NASA Contractor Report 187162

Probabilistic Structural Mechanics Research for Parallel Processing Computers

Robert H. Sues, Heh-Chyun Chen,
and Lawrence A. Twisdale
Applied Research Associate, Inc.
Raleigh, North Carolina

and

William R. Martin
University of Michigan
Ann Arbor, Michigan

August 1991

Prepared for
Lewis Research Center
Under Contract NAS3-25824

National Aeronautics and
Space Administration

TABLE OF CONTENTS

TABLE OF CONTENTS
(continued)

TABLE OF CONTENTS
(continued)

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES
(continued)

CHAPTER 1

INTRODUCTION

1.1 BACKGROUND

Aerospace structures and spacecraft are a complex assemblage of structural components that are subjected to a variety of complex, cyclic, and transient loading conditions. Significant modeling uncertainties are present in these structures, in addition to the inherent randomness of material properties and loads. To properly account for these uncertainties in evaluating and assessing the reliability of these components and structures, probabilistic structural mechanics (PSM) procedures must be used.

Significant advances have taken place in PSM over the last two decades. Much of this research has focused on basic theory development and the development of approximate analytic solution methods in random vibrations and structural reliability. Practical application of PSM methods has been hampered by their computationally intense nature. Solution of PSM problems require repeated analyses of structures that are often large, and exhibit nonlinear and/or dynamic response behavior. A single deterministic solution of such structures can strain available computational resources. These methods, however, are all inherently parallel and ideally suited to implementation on parallel processing computers. While there has been research into parallel implementation of Monte-Carlo methods in physics and nuclear engineering (see Table 1-1), no research has been conducted in PSM. A need exists to systematically study implementation of these methods on parallel architectures and identify the optimal hardware and software specifications. New hardware architectures and innovative control software and solution methodologies are needed to make solution of large scale PSM problems practical. The next decade of research in computational PSM and the promise of parallel computing may open up a whole new class of nonlinear finite element and dynamics problems to probabilistic structural analysis.

1.2 OBJECTIVES AND SCOPE

The ultimate goal of this research program is to develop an integrated system that can achieve significant reductions in computational times for large scale PSM problems. This system will exploit the inherently parallel nature of PSM problems by incorporating new and innovative parallel hardware architectures (based on current technologies), controlling software, and solution strategies.

Achieving large scale parallelism and significant reductions in computer time will require overcoming limitations of current parallel architectures and developing software strategies that can keep large numbers of processors busy while minimizing memory requirements and inter-processor communication. The purpose of this

Table 1-1. Examples of Recent Research in Implementation of Monte Carlo Methods in Parallel Computing Environments

Authors	Date	Application	Country	Computational Emphasis	Relevant Hardware
Brown,Martin	1984	Nuclear Engr.	USA	Vectorization	Cyber 205
Chauvet	1985	Nuclear Engr.	France	Vectorization Concurrency	Cray Cyber 205
Delves	1985	Physics	UK	Concurrency	DAP
Bhavsar, Isaac	1987	General	Canada	Concurrency	General
Burns, Pryor	1987	Heat Transfer	USA	Vectorization	Cyber 205
Glendinning, Hey	1987	Physics	UK	Concurrency	INMOS Multi-Transputer
Hey, Ward	1987	Nuclear Engr. General	UK	Concurrency	INMOS Multi-Transputer
Martin, Brown	1987	Nuclear Engr.	USA	Vectorization	Cray Cyber 205
Martin, Wan, Abdel-Rahman, Mudge	1987	Nuclear Engr.	USA	Concurrency	IBM 3090/400 NCUBE Hypