the solution of global systems of equations, in contrast to implicit methods. The advantages and disadvantages of both classes of algorithms have been summarized by Belytschko as follows:

Advantages(+) and disadvantages(-) of explicit schemes:

+ Simplicity.

+ Accuracy for large systems is assured if the time step (Δt) is stable.

+ No global stiffness matrix needs to be formed or factorized. Saves storage.

– Conditionally stable.

Advantages(+) and disadvantages(-) of implicit schemes:

+ Unconditionally stable.

– Complex algorithm with low reliability in nonlinear situations.

– Accuracy can deteriorate in semi-implicit algorithms.

– Newton form has large storage requirements.

Implicit/explicit partition methods [25] are an attempt to combine the best attributes of both classes of algorithms. In nodally based methods [25], the mesh is partioned in three sets: explicit, implicit and interface. In element based methods, the elements are segregated into two sets: implicit and explicit [26]. At each time step, the explicit subset is integrated first. The results from this step are then used as boundary conditions for the integration of the implicit subset.

The most widely used direct time-stepping methods are the Newmark family, which is defined:

$$\mathbf{M}\mathbf{a}_{n+1} \; + \; \mathbf{C}\mathbf{v}_{n+1} \; + \; \mathbf{K}\mathbf{d}_{n+1} \; = \; \mathbf{f}_{n+1} \tag{3.37}$$

$$\mathbf{d}_{n+1} \; = \; \mathbf{d}_n \; + \; \Delta t \mathbf{v}_n \; + \; \frac{\Delta t^2}{2}[(1-2\beta)\mathbf{a}_n \; + \; 2\beta\mathbf{a}_{n+1}] \tag{3.38}$$

$$\mathbf{v}_{n+1} \; = \; \mathbf{v}_n \; + \; \Delta t[(1-\gamma)\mathbf{a}_n \; + \; \gamma\mathbf{a}_{n+1}] \tag{3.39}$$

where $\mathbf{d}_n, \mathbf{v}_n$ and $\mathbf{a}_n$ are the approximations of $\mathbf{d}(t_n)$, $\dot{\mathbf{d}}(t_n)$, and $\ddot{\mathbf{d}}(t_n)$, respectively. The parameters β and γ determine both the accuracy and the stability of the specific algorithm being considered. Equations (3.37), (3.38) and (3.39) can be viewed as a system of equations in the unknowns $\mathbf{d}_{n+1}, \mathbf{v}_{n+1}$, and $\mathbf{a}_{n+1}$. The values of $\mathbf{d}_n, \mathbf{v}_n$, and $\mathbf{a}_n$ are assumed known from the previous time step. Some properties of selected members of the Newmark family are [27]:

1. *Central differences.* In this scheme the parameters β and γ are respectively: 0 and 1/2. This leads to the following expressions for the algorithm for the displacements and velocities:

$$\mathbf{d}_{n+1} \; = \; \mathbf{d}_n \; + \; \Delta t \mathbf{v}_n \; + \; \frac{\Delta t^2}{2}\mathbf{a}_n \tag{3.40}$$

$$\mathbf{v}_{n+1} \; = \; \mathbf{v}_n \; + \; \frac{\Delta t}{2}[\mathbf{a}_n \; + \; \mathbf{a}_{n+1}] \tag{3.41}$$

   This is an *explicit* method if both $\mathbf{M}$ and $\mathbf{C}$ are diagonal matrices. It is second order accurate and conditionally stable.

2. *Trapezoidal Rule.* Here the parameters β and γ are respectively: 1/4 and 1/2, which yields to the following expressions for the displacement:

$$\mathbf{d}_{n+1} \; = \; \mathbf{d}_n \; + \; \Delta t \mathbf{v}_n \; + \; \frac{\Delta t^2}{4}[\mathbf{a}_n \; + \; \mathbf{a}_{n+1}] \tag{3.42}$$

   This is an *implicit method*, and it is unconditionally stable. As for the central difference scheme, the trapezoidal rule is second order accurate.

3. *Linear acceleration method.* The parameters in this scheme are: $\beta = 1/6$ and $\gamma = 1/2$. The expression for the velocities remains as above, whereas the displacements are given by

$$\mathbf{d}_{n+1} = \mathbf{d}_n + \Delta t \mathbf{v}_n + \frac{\Delta t^2}{2}[\frac{2}{3}\mathbf{a}_n + \frac{1}{3}\mathbf{a}_{n+1}] \qquad (3.43)$$

This is an implicit method, but it is not unconditionally stable. It is also a second order accurate method.

One of the earliest works in the area of concurrent time step integration algorithms is that of Noor and Lambiotte [28], in 1978. The CDC Star-100 is the architecture considered there. In this work both the central difference scheme and Newmark implicit schemes were implemented in parallel. In [29] Belytschko and Gilbertsen present a concurrent explicit time integration algorithm for the non-linear equations of motion in structural dynamics. The essential feature in their implementation is that it allows the use of different time steps on different parts of the mesh. The explicit scheme chosen is central differences. To maximize the benefits of vectorization and concurrency, the elements are grouped to obtain vector lengths appropriate for vectorization. Each element group can then be integrated with a different time step. The nodal velocities and displacements are computed when all groups reach t_{mast}, which is the master time. The groups of elements are integrated concurrently and within each group the computations are vectorized.

Flanagan and Taylor [30] have proposed a concurrent explicit time integration algorithm. The main task to be performed during each time step are identified as:

1. Update stresses and assemble external forces.

2. Assemble external forces.

3. Apply kinematic constrains.

4. Update kinematics via lumped mass matrix.

Most of the effort ($\approx$ 75-80%) is spent in the first of these tasks. The method proposed subdivides the stress update into the following subtasks,

1. Uncouple-extract nodal kinematics.

2. Update element internal state.

3. Calculate element nodal force contribution.

4. Couple-assemble nodal forces.

The coupling and uncoupling operations cannot be performed concurrently, and involve gather/scatter steps based on the element connectivity table.

Ortiz and Nour-Omid [31] have advanced a method which shares some of the attributes with both implicit and explicit schemes. The method starts by partitioning the structure into element groups. Each one of these groups is treated implicitly, while the collection of groups is treated explicitly. Details of this method are amplified in chapter IV.

References

1. S. G. Akl, 'The Design and Analysis of Parallel Algorithms', *Wadsworth Pub. Co.*, 1989.

2. A. K. Noor, 'Parallel Processing in Finite Element Analysis,' *Parallel Computations and their Impact on Mechanics*, ed. by A. K. Noor, ASME, New York, 1987.

3. Linda J. Hayes, 'Systolic Arrays for Finite Element Calculations', *Parallel Computations and their Impact on Mechanics*, ed. by A. K. Noor, ASME, New York, 1987.

4. G. F. Carey, 'Parallelism in Finite Element Modeling,' *Communications in Applied Numerical Methods*, Vol. 2, No. 3, 1986, pp. 281-287.

5. G. Rodrigue, 'Some Ideas for Decomposing the Domain of Elliptic Partial Differential Equations in the Schwarz Process,' *Communications in Applied Numerical Methods*, Vol. 2, No. 3, pp. 245-249.

6. J. Flower, S. Otto and M. Salama, 'Optimal Mapping of Irregular Finite Element Domains to Parallel Processors,' *Parallel Computations and their Impact on Mechanics*, ed. by A. K. Noor, ASME, New York, 1987.

7. T. J. R. Hughes, J. Winget, I. Levi, and T. E. Tezduyar, 'New Alternating Direction Procedures in Finite Element Analysis Based Upon EBE Approximate Factorizations,' *Recent Developments in Computer Methods for Nonlinear Solid and Structural Mechanics*, N. Perrone, S. Atluri, eds., ASME, New York, June 1983, pp. 75-109.

8. N. N. Yanenko, 'The Method of Fractional Steps,' *Spriengcr-Verlag*, Berlin, 1971

9. M. Ortiz, P. M. Pinsky and R. L. Taylor, 'Unconditionally Stable Element-by-Element Algorithms for Dynamic Problems,' *Computer Methods in Applied Mechanics and Engineering*, Vol. 36, No. 2, 1983, pp. 223-239

10. J. H. Biffle, 'Indirect Solution of Static Problems using Concurrent Vector Processing Computers,' *Parallel Computations and their Impact on Mechanics*, ed. by A. K. Noor, ASME, New York, 1987.

11. R. S. Varga, 'Matrix Iterative Analysis,' *Prentice-Hall, Inc.*, 1962.

12. D. G. Luenberger, 'Linear and Nonlinear Programming,' *Addison-Wesley Publishing Co.*, 1984.

13. M. Seager, 'Overhead Considerations for Parallelizing Conjugate Gradient,' *Communications in Applied Numerical Methods*, Vol. 2, No. 3, 1986b, pp. 273-279.

14. L. Adams, 'Reordering Computations for Parallel Execution,' *Communications in Applied Numerical Methods*, Vol. 2, No. 3, pp. 263-272.

15. L. M. Adams and J. M. Ortega, 'A multi-color SOR Method for Parallel Computation,' *Proc. 1982 Int. Conf. on Parallel Processing*, IEEE Catalog No. 82ch1794-7, Aug 1982, pp. 53-56.

16. Adams and Jordan, 'Is SOR Color Blind', *ICASE Report No. 85-12*, 1985.

17. S. Doi and K. Shoichi, 'A Parallel Computation Technique for Finite-Element Methods,' *Systems-Computers-Controls*, Vol. 13, No. 2, pp. 76-84.

18. R. G. Melhem, 'On the Design of a Pipelined/Systolic Finite Element System,' *Computers and Structures*, Vol. 20, No. 1-3, 1985a, pp. 67-75.

19. B. Nour-Omid, A. Raefsky and G. Lyzenga, 'Solving Finite Element Equations on Concurrent Computers,' *Parallel Computations and their Impact on Mechanics*, ed. by A. K. Noor, ASME, New York, 1987.

20. A. George and J. W. Liu, 'Computer Solution of Large Sparse Positive Definite Systems', *Prentice Hall Series in Computational Mathematics*, 1981.

21. K. H. Law, 'Systolic Arrays for Finite Element Analysis,' *Computers and Structures*, Vol. 20, No. 1-3, 1985, pp. 55-65

22. C. Farhat, 'Multiprocessors in Computational Mechanics,' *Ph.D dissertation.* University of California at Berkeley, 1986.

23. S. L. Johnsson, 'Highly Parallel Banded Systems Solvers,' *Parallel Computations and their Impact on Mechanics*, ed. by A. K. Noor, ASME, New York, 1987.

24. H. Chen and A. Sameh, 'Highly Linear Algebra Algorithms on the Cedar System', *Parallel Computations and their Impact on Mechanics*, ed. by A. K. Noor, ASME, New York, 1987.

25. T. Belytschko, and T. J. R. Hughes, 'Mesh Partitions of Explicit- Implicit Time Integration,' *Formulations and Computational Algorithms in Finite Element Analysis*, K. Bathe, J. Oden, and W. Wunderlich, eds., MIT Press, Cambridge, Mass., 1976, pp. 673-690.

26. T. J. R. Hughes, K. S. Pister and R. L. Taylor, 'Implicit-Explicit Finite Elements in Nonlinear Transient Analysis,' *Computer Methods in Applied Me-*

chanics and Engineering, Vol 17/18, 1979, pp. 159-182.

27. T. J. R. Hughes and T. Belytschko, 'A Precis of Developments in Computational Methods for Transient Analysis,' *ASME Journal of Applied Mechanics*, Vol. 50, 1983, pp. 1033-1041.

28. A. Noor and J. Lambiotte, 'Finite Element Dynamic Analysis on the CDC STAR-100 Computer,' *Computers and Structures*, Vol. 10, No. 1-2, 1979, pp. 7-19.

29. T. Belytschko and N. Gilbertsen, 'Concurrent and Vectorized Mixed Time, Explicit Nonlinear Structural Dynamics,' *Parallel Computations and their Impact on Mechanics*, ed. by A. K. Noor, ASME, New York, 1987.

30. D. P. Flanagan and L. M. Taylor, 'Structuring Data for Concurrent Vectorized Processing in a Transient Dynamics Finite Element Program,' *Parallel Computations and their Impact on Mechanics*, ed. by A. K. Noor, ASME, New York, 1987.

31. M. Ortiz and B. Nour-Omid, 'Unconditionally stable concurrent procedures for transient finite element analysis,' *Comp. Meth. Appl. Mech. Engng.*, 58, 1986, pp. 151-174.

CHAPTER IV

Group Implicit Algorithms

The concept of Group Implicit (GI) algorithm was introduced by Ortiz and Nour-Omid [1] in 1985. In essence, these algorithms are constructed by partitioning the finite element mesh into groups of elements, and processing each group implicitly and independently over a time step. This last feature introduces the desired concurrency into the computations. The requisite compatibility between the subdomains is enforced *a posteriori*, by means of a mass averaging rule. We show that the resulting algorithms have ranges of unconditional stability similar to those of globally implicit methods. Guidelines are given for choosing the time steps so that accuracy does not deteriorate as the number of element groups is increased.

The appeal of GI algorithms is twofold. Firstly, they are highly parallelizable, with interprocessor communications limited to the exchange of one interface vector per time step. Secondly, they speed up the computations by reducing the equation solving effort, even on a single-processor machine. This is so because, as the subdomains are reduced, the bandwidths of the local arrays decrease steadily. Fill-in by off-diagonal zeros is consequently diminished as well. The result is a net gain in efficiency during the factorization phase.

Our numerical simulations have born out these conclusions: by cofirming the theoretically derived ranges of unconditional stability and accuracy estimates; by demonstrating the low communication overhead incurred during the computations;

and by showing how the equation solving effort is diminished well beyond the linear speed-up expected from concurrency alone. Simulations run on a 32-node hypercube have consistently given efficiencies over 95% on a variety of problems. Tests run on an eight-processor Alliant FX8 have given factorization speed-ups of the order of 34 in selected applications. In view of these results, it would appear that GI algorithms hold considerable promise for application in nonlinear structural dynamics problems, particularly on medium-grained and fine-grained parallel machines, such as the Alliant and Cray series and the ETA 10 machine.

4.1. Theoretical Basis

Throughout this work we focus on the structural dynamics problem governed by the semidiscrete equations of motion of the form

$$\begin{aligned} &\mathbf{M}\ddot{\mathbf{d}}(t) \ + \ \mathbf{G}\big(\mathbf{d}(t), \dot{\mathbf{d}}(t)\big) = \mathbf{f}(t), \\ &\mathbf{d}(0) = \mathbf{d}_0, \\ &\dot{\mathbf{d}}(0) = \mathbf{v}_0 \end{aligned} \tag{4.1}$$

where, following standard notation, $\mathbf{M}$ signifies the mass matrix, $\mathbf{G}$ and $\mathbf{f}$ the internal and external force vectors, $\mathbf{d}$ the displacement vector, $\mathbf{d}_0$ and $\mathbf{v}_0$ the initial displacements and velocities, respectively, and a superimposed dot is used to denote differentiation with respect to time t. The tangent stiffness and damping matrices of the structure are defined as

$$\begin{aligned} \mathbf{K} &= \partial \mathbf{G}(\mathbf{d}, \dot{\mathbf{d}}) / \partial \mathbf{d}, \\ \mathbf{C} &= \partial \mathbf{G}(\mathbf{d}, \dot{\mathbf{d}}) / \partial \dot{\mathbf{d}}, \end{aligned} \tag{4.2}$$

respectively. In the linear case, $\mathbf{K}$ and $\mathbf{C}$ are constant and

$$\mathbf{G}(\mathbf{d}, \dot{\mathbf{d}}) = \mathbf{K}\mathbf{d} + \mathbf{C}\dot{\mathbf{d}}. \tag{4.3}$$

The linearized equations of motion can be written as

$$\begin{aligned} &\mathbf{M}\ddot{\mathbf{d}}(t) + \mathbf{C}\dot{\mathbf{d}}(t) + \mathbf{K}\mathbf{d}(t) = \mathbf{f}(t), \\ &\mathbf{d}(0) = \mathbf{d}_0, \\ &\dot{\mathbf{d}}(0) = \mathbf{v}_0. \end{aligned} \tag{4.4}$$

These equations can be reduced to their first order correspondents by means of a change of variables. Introducing

$$\mathbf{z} = \begin{bmatrix} \mathbf{d}(t) \\ \dot{\mathbf{d}}(t) \end{bmatrix}, \tag{4.5}$$

equation (4.4) takes the following form

$$\begin{aligned} &\begin{bmatrix} \mathbf{M} & \mathbf{0} \\ \mathbf{0} & \mathbf{K} \end{bmatrix} \begin{bmatrix} \ddot{\mathbf{d}}(t) \\ \dot{\mathbf{d}}(t) \end{bmatrix} + \begin{bmatrix} \mathbf{C} & \mathbf{K} \\ \mathbf{K} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \dot{\mathbf{d}}(t) \\ \mathbf{d}(t) \end{bmatrix} = \begin{bmatrix} \mathbf{f}(\mathrm{t}) \\ \mathbf{0} \end{bmatrix} \\ &\begin{bmatrix} \dot{\mathbf{d}}(0) \\ \mathbf{d}(0) \end{bmatrix} = \begin{bmatrix} \mathbf{v}_0 \\ \mathbf{d}_0 \end{bmatrix}. \end{aligned} \tag{4.6}$$

In matrix notation the reduced system of equations can be written as

$$\begin{aligned} &\mathbf{A}\dot{\mathbf{z}}(t) + \mathbf{B}\mathbf{z}(t) = \mathbf{g}(t), \\ &\mathbf{z}(0) = \mathbf{z}_0. \end{aligned} \tag{4.7}$$

The ***energy norm*** corresponding to system (4.7) is defined as

$$\|\mathbf{z}\| = (\mathbf{z}^T\mathbf{A}\mathbf{z})^{1/2} \tag{4.8}$$

For instance, for (4.6), the square of the energy norm is twice the total (strain + kinetic) energy of the system, and hence the name assigned to the norm.

An algorithm for integrating (4.7) is defined as a matrix $\mathbf{F}(h)$, or *amplification matrix*, such that

$$\mathbf{z}_{n+1} = \mathbf{F}(h)\mathbf{z}_n, \tag{4.9}$$

where h is the time step and z_n is the approximate solution at $t_n = nh, n = 1, 2, 3, \ldots$, with the property that $\mathbf{z}_{t/h} \to \mathbf{z}(t)$ as $h \to 0$. For linear systems of ODE's, an algorithm is convergent iff it is consistent and stable. Consistency of the algorithm with the governing equations is defined as the condition that

$$\lim_{h \to 0} \frac{\mathbf{z}_{n+1} - \mathbf{z}_n}{h} = \dot{\mathbf{z}}_n = -\mathbf{A}^{-1}\mathbf{B}\mathbf{z}_n. \tag{4.10}$$

This condition can be rewritten in terms of the amplification matrix by substituting (4.9) into (4.10), which yields

$$\left[\frac{d}{dh}\mathbf{F}(h)\right]_{h=0} = -\mathbf{A}^{-1}\mathbf{B}. \tag{4.11}$$

The stability of the algorithm, on the other hand, requires that

$$\|\mathbf{F}(h)\| \leq 1 \tag{4.12}$$

where the energy norm of the amplification matrix is defined as

$$\|\mathbf{F}(h)\| = \max_{\mathbf{z}} \frac{\|F(h)\mathbf{z}\|}{\|\mathbf{z}\|} \tag{4.13}$$

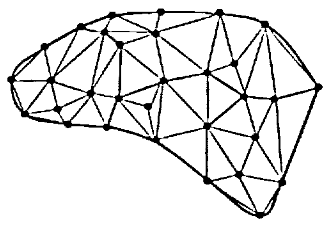


Figure 1. Model.

A first step in constructing the method is to partition the structure into groups of elements. The finite element mesh can then be viewed as a collection of disconnected subdomains, Figures 1 and 2. The field variables within a generic subdomain r are fully described in terms of local arrays, such as $\mathbf{z}^r$.

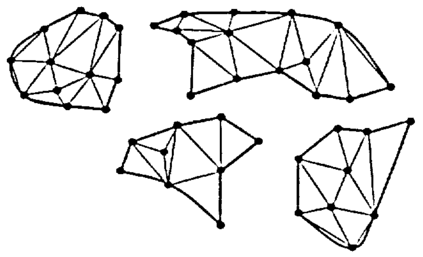


Figure 2. Partitioned mesh

The *extended* variable array $\bar{\mathbf{z}} = \{\mathbf{z}^1, ..., \mathbf{z}^r, ..., \mathbf{z}^s\}$, where s is the number of subdomains, completely describes the structure. Moreover, $\bar{\mathbf{z}}$ contains the same information as $\mathbf{z}$, the global nodal array. The relation between them is given by the following linear mapping

$$\mathbf{z}_r = \mathbf{L}_r \mathbf{z}, \tag{4.14}$$

where $\mathbf{L}_r$ is a Boolean matrix which localizes $\mathbf{z}$ to the subdomain r to obtain $\mathbf{z}_r$. In matrix form this operation can be written as

$$\bar{\mathbf{z}} = \begin{bmatrix} \mathbf{L}_1 \\ \mathbf{L}_2 \\ \vdots \\ \mathbf{L}_s \end{bmatrix} \mathbf{z} = \mathbf{L}\mathbf{z}. \tag{4.15}$$

It is readily verified by recourse to the principle of virtual work that

$$\mathbf{g} = \mathbf{L}^T \bar{\mathbf{g}}, \tag{4.16}$$

where $\mathbf{g}$ and $\bar{\mathbf{g}}$ denote the global force array and the extended force array $\{\mathbf{g}^1, ..., \mathbf{g}^r, ..., \mathbf{g}^s\}$. The extended matrices corresponding to $\bar{\mathbf{z}}$ are defined as

$$\bar{\mathbf{A}} = \begin{bmatrix} \mathbf{A}^1 & & & \\ & \mathbf{A}^2 & & \\ & & \cdot & \\ & & & \mathbf{A}^s \end{bmatrix}, \qquad \bar{\mathbf{B}} = \begin{bmatrix} \mathbf{B}^1 & & & \\ & \mathbf{B}^2 & & \\ & & \cdot & \\ & & & \mathbf{B}^s \end{bmatrix}.$$

The assembly operation for global arrays can then be expressed in the form

$$\begin{aligned} \mathbf{A} &= \mathbf{L}^T \bar{\mathbf{A}} \mathbf{L}, \\ \mathbf{B} &= \mathbf{L}^T \bar{\mathbf{B}} \mathbf{L}, \end{aligned} \tag{4.17}$$

The essential idea of the methods derived here is to allow the various subdomains in the partition to evolve independently over one time step, and to restore compatibility by somehow *projecting* the extended solution so obtained onto a suitably defined compatible solution. Three different methods for constructing algorithms of this type are given next.

4.1.1. GI Algorithms for First Order Systems

In this section we focus on first order systems of the type (4.7). For simplicity, we consider the unforced case, $\mathbf{g} = \mathbf{0}$. A general class of parallel algorithms can be defined as follows:

- Localize initial conditions $\mathbf{z}_n$ to the subdomains to obtain the extended array $\bar{\mathbf{z}}_n$.

- Update the extended array by solving the decoupled equations of motion at the subdomain level

$$\mathbf{A}^r \dot{\mathbf{z}}^r + \mathbf{B}^r \mathbf{z}^r = \mathbf{0}. \tag{4.18}$$

The extended predictor obtained this way is called $\bar{\mathbf{z}}^*_{n+1}$.

- The extended predictor information is multivalued at the nodes that belong to more than one subdomain. The algorithm is then completed by averaging those values at the interface nodes by means of a suitable averaging rule, so that consistency is restored.

The amplification matrix for this algorithm is given by

$$\mathbf{F}(h) = \mathbf{P}\bar{\mathbf{F}}(h)\mathbf{L}, \tag{4.19}$$

where $\bar{\mathbf{F}}(h) = diag(\mathbf{F}^1(h), ..., \mathbf{F}^r(h), ..., \mathbf{F}^s(h))$, $\mathbf{F}^r(h)$ is the amplification matrix consistent with equation (4.18) and the matrix $\mathbf{P}$ defines a projection form the space of extended arrays to the subspace of compatible arrays.

The subdomain algorithms $\mathbf{F}^r(h)$, that constitute the extended algorithm $\bar{\mathbf{F}}(h)$ in (4.19), must be consistent with the decoupled equations (4.18), i. e.,

$$\left[\frac{d}{dh}\bar{\mathbf{F}}(h)\right]_{h=0} = -\bar{\mathbf{A}}^{-1}\bar{\mathbf{B}}. \tag{4.20}$$

Using (4.19) one has

$$\mathbf{A}\left[\frac{d}{dh}\mathbf{F}(h)\right]_{h=0} = \mathbf{AP}\left[\frac{d}{dh}\bar{\mathbf{F}}(h)\right]_{h=0}\mathbf{L}, \tag{4.21}$$

But by the consistency (4.20) of the local algorithms, equation (4.21) may be rewritten as

$$\mathbf{A}\left[\frac{d}{dh}\mathbf{F}(h)\right]_{h=0} = \mathbf{AP}(-\bar{\mathbf{A}}^{-1}\bar{\mathbf{B}})\mathbf{L}. \tag{4.22}$$

On the other hand, using (4.11)

$$\mathbf{A}\left[\frac{d}{dh}\mathbf{F}(h)\right]_{h=0} = -\mathbf{B} = -\mathbf{L}^T\bar{\mathbf{B}}\mathbf{L}. \tag{4.23}$$

Comparing (4.22) and (4.23) one concludes that

$$\mathbf{AP}\bar{\mathbf{A}}^{-1} = \mathbf{L}^T \tag{4.24}$$

or,

$$\mathbf{P} = \mathbf{A}^{-1}\mathbf{L}^T\bar{\mathbf{A}}. \tag{4.25}$$

From equation (4.25) it is apparent that the sought projection $\mathbf{P}$ is a *mass* averaging rule on the interface degrees of freedom. This rule can be expressed as

$$\mathbf{z}_{n+1} = \mathbf{P}\mathbf{z}^*_{n+1} = \mathbf{A}^{-1}\sum_{r=1}^{s}\mathbf{A}^r\mathbf{z}^{*r}_{n+1}. \tag{4.26}$$

Thus, z^{*r}_{n+1} is first to be weighted by the subdomain mass matrix $\mathbf{A}^r$, the resulting local vectors assembled into a global array which is finally multiplied by $\mathbf{A}^{-1}$.

4.1.2. Interface Compatibility as a Constraint

In [2], a class of concurrent procedures was obtained by regarding the compatibility conditions between subdomains as algebraic constraints operating on the extended solution array. The effect of these constraints is built into the extended governing equations by means of Lagrange multipliers. Physically, these represent the reactions between subdomains. By using splitting techniques, the constraints are relaxed during each time step, which results in the desired level of concurrency. Compatibility is enforced *a posteriori* on the extended predictor. This alternative methodology is suggestive of various generalizations of the method discussed in the preceding section, and is outlined next.

Start by writing (4.7) in the form

$$\begin{bmatrix} \bar{\mathbf{A}} & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}\begin{bmatrix} \dot{\bar{\mathbf{z}}} \\ \dot{\Lambda} \\ \dot{\mathbf{z}} \end{bmatrix} + \begin{bmatrix} \bar{\mathbf{B}} & \mathbf{I} & 0 \\ \mathbf{I} & 0 & -\mathbf{L} \\ 0 & -\mathbf{L}^T & 0 \end{bmatrix}\begin{bmatrix} \bar{\mathbf{z}} \\ \Lambda \\ \mathbf{z} \end{bmatrix} = \begin{bmatrix} \bar{\mathbf{g}} \\ 0 \\ 0 \end{bmatrix} \tag{4.27}$$

or, in full,

$$\begin{aligned} &\bar{\mathbf{A}}\dot{\bar{\mathbf{z}}} + \bar{\mathbf{B}}\bar{\mathbf{z}} + \Lambda = \bar{\mathbf{g}} \\ &\bar{\mathbf{z}} - \mathbf{L}\mathbf{z} = 0 \\ &-\mathbf{L}^T\Lambda = 0 \end{aligned} \tag{4.28}$$

The first equation governs the evolution of the decoupled subdomains, the second is a statement of the compatibility condition, and the third requires that the reactions between the subdomains be equilibrated. To obtain a class of concurrent algorithms, we decompose the evolutionary operator in (4.27) as

$$\begin{bmatrix} \bar{\mathbf{B}} & \mathbf{I} & \mathbf{0} \\ \mathbf{I} & \mathbf{0} & -\mathbf{L} \\ \mathbf{0} & -\mathbf{L}^T & \mathbf{0} \end{bmatrix} = \begin{bmatrix} \bar{\mathbf{B}} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix} + \begin{bmatrix} \mathbf{0} & \mathbf{I} & \mathbf{0} \\ \mathbf{I} & \mathbf{0} & -\mathbf{L} \\ \mathbf{0} & -\mathbf{L}^T & \mathbf{0} \end{bmatrix} \tag{4.29}$$

Next, we introduce a product formula based on this split [3]. The first step of the product formula is governed by the equations

$$\begin{bmatrix} \bar{\mathbf{A}} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \dot{\bar{\mathbf{z}}} \\ \dot{\Lambda} \\ \dot{\mathbf{z}} \end{bmatrix} + \begin{bmatrix} \bar{\mathbf{B}} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \bar{\mathbf{z}} \\ \Lambda \\ \mathbf{z} \end{bmatrix} = \begin{bmatrix} \bar{\mathbf{g}} \\ \mathbf{0} \\ \mathbf{0} \end{bmatrix} \tag{4.30}$$

which reduce to

$$\bar{\mathbf{A}}\dot{\bar{\mathbf{z}}} + \bar{\mathbf{B}}\bar{\mathbf{z}} = \bar{\mathbf{g}} \tag{4.31}$$

These are simply the governing equations for the decoupled subdomains. The initial conditions for this step are simply $\bar{\mathbf{z}}_n = \mathbf{L}\mathbf{z}_n$, and the result is an extended predictor $\bar{\mathbf{z}}^*_{n+1}$.

The second step of the product formula is governed by the remainder of the evolutionary operator, i. e.,

$$\begin{bmatrix} \bar{\mathbf{A}} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \dot{\bar{\mathbf{z}}} \\ \dot{\Lambda} \\ \dot{\mathbf{z}} \end{bmatrix} + \begin{bmatrix} \mathbf{0} & \mathbf{I} & \mathbf{0} \\ \mathbf{I} & \mathbf{0} & -\mathbf{L} \\ \mathbf{0} & -\mathbf{L}^T & \mathbf{0} \end{bmatrix} \begin{bmatrix} \bar{\mathbf{z}} \\ \Lambda \\ \mathbf{z} \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \end{bmatrix} \tag{4.32}$$

or, in full,

$$\begin{aligned} \bar{\mathbf{A}}\dot{\bar{\mathbf{z}}} + \Lambda &= \mathbf{0} \\ \bar{\mathbf{z}} - \mathbf{L}\mathbf{z} &= \mathbf{0} \\ -\mathbf{L}^T\Lambda &= \mathbf{0} \end{aligned} \tag{4.33}$$

The initial conditions for this phase of the algorithm are the results from the first step of the product formula, i. e., the extended predictor $\bar{\mathbf{z}}^*_{n+1}$. The outcome of the second step is the updated solution $\mathbf{z}_{n+1}$. Eqs. (4.33) may be discretized by means of the backward-Euler algorithm, to yield

$$\begin{aligned} &\bar{\mathbf{A}}(\bar{\mathbf{z}}_{n+1} - \bar{\mathbf{z}}^*_{n+1}) + \Delta t \mathbf{\Lambda}_{\mathbf{n+1}} = \mathbf{0} \\ &\bar{\mathbf{z}}_{n+1} - \mathbf{L}\mathbf{z}_{n+1} = \mathbf{0} \\ &-\mathbf{L}^T \mathbf{\Lambda}_{\mathbf{n+1}} = \mathbf{0} \end{aligned} \tag{4.34}$$

The solution of this system of algebraic equations can be found readily. Combine the first two equations of (4.34) to obtain

$$\bar{\mathbf{A}}\mathbf{L}\mathbf{z}_{n+1} = \bar{\mathbf{A}}\bar{\mathbf{z}}^*_{n+1} - \Delta t \mathbf{\Lambda}_{\mathbf{n+1}} \tag{4.35}$$

Multiply this expression by $\mathbf{L}^T$ and use the third of (4.34), with the result

$$\mathbf{L}^T\bar{\mathbf{A}}\mathbf{L}\mathbf{z}_{n+1} = \mathbf{M}\mathbf{z}_{n+1} = \mathbf{L}^T\bar{\mathbf{A}}\bar{\mathbf{z}}^*_{n+1} \tag{4.36}$$

Finally, solve for $\mathbf{z}_{n+1}$ to obtain

$$\mathbf{z}_{n+1} = \mathbf{M}^{-1}\mathbf{L}^T\bar{\mathbf{A}}\bar{\mathbf{z}}^*_{n+1} \tag{4.37}$$

which is a restatement of the mass-averaging rule formulated in Section 4.1.1.

Thus, the algorithm derived in Section 4.1.1 is recovered as a product formula associated with a particular splitting of the lagrangean form of the governing equations as expressed in (4.27). From general results concerning product formulae [3], it follows that the resulting algorithm is unconditionally stable provided that the extended equations (4.31) are integrated by means of an unconditionally stable scheme. An alternative proof of this fact was given in [2]. It also follows from the general theory that the algorithm can only be expected to be first order accurate even when the extended equations (4.31) are integrated by means of a second order accurate scheme.

4.1.3. Group Implicit Algorithms in Structural Dynamics

In [4] a class of concurrent algorithms for structural dynamics, named there Group Implicit algorithms, were proposed. Although the methodology derived in Sections 4.1.1 and 4.1.2 can be applied to second order ODE's by recourse to the equivalence with linear systems established in (4.6), it was found, partly by trial and error, that GI algorithms generally result in superior accuracy. The motivation for GI algorithms is partly provided by the work of Petzold on systems of ODE's with algebraic constraints. Gear and Petzold showed in [5] that the order of accuracy of numerical methods of solution of differential/algebraic systems is enhanced if the constraints are differentiated and enforced in differential form. In structural dynamics, this means enforcing compatibility of *accelerations* between subdomains, rather than displacements or velocities.

A method suggested by these ideas is outlined in Box 1, for the nonlinear case, and in Box 2, for the special case of a linear structure. As may be seen, the predictor and corrector phases are chosen to be identical to those in Newmark's method [6]. The present scheme is at variance with Newmark's algorithm in the equation solving phase, where concurrency is introduced. Thus, the predictor displacements $\tilde{\mathbf{d}}_{n+1}$ for time t_{n+1} are first localized into the subdomains to obtain a collection of local predictors $\{\tilde{\mathbf{d}}^r_{n+1},\ r = 1, \ldots, s\}$, where s denotes the number of subdomains in the partition. The corresponding local acceleration arrays $\tilde{\mathbf{a}}^r_{n+1}$ are then computed from $\mathbf{d}^r_{n+1}$ by applying Newmark's update at the subdomain level. To restore compatibility between subdomains the mass averaging rule

$$\mathbf{a}_{n+1} = \mathbf{M}^{-1} \sum_{r=1}^{s} \mathbf{M}^r \tilde{\mathbf{a}}^r_{n+1} \tag{4.38}$$

must be applied. This completes one application of the algorithm.

Box 1. A Group Implicit Concurrent Algorithm

- Predictor phase:

$$\tilde{\mathbf{d}}_{n+1} = \mathbf{d}_n + \Delta t \mathbf{v}_n + (1/2 - \beta)\Delta t^2 \mathbf{a}_n$$

$$\tilde{\mathbf{v}}_{n+1} = \mathbf{v}_n + (1 - \gamma)\Delta t \mathbf{a}_n$$

- Equation solving phase:

```
$\mathbf{a}_{n+1} = 0$
    $for\ r = 1, s\ do$
    $Solve:$
    $\mathbf{M}^r \tilde{\mathbf{a}}^r_{n+1} + \mathbf{G}^r(\tilde{\mathbf{d}}^r_{n+1} + \beta\Delta t^2 \tilde{\mathbf{a}}^r_{n+1}, \tilde{\mathbf{v}}^r_{n+1} + \gamma\Delta t \tilde{\mathbf{a}}^r_{n+1}) = 0$
    $\mathbf{a}_{n+1} \leftarrow \mathbf{a}_{n+1} + \mathbf{M}^r \tilde{\mathbf{a}}^r_{n+1}$
$\mathbf{a}_{n+1} \leftarrow \mathbf{M}^{-1} \mathbf{a}_{n+1}$
```

- Corrector phase:

$$\mathbf{d}_{n+1} = \tilde{\mathbf{d}}_{n+1} + \beta\Delta t^2 \mathbf{a}_{n+1}$$

$$\mathbf{v}_{n+1} = \tilde{\mathbf{v}}_{n+1} + \gamma\Delta t \mathbf{a}_{n+1}$$

Box 2. Group Implicit Algorithm - Linear case

- Predictor phase:

$$\tilde{\mathbf{d}}_{n+1} = \mathbf{d}_n + \Delta t \mathbf{v}_n + (1/2 - \beta)\Delta t^2 \mathbf{a}_n$$
$$\tilde{\mathbf{v}}_{n+1} = \mathbf{v}_n + (1-\gamma)\Delta t \mathbf{a}_n$$

- Equation solving phase:

```
$\mathbf{a}_{n+1} = 0$
    $for\ r = 1, s\ do$
    $\tilde{\mathbf{a}}^r_{n+1} = -(\mathbf{M}^r + \gamma\Delta t\mathbf{C}^r + \beta\Delta t^2\mathbf{K}^r)^{-1}\mathbf{K}^r\tilde{\mathbf{d}}^r_{n+1}$
    $\mathbf{a}_{n+1} \leftarrow \mathbf{a}_{n+1} + \mathbf{M}^r\tilde{\mathbf{a}}^r_{n+1}$
$\mathbf{a}_{n+1} \leftarrow \mathbf{M}^{-1}\mathbf{a}_{n+1}$
```

- Corrector phase:

$$\mathbf{d}_{n+1} = \tilde{\mathbf{d}}_{n+1} + \beta\Delta t^2 \mathbf{a}_{n+1}$$
$$\mathbf{v}_{n+1} = \tilde{\mathbf{v}}_{n+1} + \gamma\Delta t \mathbf{a}_{n+1}$$

It is noted that the algorithm reduces to Newmark's method for the trivial partition, $s = 1$. For $s > 1$, the subdomains are effectively decoupled during the equation solving phase. Consequently, they can all be processed in parallel. No global stiffness arrays need to be formed or factorized at any time during the computations. In addition, the communication between processors during each time step reduces to the transfer of the local acceleration arrays. This keeps the communication overhead to a minimum.

In the nonlinear case, local systems of nonlinear equations need to be solved for the accelerations $\tilde{\mathbf{a}}^r_{n+1}$. In the procedure outlined in Box 2, this may be accomplished by means of a local Newton-Raphson iteration.

4.2. Algorithmic Properties

In this section, the Group Implicit algorithms are analyzed. It is shown that, although the algorithm has the same range of unconditional stability as Newmark's method, a Courant type condition must be complied with to avoid accuracy breakdown in the form of inadmissible phase errors. Numerical examples on a membrane follow to show that such condition leads to a conservative criterion. Theoretical estimates of the efficiency of such methods are presented.

4.2.1. Accuracy. Phase Errors

Past experience with semi-implicit algorithms points to their limited ability to propagate information between distant parts of the structure is the main source of numerical error. For the method under consideration, the fact that information is exchanged only between neighboring subdomains during each time step places some restrictions on the time step size necessary to attain a given level of accuracy. This limitation is common to all semi-implicit algorithms and was first noted by Mullen and Belytschko [7]. Although their original analysis was specifically concerned with Trujillo's algorithm, the main conclusions carry over to the present setting as well, as shown next.

A one-dimensional continuum undergoing displacements $u(x,t)$ governed by the wave equation

$$\ddot{u} = c^2 u_{,xx} \tag{4.39}$$

is considered next, where c is the celerity of the waves. The continuum is then discretized into a finite element mesh of uniform size Δx consisting of two-node linear elements. For simplicity, element-by-element partitions of the mesh are considered first, in which the subdomains are taken to coincide with the elements themselves. Under these conditions, the local acceleration predictors are computed entirely at the element level. The local amplification matrices (see Box 2), for the undamped case, are computed to be

$$\mathbf{F}(\Delta t) = \mathbf{M}^e(\mathbf{M}^e + \beta \Delta t^2 \mathbf{K}^e)^{-1}\mathbf{K}^e, \tag{4.40}$$

which in the case considered here reduces to

$$\mathbf{F}(\Delta t) = \mathbf{K}^e/(1 + \beta(2r)^2), \tag{4.41}$$

where, r is the Courant number and is defined as

$$r = \frac{c\Delta t}{\Delta x}. \tag{4.42}$$

The equation solving phase of the concurrent algorithm takes, then, the trivial form

$$\mathbf{M}\mathbf{a}_{n+1} + \frac{1}{1+\beta(2r)^2}\mathbf{K}\tilde{\mathbf{d}}_{n+1} = \mathbf{0} \tag{4.43}$$

In full, these equations read

$$a_j^{n+1} = \frac{c^2/\Delta x^2}{1+\beta(2r)^2}(\tilde{d}_{j-1}^{n+1} + \tilde{d}_{j+1}^{n+1} - 2\tilde{d}_j^{n+1}) \tag{4.44}$$

where the symbols d_j^n, v_j^n and a_j^n are used to denote the displacement, velocity and acceleration at $x = j\Delta x$ and $t = n\Delta t$.

Taking for simplicity $\gamma = 1/2$ and $\beta = 1/4$, the preditor phase reduces to

$$\tilde{\mathbf{d}}_{n+1} = \mathbf{d}_n + \Delta t\mathbf{v}_n + \frac{\Delta t^2}{4}(\mathbf{a}_n + \mathbf{a}_{n+1}) \tag{4.45}$$

and the corrector phase to

$$\mathbf{d}_{n+1} = \tilde{\mathbf{d}}_{n+1} + \frac{\Delta t^2}{4}\mathbf{a}_{n+1} \tag{4.46}$$

$$\mathbf{v}_{n+1} = \frac{\Delta t}{2}(\mathbf{a}_n + \mathbf{a}_{n+1}) \tag{4.47}$$

These equations can be combined to obtain

$$\mathbf{d}_{n+1} = \mathbf{d}_n + \frac{\Delta t}{2}(\mathbf{v}_n + \mathbf{v}_{n+1}) \tag{4.48}$$

Next, a simple wave

$$d_j^n = Ae^{i(\omega n\Delta t + kj\Delta x)}$$
$$v_j^n = Be^{i(\omega n\Delta t + kj\Delta x)} \qquad (4.49)$$
$$a_j^n = Ce^{i(\omega n\Delta t + kj\Delta x)}$$

is considered, where ω and k are the frequency and wave number, A, B and C are the amplitudes of displacement, velocity and acceleration, respectively, and $i = \sqrt{-1}$. Substituting (4.49) into (4.48) and (4.47) one obtains

$$B = \frac{2}{\Delta t}\,\frac{e^{i\omega\Delta t} - 1}{e^{i\omega\Delta t} + 1}\,A, \qquad C = \left(\frac{2}{\Delta t}\,\frac{e^{i\omega\Delta t} - 1}{e^{i\omega\Delta t} + 1}\right)^2 A \qquad (4.50)$$

whereby the amplitudes of velocity and acceleration are related to the amplitude of displacement.

Substituting (4.49) and (4.50) into (4.44) and making use of (4.45), simple manipulations result in the transcendental equation

$$\frac{c_p}{c} = \frac{1}{k\Delta x}\,\frac{1}{r}\left[1 \; - \; \frac{r^2}{1+r^2}(1 - cos k\Delta x)\right] \qquad (4.51)$$

where $c_p = \omega/k$ is the celerity at which the wave is propagated by the algorithm.

A plot of eq. (4.51) is shown in Figure 3. For comparison, the corresponding relation for Newmark's algorithm is depicted in Figure 4.

It is seen from this plots that both algorithms retard the waves as the ratio $r = c\Delta t/\Delta x$ is increased. The retardation effect is worst for short wave lengths. The differences between both algorithms become more apparent in the long wave length range, owing to the fact that the concurrent algorithm exhibits a maximum celerity

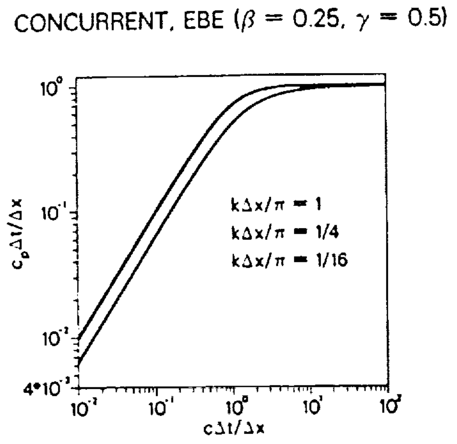


Figure 3. Transcendental equation.

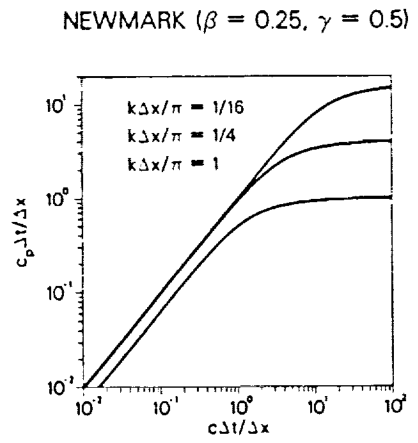


Figure 4. Newmark's Algorithm relation.

$c_{max} = \Delta x/\Delta t$, independent of the wave length. In other words, the element-by-element concurrent algorithm can propagate information at a maximum rate of one mesh size per time step independently of the wave length, a limitation which is not shared by Newmark's method. Clearly, this is a consequence of the fact that the concurrent algorithm allows only for next neighbor interactions between the subdomains over a time step.

For arbitrary partitions into subdomains of length L_s, the above reasoning leads to the conclusion that the celerity of the waves as computed from the concurrent algorithm is bounded by the maximum value

$$c_{max} = \frac{L_s}{\Delta t} \tag{4.52}$$

independent of the wave length. Therefore, it is clear that for the computations to be accurate the time step size has to be chosen such that $c_{max} \leq c$, i. e.,

$$\Delta t \leq \frac{L_s}{c}. \tag{4.53}$$

This is a Courant-type condition based on the dimensions of the subdomains.

Condition (4.53) places some restrictions on the time step size to be used in the computations. It should be emphasized that condition (4.53) stems from accuracy rather than stability considerations. In fact, the algorithm is unconditionally stable, as shown in section 4.2.2, and the solution remains bounded always. This limitation is common to most unconditionally stable semi-implicit algorithms [7]. It is noticed, however, that the Courant condition (4.53) is formulated on the basis of the subdomain size L_s. This is in contrast to methods of explicit integration for which the Courant stability condition is based on the mesh size. Thus, for coarse partitions of the mesh comprising a small number of relatively large subdomains, condition (4.53) is far less stringent than the stability requirements for explicit integration. This enables the use of practical time step sizes commensurate with those appropriate for implicit methods.

The next set of examples aims at testing the hypothesis formulated above, namely, that the accuracy of the concurrent algorithm in wave propagation computations is governed by a Courant-type condition based on the dimensions of the subdomains. The test problem used for this purpose concerns a square membrane of size L supported on stiff springs all around its perimeter. The problem is made nonlinear by supporting the membrane on a nonlinear elastic foundation obeying a force-deflection law

$$f = [1 + a(w/w_0)^2]k_f w \tag{4.54}$$

where k, a and w_0 are material constants. The values of the parameters used in the computations are: $c \equiv \sqrt{k_m/\rho} = 1$, $k_f/(k_m/L) = 1$, $a/(w_0/L) = 1$ and $k_b/(k_m/L) = 10^4$, where k_m, k_f and k_b are the stiffness of the membrane, foundation and boundary springs, respectively, and ρ is the mass density of the membrane. In all the results reported below, deflections are normalized by L and time by L/c. The initial conditions investigated consist of a uniform initial velocity $v_0 = c$ imposed on the underformed membrane. In the linear case, these initial conditions excite all the modes of vibration of the membrane.

The reason for supporting the membrane on stiff springs rather than on rigid supports is to illustrate the importance of unconditional stability in inertia-dominated structural computations. In the linear case, the effect of introducing the stiff supports is to add a set of very high frequency components to the spectrum of the structure. If the main interest of the analysis lies in the response of the membrane, methods which accurately integrate the low frequency components without having to resolve the short periods of vibration become advantageous. This property is tantamount to unconditional stability. By contrast, conditionally stable methods

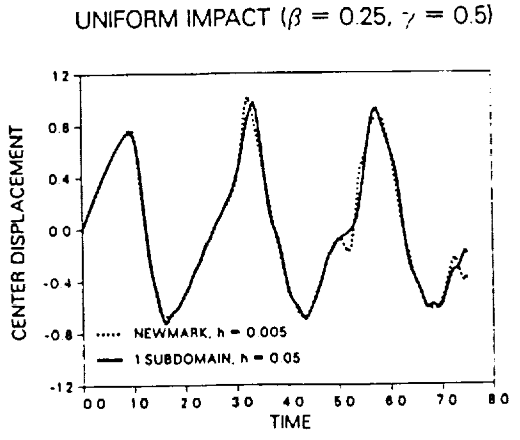


Figure 5. Uniform Impact ($\beta = 0.25,\ \gamma = 0.5$)

such as explicit integration are restricted to time step sizes governed by the high frequency components. In the example under consideration, the critical time step for explicit integration can be made arbitrarily small by increasing the stiffness of the boundary supports. This process leaves the response of the membrane virtually unchanged, and thus renders explicit integration increasingly inadequate. By contrast, the concurrent algorithms under consideration enjoy the unconditional stability property and the time step can be chosen independently of the stiffness of the boundary springs without any appreciable effect on the computed response of the membrane. Thus, in the context of structural computations on parallel machines this simple example illustrates the importance of achieving concurrency and accuracy without compromising stability.

Figures 5-10 show the results obtained from the concurrent algorithm for different degrees of mesh refinement and time steps chosen according to the Courant contition (4.53). Since the size L_s of the subdomains decreases with the number NS of subdomains as $L_s = L/\sqrt{NS} \sim NS^{-1/2}$, the Courant criterion calls for reducing the time step also as $NS^{-1/2}$ in order to maintain the level of accuracy. Figure 5 shows the results corresponding to the trivial partition, for which Newmark's method is recovered, and a time step $\Delta t = 0.05$.

Also shown for reference in Figure 5 are the results obtained from the direct application of Newmark's method with a time step which renders the algorithm virtually exact. Thus, Figure 5 illustrates the kind of accuracy which is obtained from Newmark's method for $\Delta t = 0.05$. Next, Figure 6 shows the results obtained from the concurrent algorithm with $NS = 4$ and a time step $\sqrt{NS} = 2$ times smaller than that used with Newmark's method. Comparing Figures 5 and 6, it is concluded that the level of accuracy achieved from the concurrent algorithm is comparable to that obtained from Newmark's method, with $\Delta t = 0.05$.

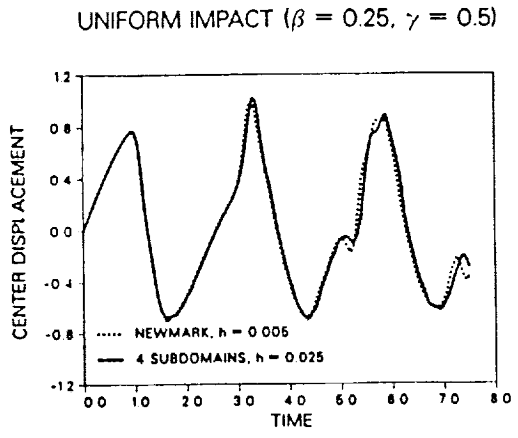


Figure 6. Uniform Impact ($\beta = 0.25$, $\gamma = 0.5$)

Figures 7 and 8, depict the results corresponding to $NS = 16$ and $\Delta t = 0.0125$, and to $NS = 64$ and $\Delta t = 0.00625$, which again do not exhibit any appreciable accuracy deterioration with respect to Newmark's algorithm.

By the time the number of subdomains is increased to 256 and the time step is reduced to $\Delta t = 0.003125$, it becomes apparent that the accuracy of the computations is not only maintained but is improved significantly, Figure 9. For an element-by-element partition with $\Delta t = 0.0015625$, Figure 10, the computed results are virtually exact. It should be emphasized that all the time steps utilized

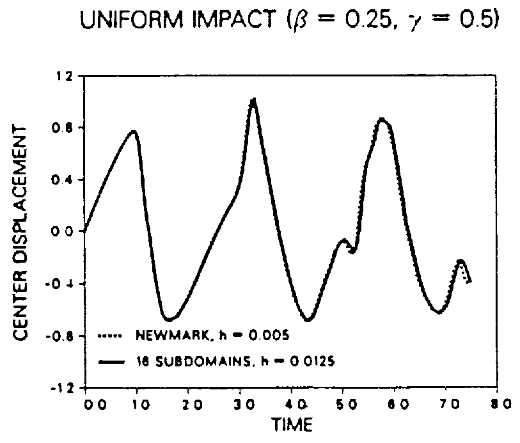


Figure 7. Uniform Impact ($\beta = 0.25$, $\gamma = 0.5$)

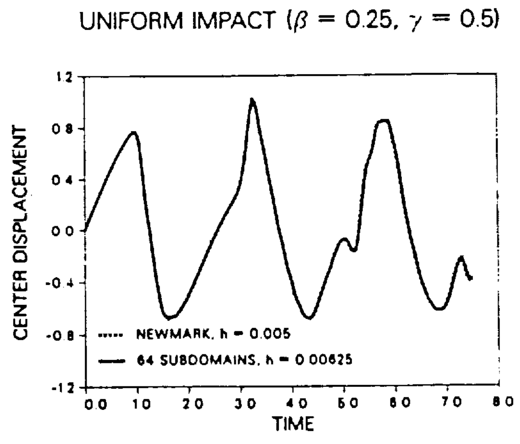


Figure 8. Uniform Impact ($\beta = 0.25$, $\gamma = 0.5$)

in the computations are orders of magnitude above the critical time step for explicit integration by virtue of the stiff boundary supports. As discussed above, in cases like this explicit methods are placed at a clear disadvantage with respect to unconditionally stable algorithms.

These results suggest that the Courant condition based on the subdomain dimensions is indeed a conservative criterion which can be confidently used in wave propagation problems. However, in some cases this criterion seems to be overly conservative and results in increased accuracy as the mesh partition is refined.

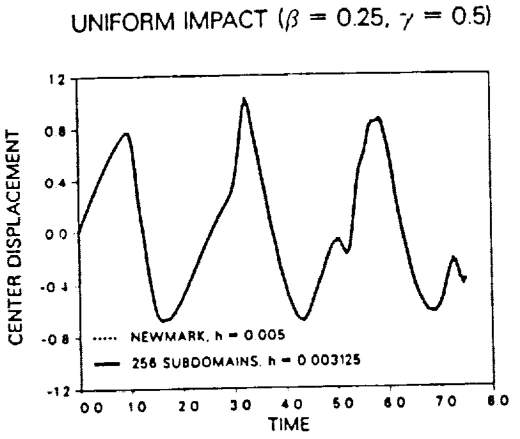


Figure 9. Uniform Impact ($\beta = 0.25,\ \gamma = 0.5$)

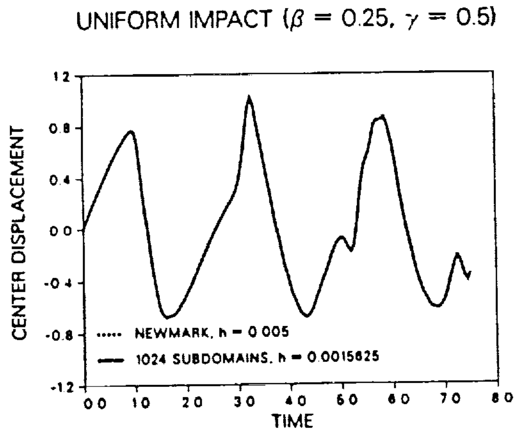


Figure 10. Uniform Impact ($\beta = 0.25,\ \gamma = 0.5$)

This bears on the fact that the Courant condition is based on a worst possible scenario, namely, a solution dominated by short wave lengths. As discussed in before, this situation exacerbates the accuracy limitations of concurrent algorithms. It is interesting to note that fully implicit algorithms such as Newmark's method do not fare particularly well either in situations dominated by high frequency modes such as shock waves. In fact, such cases are optimal for the application of explicit methods. Thus, in typical structural applications it is likely that the Courant condition can be substantially relaxed without detriment to the accuracy of the solution.

4.2.2. Stability

The stability properties of the concurrent algorithm for first order systems outlined in Section 4.1.1 were established in [1]. Here we establish the range of unconditional stability of the GI algorithm defined in Box 2. The stability properties of this algorithm are summarized in the following proposition.

Theorem. *The GI algorithm is unconditionally stable if* $\gamma \geq 1/2$, $\beta \geq \gamma/2$.

Proof.

Start by expressing the algorithm in Box 2 as

$$\tilde{\mathbf{d}}_{n+1} = \mathbf{d}_n + \Delta t \mathbf{v}_n + (1/2 - \beta)\Delta t^2 \mathbf{a}_n \tag{4.55}$$

$$\tilde{\mathbf{v}}_{n+1} = \mathbf{v}_n + (1 - \gamma)\mathbf{a}_n \tag{4.56}$$

$$\mathbf{a}_{n+1} = -\mathbf{M}^{-1}\mathbf{L}^T\bar{\mathbf{M}}(\bar{\mathbf{M}} + \beta\Delta t^2\bar{\mathbf{K}})^{-1}\bar{\mathbf{K}}\mathbf{L}\tilde{\mathbf{d}}_{n+1} \tag{4.57}$$

$$\mathbf{d}_{n+1} = \tilde{\mathbf{d}}_{n+1} + \beta\Delta t^2 \mathbf{a}_{n+1} \tag{4.58}$$

$$\mathbf{v}_{n+1} = \tilde{\mathbf{v}}_{n+1} + \gamma\Delta t \mathbf{a}_{n+1} \tag{4.59}$$

The first and last two equations are indentical to those in the predictor and corrector phases of Newmark's method. Rewrite (4.57) as

$$\mathbf{a}_{n+1} = -\mathbf{M}^{-1}\mathbf{H}(\Delta t)\tilde{\mathbf{d}}_{n+1}, \tag{4.60}$$

$$\mathbf{H}(\Delta t) = \mathbf{L}^T\bar{\mathbf{M}}(\bar{\mathbf{M}} + \beta\Delta t^2\bar{\mathbf{K}})^{-1}\bar{\mathbf{K}}\mathbf{L} \tag{4.61}$$

where $\mathbf{L}$ is the localization operator defined in Section 4.1.1. Note that the matrix $\mathbf{H}(\Delta t)$ is assembled from subdomain contributions, i. e.,

$$\mathbf{H}(\Delta t) = \mathbf{L}^T\bar{\mathbf{H}}(\Delta t)\mathbf{L}, \tag{4.62}$$

$$\bar{\mathbf{H}}(\Delta t) = \bar{\mathbf{M}}(\bar{\mathbf{M}} + \beta\Delta t^2\bar{\mathbf{K}})^{-1}\bar{\mathbf{K}} \tag{4.63}$$

Next, formulate the eigenvalue problem

$$\mathbf{H}(\Delta t)\mathbf{x} - \lambda(\Delta t)\mathbf{M}\mathbf{x} = \mathbf{0} \tag{4.64}$$

When expressed in the basis defined by the eigenvectors of (4.64), eqs. (4.55-59) reduce to

$$\tilde{d}_{n+1} = d_n + \Delta t v_n + (1/2 - \beta)\Delta t^2 a_n \tag{4.65}$$

$$\tilde{v}_{n+1} = v_n + (1 - \gamma)a_n \tag{4.66}$$

$$a_{n+1} + \lambda(\Delta t)\tilde{d}_{n+1} = 0 \tag{4.67}$$

$$d_{n+1} = \tilde{d}_{n+1} + \beta\Delta t^2 a_{n+1} \tag{4.68}$$

$$v_{n+1} = \tilde{v}_{n+1} + \gamma\Delta t a_{n+1} \tag{4.69}$$

where the scalar variables d, v and a represent the modal amplitudes of displacement, velocity and acceleration corresponding a generic eigenmode. Again, these equations are identical to the modal Newmark relations except for equation (4.67), which in Newmark's method is replaced by

$$a_{n+1} + \frac{\omega^2}{1 + \beta\omega^2\Delta t^2}\tilde{d}_{n+1} = 0 \tag{4.70}$$

where ω is the corresponding eigenfrequency. However, we can rephrase (4.67) in a form similar to (4.70) by defining a time-step dependent ficticious frequency

$$\hat{\omega}^2(\Delta t) = \frac{\lambda(\Delta t)}{1 - \beta\lambda(\Delta t)\Delta t^2} \tag{4.71}$$

in terms of which (4.67) becomes

$$a_{n+1} + \frac{\hat{\omega}^2(\Delta t)}{1 + \beta\hat{\omega}^2(\Delta t)\Delta t^2}\tilde{d}_{n+1} = 0 \tag{4.72}$$

Next we show that

$$\omega^2(\Delta t) < \infty, \qquad \forall \Delta t > 0 \tag{4.73}$$

By Iron's bounding principle [8],

$$\lambda(\Delta t) < \bar{\lambda}_{\max}(\Delta t) \tag{4.74}$$

where $\bar{\lambda}_{\max}(\Delta t)$ is the maximum eigenvalue of the extended problem

$$\bar{\mathbf{H}}(\Delta t)\bar{\mathbf{x}} - \bar{\lambda}(\Delta t)\bar{\mathbf{M}}\bar{\mathbf{x}} = \mathbf{0} \tag{4.75}$$

Using definition (4.63), this can be rewritten as

$$\bar{\mathbf{K}}\bar{\mathbf{x}} - \bar{\lambda}(\Delta t)(\bar{\mathbf{M}} + \beta\Delta t^2\bar{\mathbf{K}})\bar{\mathbf{x}} = \mathbf{0} \tag{4.76}$$

Inserting for $\mathbf{x}$ the maximum eigenmode of $\bar{\mathbf{K}}$ with respect to $\bar{\mathbf{M}}$, we obtain the relation

$$\bar{\lambda}_{\max} = \frac{\bar{\omega}^2_{\max}}{1 + \beta\bar{\omega}^2_{\max}\Delta t^2} \tag{4.77}$$

where $\hat{\omega}_{\max}$ is the maximum eigenfrequency for the extended system with stiffness $\bar{\mathbf{K}}$ and mass $\bar{\mathbf{M}}$. From (4.77) we have

$$\beta\lambda(\Delta t)\Delta t^2 < \beta\bar{\lambda}_{\max}(\Delta t)\Delta t^2 = \frac{\beta\bar{\omega}^2_{\max}\Delta t^2}{1 + \beta\bar{\omega}^2_{\max}\Delta t^2} < 1, \qquad \forall \Delta t < \infty \tag{4.78}$$

From this and (4.71), it follows that $\hat{\omega}^2(\Delta t)$ is indeed bounded for arbitrarily large Δt. The implications of this result are as follows. For a given Δt, eqs. (4.65), (4.66), (4.68), (4.69) and (4.72) are identical to Newmark's modal equations with a frequency $\hat{\omega}(\Delta t) < \infty$. Hence, the GI algorithm has the same range of unconditional stability as Newmark's method, i. e., it is unconditionally stable for $\gamma \geq 1/2$, $\beta \geq \gamma/2$.

♠

4.2.3. Operation Counts.

In this section the GI algorithms are compared with the one-way dissection method. This method is used for the solution of systems of linear algebraic equations arising in finite element applications [9]. In essence, the one-way dissection method amounts to a reordering of the elements in the model. Its main advantage is the reduction in storage requirements and in the operations needed for factorizing the matrix of coefficients of the system.

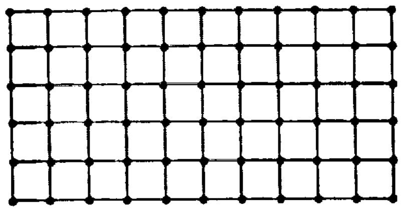


Figure 11. An m by l grid with $m = 6$ and $l = 11$, [9].

Consider a $m \times l$ grid, Figure 11, with $m \leq l$. The total number of nodes is then given by $N = lm$. For simplicity of notation, one degree of freedom per node is assumed. The one-way dissection method is based on a partitioning of the grid by σ vertical lines (*separators*), which dissect the mesh into $\sigma + 1$ independent blocks. Each "sub-mesh" is numbered row by row followed by the separators. Figure 12 shows the case $\sigma = 4$.

The leading term of the operation count for matrix factorization resulting from the one-way dissection ordering is found to be [9]

$$\theta_f^{\mathrm{OWD}}(\sigma) \sim \frac{ml^3}{2(\sigma + 1)^2} \tag{4.79}$$

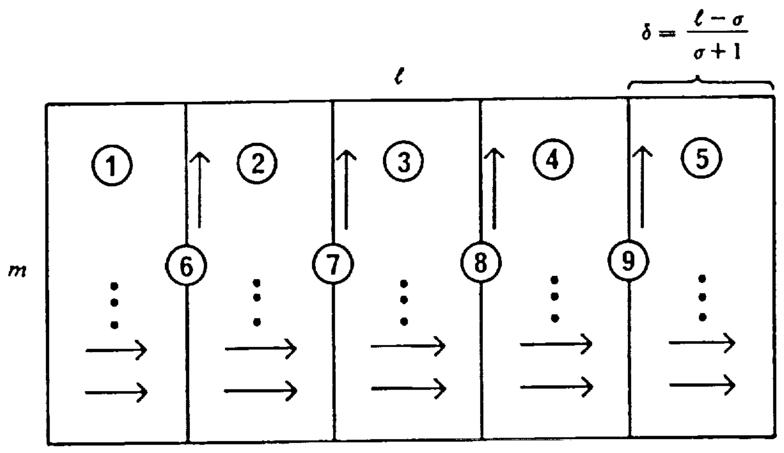


Figure 12. One way dissection ordering of an m by l grid, indicated by the arrows ($\sigma = 4$), [9].

On the other hand, the number of operations required for the factorization of a banded matrix is asymptotically [9]

$$\theta_f = \frac{1}{2} N b^2 \tag{4.80}$$

with N as defined above and b denoting the semi-bandwidth of the matrix. Each subdomain in the partition contains a $(l+1)/(\sigma+1) \times m$ grid. The bandwith of the subdomain matrices is $b = (l+1)/(\sigma+1)+1$. Thus, the factorization cost in the GI algorithm is of the order

$$(\theta_f)^{\mathrm{GI}} = 2\left[\frac{1}{2} m \left(\frac{l+1}{\sigma+1}\right)\left(\frac{l+1}{\sigma+1}+1\right)^2\right] \tag{4.81}$$

the leading term for which is

$$(\theta_f)_2^{\mathrm{GI}} \sim \frac{ml^3}{2(\sigma+1)^2} \tag{4.82}$$

It is thus concluded that, to leading order, the factorization cost of GI algorithms is of the same order than that of one-way dissection. The principles at work in both methods are, in fact, identical. In both cases, the renumbering associated with the partition or the dissection of the grid eliminates fill-in by off-diagonal zeros, thus cutting down on unnecessary zero multiplies during factorization.

4.2.4. Theoretical Speed-ups

To estimate the computational efficiency of the method, let us start by recalling that the number of operations involved in matrix factorization and forward and backward substitution is

$$FACTORIZ \approx \frac{1}{2}nb^2, \qquad SUBSTIT \approx 2nb \tag{4.83}$$

where b is the semiband width and n, as before, is the number of degrees of freedom of the structure. In typical structural applications, the cost of large scale nonlinear computations is dominated by the equation solving phase. Under these conditions, a good estimate of the computational cost is given by

$$COST \approx (FACTORIZ + SUBSTIT) \times ITER \times STEPS \tag{4.84}$$

where $ITER$ is the average number of equilibrium iterations per time step and $STEPS$ is the number of time steps required for a given duration of the analysis T, i. e., $STEPS = T/\Delta t$.

In two dimensions, consider a square mesh of l^2 elements. Then, $b = l + 2$, $n = (l+1)^2$ and, thus, a global system solution requires

$$GLOBAL \approx \frac{1}{2}(l+2)^2(l+1)^2 + 2(l+2)(l+1)^2 \tag{4.85}$$

operations. Assume now that the mesh is partitioned into $s = m^2$ subdomains. Then, the equation solving effort involved in one application of the algorithm is

$$PARTIT \approx s\left[\frac{1}{2}\left(\frac{l}{m}+2\right)^2\left(\frac{l}{m}+1\right)^2 + 2\left(\frac{l}{m}+2\right)\left(\frac{l}{m}+1\right)^2\right] \tag{4.86}$$

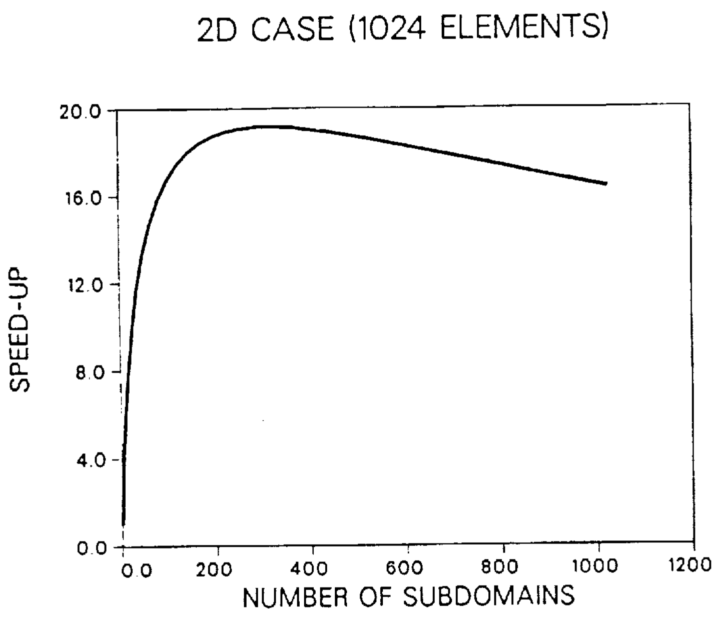


Figure 13. Reduction in number of operations required for one factorization and backsubstitution in square membrane problem as the mesh is partitioned into an increasing number of subdomains

For nontrivial partitions, this count is less than that pertaining to the global system. Thus, the net speed-up in equation solving afforded by the partitioning is given by

$$EQUATION\ SOLVING\ SPEED-UP = \frac{GLOBAL}{PARTIT} \tag{4.87}$$

The dependence of this function on the number of subdomains is shown in Figure 13. It is readily verified that a speed-up of order

$$EQUATION\ SOLVING\ SPEED-UP(2D) \approx O(s) \tag{4.88}$$

is attained asymptotically in the large scale limit $n/s \to \infty$.

The three dimensional case is amenable to an entirely similar analysis. The resulting speed-up is shown in Figure 14 as a function of the number of subdomains.

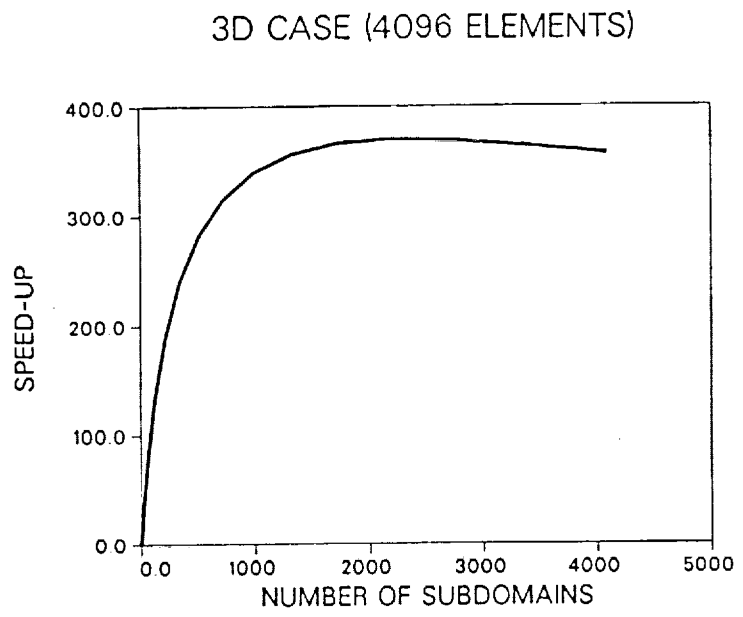


Figure 14. Reduction in number of operations required for one factorization and backsubstitution in cube problem as the mesh is partitioned into an increasing number of subdomains

Here, an asymptotic speed-up of order of

$$EQUATION\ SOLVING\ SPEED - UP(3D) \approx O(s^{4/3}) \qquad (4.89)$$

is reached in the large scale limit.

Some aspects of these estimates are noteworthy. Firstly, it is seen from Figures 13 and 14 that some efficiency is gradually lost for a given size n as the number of subdomains s is increased. This loss is due to the fact that the interface nodes need to be reduced more than once during the subdomain factorizations. On the other hand, it should be noted that these speed-ups cannot be fully realized in practice due to the fact that, in order to maintain the accuracy of the solution, the time step needs to be reduced as the number of subdomains is increased, as discussed in Section 4.2.1.

It turns out, however, that accuracy constraints offset the equation solving speed-ups only partially, and thus net gains remain. To see this, recall that the Courant condition (4.53) requires the time step to be reduced as $O(1/s^{1/2})$ in 2D, equation (4.93), and as $O(1/s^{1/3})$ in 3D, equation (4.97). This leaves a net speed-up of $O(s^{1/2})$ in 2D and $O(s)$ in 3D, which in conjunction with the $O(p)$ speed-up afforded by concurrency yields

$$\begin{aligned} NET\ SPEED-UP(2D) &= O(p\sqrt{s}), \\ NET\ SPEED-UP(3D) &= O(ps) \end{aligned} \tag{4.90}$$

It should be emphasized that these speed-up estimates involve two parameters, namely, the number of subdomains s in the partition and the number of processors p in the machine. The speed-ups represent the reduction in execution time obtained with respect to the straight application of Newmark's method ($s = 1$) on a sequential machine ($p = 1$). Factored into the estimates are three effects: a) the reduction in equation solving effort due to the the partition of the mesh; b) the linear speed-up afforded by the concurrency of the computations; and c) the gradual loss of accuracy incurred as the number of subdomains is increased. Even with this latter effect factored in, it is seen that net speed-ups result, even on one processor.

4.3. Numerical Experiments

In this section, we endeavor to assess the performance of GI algorithms by way of numerical testing. Firstly, we seek to verify the time step requirements for accuracy derived in Section 4.2.1. Our numerical simulations suggest that the Courant-like condition (4.53) is indeed of value in actual 2D and 3D applications. Next, we compute the actual speed-ups afforded by the GI algorithm as the number of subdomains and processors is increased, while choosing the time step so as to maintain a constant level of accuracy in all cases. Finally, some simulations are shown which demonstrate the high communication efficiency of the method.

4.3.1. Actual Time Step Required in 2D

In Section 4.2.1, it was demonstrated how accuracy considerations place limits on the size of the time step which become increasingly stringent as the partition is refined. Here we seek to determine the exact time steps requirements in an actual application. As a two dimensional example, the problem of an elastic membrane undergoing finite deflections is considered. This analysis is representative of structural computations in that the element operations are relatively inexpensive, so that the cost of the analysis is dominated by equation solving. The purpose of the simulation is to determine the actual time step required to maintain a prescribed level of accuracy as the number of subdomains is increased.

The element utilized in the calculations is a four node quadrilateral obtained by averaging two triangular assemblies, each splitting the quadrilateral along one of its diagonals. The constituent triangular elements are endowed with a strain energy of the form

$$W = \frac{T}{2}\frac{A^2}{A_0} \tag{4.91}$$

where T is the tension of the membrane, and A and A_0 are the areas of the deformed and undeformed triangles. It is easily checked that this formulation reduces to the usual small deflection theory of membranes when $A \approx A_0$.

The membrane in the analysis is taken to be square and to be simply supported all around its perimeter. The values of the material parameters adopted are $T = 1$ and a mass density $\rho = 1$. Initially, the membrane is supposed to lie in its undeformed configuration, and to be subjected to blast loading resulting in a uniform initial velocity throughout its surface. The magnitude of the prescribed initial

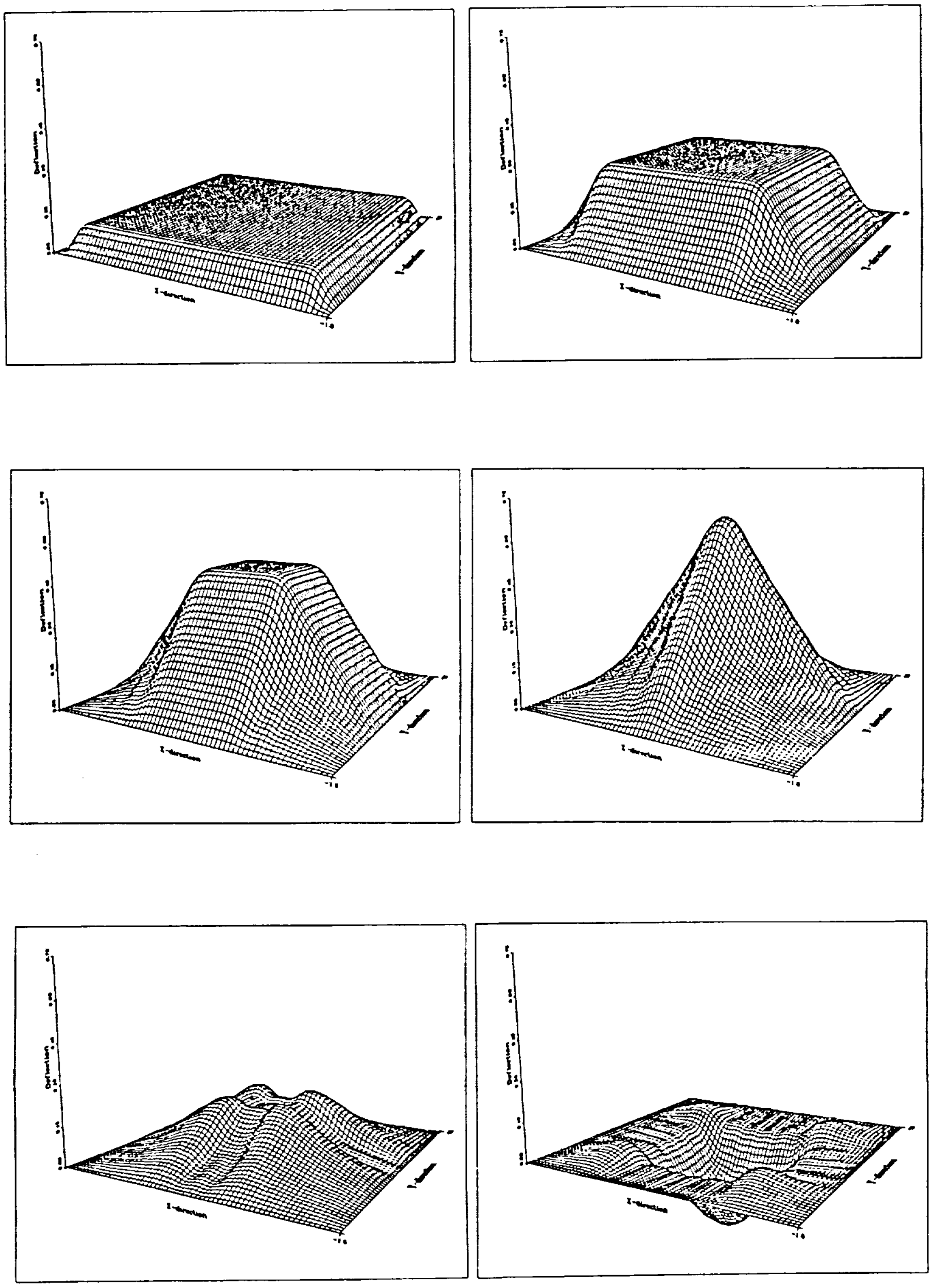


Figure 15. Deflected shapes.

velocity, $v_0 = 1$, is enough to generate strains of the order of 30% and rotations of the order of 45°. The half size of the membrane is taken to be $L = 1$. By virtue of the symmetries of the solution, the analysis may be restricted to one quarter of the membrane. Throughout the computations, this domain is discretized into a 64×64 regular mesh. The deflected shapes of the membrane at various stages of the solution are shown in Figure 15.

The partitions adopted in the calculations divide the domain of the analysis into equal square subdomains. Let m be the number of subdomains per side. Then, the number of subdomains in the partition is $s = m^2$. Clearly, with increasing m the size of the subdomains diminishes according to

$$\Delta L = L/m \tag{4.92}$$

Hence, the Courant condition (4.53) necessitates a steady reduction of the time step of the order

$$\Delta t \approx \Delta t_0/m = \Delta t_0/\sqrt{s} \tag{4.93}$$

for the accuracy of the calculations to remain unchanged under increasing refinement of the partition. Δt_0 denotes a choice of time step appropriate for Newmark's algorithm, i. e., for the case $s = 1$. It is seen that, according to estimate (4.93), the required time step is a decreasing function of the number of subdomains. In Section 4.3.3 it is shown that this effect is amply offset by the reduction in equation solving effort afforded by the method. Thus, a net gain in efficiency remains over Newmark's algorithm, even on a single processor.

The membrane calculations are carried out for an increasing number of subdomains, with several time steps around the theoretical value (4.93). The time step adopted for Newmark's algorithm is $\Delta t = 0.05$. Figure 16 depicts the time history

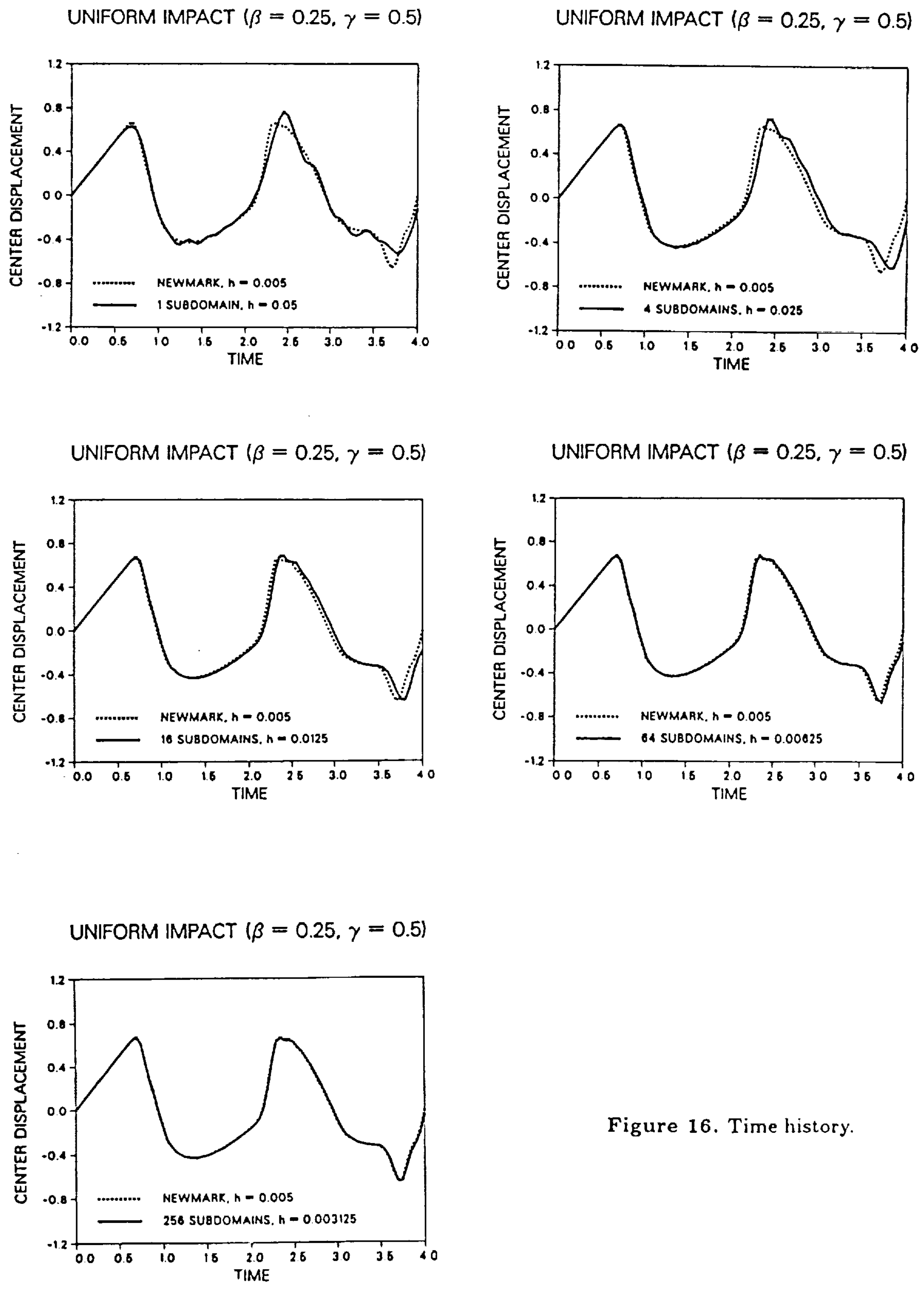


Figure 16. Time history.

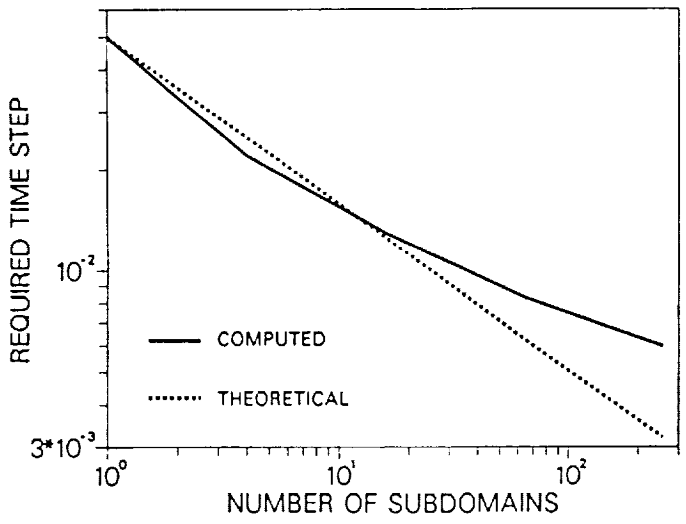


Figure 17. Estimate of actual time step, 2D.

of center deflection computed from various partitions of the mesh. The error in the solution is then computed as

$$ERROR^2 = \int_0^T \mid w(t) - w_{exact}(t) \mid^2 \frac{dt}{t^2} \tag{4.94}$$

where $w(t)$ and $w_{exact}(t)$ are the computed and exact center deflections, respectively. In lieu of an exact solution, the results from Newmark's method with a small time step ($\Delta t = 0.005$) are utilized. The above definition of the error provides a measure of the period elongation in the computed solution. In particular, it can be shown that

$$\lim_{T\to\infty} \left[\int_0^T \mid sin(\omega t) - sin((\omega + \Delta\omega)t) \mid^2 \frac{dt}{t^2} \right]^{1/2} \propto \Delta\omega \tag{4.95}$$

For each partition of the mesh, the calculations are repeated for several time steps around the theoretical estimate (4.93), and the error measure (4.94) computed. Then, by linear interpolation, the actual time step is computed to maintain the level of error resulting from Newmark's method. The results are shown in Figure 17, together with the estimate (4.93). As may be seen, the theoretical accuracy requirements are realized quite closely.

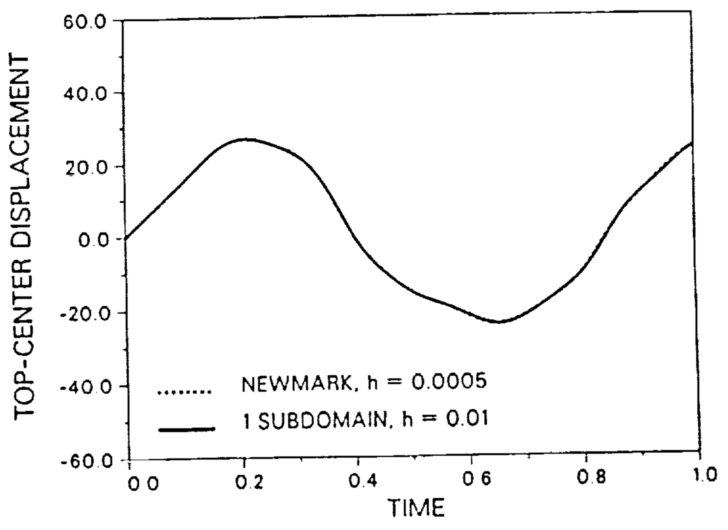
4.3.2. Actual Time Step Required in 3D

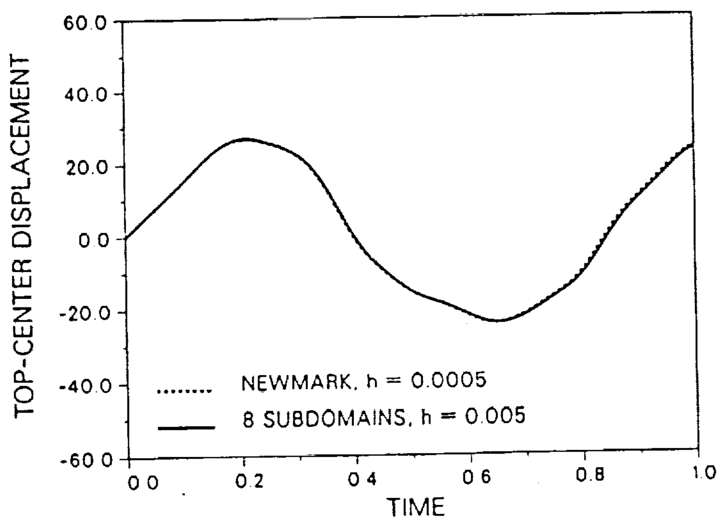
Here we repeat the analysis of Section 4.3.1 for a three-dimensional problem. We consider the case of an elastic cube supported on a rigid foundation and undergoing finite deformations. The material behavior is characterized by the simple stress-strain relation

$$S_{IJ} = \lambda E_{KK}\delta_{IJ} + 2\mu E_{IJ} \tag{4.96}$$

where S_{IJ} and E_{IJ} are the components of the second Piola-Kirchhoff stress tensor and the Lagrangean strain tensor, respectively, and λ and μ are Lame-type constants. The material parameters used in the calculations are $\lambda = 8 \times 10^9$ and $\mu = 8 \times 10^7$, which results in nearly incompressible behavior. The mass density of the material is taken to be $\rho = 200$. The dimensions of the cube are $L = 100$. The body is loaded by means of uniform velocities $v_1 = 150$, $v_2 = 300$ suddenly applied on the foundation in the directions of the sides of the cube. The magnitude of the velocities suffices to produce strains of the order of 30%. The cube is discretized into 64 brick elements. The method used to avoid mesh locking due to near-incompressibility is described in [10].

Following the application of the initial velocities, the cube undergoes a sloshing motion. In the linear range, the low frequency modes of this response are dominated by the shear response of the solid. Our choice of parameters is intended to underscore the benefits derived from unconditional stability. Thus, whereas the response of interest lies mainly in the low frequency part of the spectrum, explicit





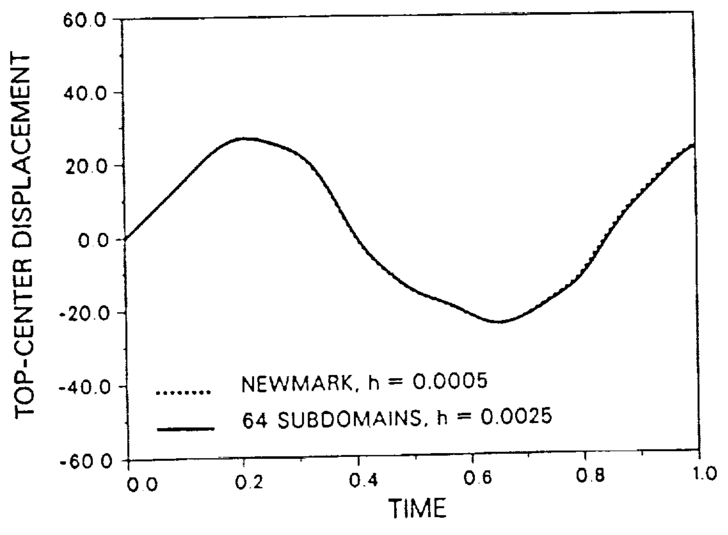


Figure 18. Horizontal displacement at the center node.

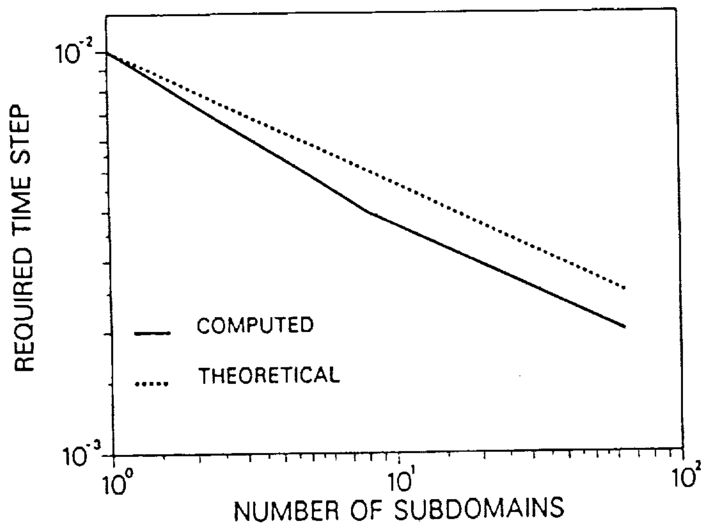


Figure 19. Estimate of actual time step, 3D.

algorithms are required to resolve the high frequency modes corresponding to the volumetric response, which in the problem under consideration necessitates the use of impractically small time steps.

As in the two dimensional simulation, the mesh is partitioned into a varying number of cubic subdomains. If m is the number of subdomains per edge of the cube, so that $s = m^3$, the Courant condition (4.53) then demands that

$$\Delta t \approx \Delta t_0 / m = \Delta t_0 / s^{1/3} \tag{4.97}$$

To determine the actual time step requirements, the accuracy of the computations is monitored at the uppermost center node of the cube. The histories of one of the horizontal displacements at this location are shown in Figure 18 for the various mesh partitions used in the computations. Figure 19 shows the time step requirements resulting from the accuracy analysis. As in the two dimensional case, the actual time step requirements conform closely to the theoretical estimate (4.53).

4.3.3. Performance Assessment

From the theoretical estimates derived in Section 4.2.4, it is evident that the execution times are primarily dependent on two variables: the number of processors p and the number of subdomains s in the partition. Figure 20 shows the execution times for the membrane problem described in Section 4.2.1 as a function of the algorithmic parameters p and s. The calculations are run with the actual time step required to maintain the level of accuracy as the number of subdomains is varied. For the test problem under consideration, these time step requirements are given in Section 4.2.1. Thus, the execution times being compared correspond to solutions of comparable accuracy.

It should be noted that Figure 20 gives equation solving times only. For typical large scale nonlinear structural problems, the execution times are indeed dominated by the equation solving phase of the computations. A main motivation for reporting these data, however, is that equation solving, unlike other aspects of finite element computations, is a fairly standardized procedure. This renders comparisons of data from different codes more straightforward. In the calculations reported here, we have used Taylor's variable bandwidth implementation of Crout's method [11].

It is seen from Figure 20 that, for a fixed number of subdomains, the speed-ups obtained are roughly linear in the number of processors p. The proportionality factor between speed-up and p is a measure of the efficiency of the computations, and is investigated in the next section. For a fixed number of processors, a net

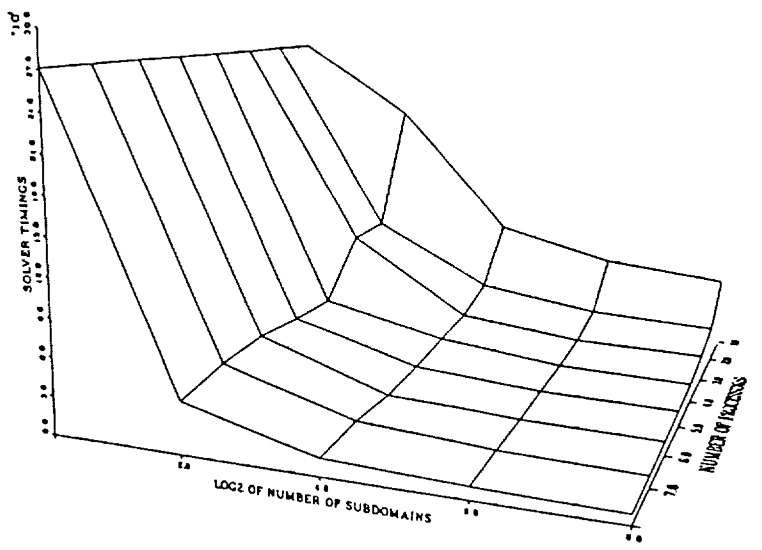


Figure 20. Execution times for the membrane as a function of the number of processors and the number of subdomains.

speed-up is obtained as the number of subdomains is increased. The single processor case, $p = 1$, is shown in Table 1. As may be seen, the observed speed-ups lag behind the $O(\sqrt{s})$ asymptotic estimate. This is not unexpected since such estimate is only realized in the large scale limit $n/s \to \infty$. Despite the relatively small size of the test problem under consideration, the net gains afforded by the algorithm may be quite substantial even on one processor. For instance, for 256 subdomains an almost five fold speed-up is obtained over Newmark's method.

Figure 20 also illustrates the synergism between concurrency and refinement of the partition, i. e., the fact that the corresponding speed-ups combine multiplicatively, rather than additively. For the case $p = 8$, the maximum number of processors in the FX8, and 256 subdomains, the net equation solving speed-up is of the order of 33.7, a rather formidable performance enhancement. By comparison, parallel solvers result in speed-ups which are, at best, linear in the number

C-2

of processors. In the present method, by contrast, an additional speed-up (asymptotically $O(\sqrt{s})$ in 2D, $O(s)$ in 3D) is introduced by the partitioning of the mesh which provides an additional edge over parallel solvers.

TABLE 1.- Equation solving timings on one processor. Membrane example.

4096 ELEMENT CASE			
s	Seconds	Speed-up	Asymp.
1	27100	1	1
4	20515	1.32	2
16	9529	2.84	4
64	7006	3.87	8
256	5898	4.59	16

4.3.4. Computational Efficiency on the CalTech Hypercube

Another important performance measure is the fraction of time the processors are actually kept busy, i. e., the efficiency of the computations. Overhead due to extensive interprocessor communication has a negative effect on computational efficiency. The minimization of the extent of data transfer between processors thus becomes a principal concern in algorithm design. For the *GI* algorithms considered here, the exchange of information between processors is reduced to the transfer of one linear array per time step. Thus, interprocessor communications are kept to a minimum. An illustration of the high performance of the algorithm is given in [12], where actual simulations on the CalTech/JPL Mark III hypercube machine are presented. This computer consists of 32 (2^5) processors (or nodes), configured as a 5-dimensional hypercube.

The GI algorithm was implemented within a finite element program on the Mark III hypercube and used to analyze the plane stress model of a cantilever beam with a tip load, Figure 21.

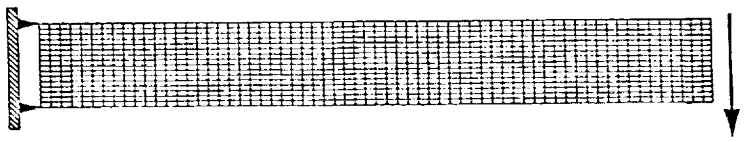


Figure 21. Cantilever beam, $neq = 2142$, $nel = 1024$

The purpose of this study is to evaluate the efficiency of the algorithm on a local memory architecture. Computational efficiency is here defined as

$$e = \frac{T_1}{pT_p} \tag{4.98}$$

where T_p is the time for performing the analysis on p processors, and T_1 is the time for an identical analysis on a single processor. In a typical run all the processors begin simultaneously, but end their tasks at different times. T_p is then the time for the slowest processor. This time difference is due to two effects. The first is a load imbalance whereby some processors may have a larger task (in our case more elements to process). The second arises when some processors have more information to send/receive than others.

The beam is discretized into a 16×64 regular mesh of plane stress elements, Figure 21. The mesh is then partitioned into 4,8,16, and 32 identical sub-structures. All the separators (partition lines) are through the thickness (vertical). Since all processors are assigned the same number of elements, these partitions ensure optimum load balancing and are chosen to illustrate the performance of the algorithm. As a result of this partitioning scheme all processors will have the same computation time. Thus, the difference between T_1/p and T_p is the required communication time and for this problem the efficiency becomes

$$e = 1 - \frac{T_{comm}(p)}{T_{calc}(p)} \tag{4.99}$$

where T_{comm} and T_{calc} are the maximum communication time and calculation time, respectively. Note that the above choice of partitioning results in the same number of interface nodes on all processors (with the exception of the domains at each end) and thus the same communication time.

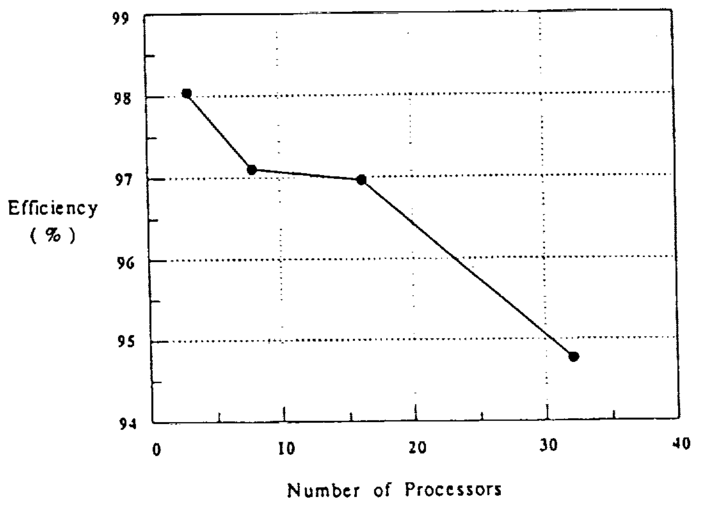


Figure 22. Efficiency Results for the beam problem.

Figure 22 gives a plot of the measured efficiency rate for an increasing number of processors. In all cases, an efficiency of well over 90% is observed. Since the mesh is partitioned vertically, the number of interface nodes between subdomains, and thus the communication time among processors, does not change with the number of processors. However, as the number of processors increases, the number of elements per processor decreases. As a result, the calculation time drops resulting in a reduction in the efficiency rates. When 32 processors are used, the subdomains are small (32 elements per processors). In larger problems, the efficiency rates are expected to improve further. Note that when going from 16 to 32 processors, one could choose the neutral axis of the beam as a partitioning line. This in turn would reduce the communication overhead and thus increase efficiencies over those shown in Figure 22.

References

1. M. Ortiz and B. Nour-Omid, 'Unconditionally stable concurrent procedures for transient finite element analysis,' *Comp. Meth. Appl. Mech. Engng.*, **58**, 1986, pp.151-174.

2. Nour-Omid and Ortiz, 'A Family of Concurrent Algorithms for Transient finite Element Solutions', *Proceedings of ASCE Structures Congress on Parallel Processing and Computational Strategies in Structural engineering*, May 1-5, 1989.

3. M. Ortiz, P. M. Pinsky and R. L. Taylor, 'Unconditionally Stable Element-by-Element Algorithms for Dynamic Problems,' *Computer Methods in Applied Mechanics and Engineering*, Vol. 36, No. 2, 1983, pp. 223-239

4. M. Ortiz, E. D. Sotelino, and B. Nour-Omid, 'Efficiency of Group Implicit Concurrent Algorithms for Transient Finite Element Analysis,' *International Journal for Numerical Methods in Engineering*, to appear.

5. C. W. Gear and L. R. Petzold, 'ODE Methods for the Solution of Systems,' *SIAM, Journal on Analysis*, Vol. 21, No. 4, Aug 1984, pp. 716-728.

6. N. M. Newmark, 'A Method of Computation for Structural Dynamics,' *Journal of the Engineering Mechanics Division*, ASCE, 1959, pp. 67-94.

7. R. Mullen and T. Belytchko, 'An analysis of an unconditionally stable explicit method, *Computers and Structures*, **16**, 1983, pp. 691-696.

8. B. M. Irons, 'Applications of a Theorem on Eigenvalues to Finite Element Problems,' (CR/132/70), University of Wales, Department of Civil Engineering, Swansea, 1970.

9. A. George and J. W. Liu, 'Computer Solution of Large Sparse Positive Definite Systems', *Prentice Hall Series in Computational Mathematics*, 1981.

10. M. Ortiz, 'Some Computational Aspects of Finite Deformation Plasticity,' in: D.R.J. Owen, E. Hinton and E. Oñate (eds.), *Computational Plasticity*, Pineridge Press, Swansea, 1987, pp. 1717-1756.

11. R. L. Taylor, 'Computer Procedures for Finite Element Analysis,' in: O. C. Zienkiewicz, *The Finite Element Method*, 3rd edition, McGraw-Hill, 1977, Chapter 24.

12. M. Ortiz, B. Nour-Omid, and Elisa D. Sotelino, 'Accuracy of a Class of Concurrent Algorithms for Transient Finite Element Analysis,' *International Journal for Numerical Methods in Engineering*, Vol. 26, 1988, pp. 379-391.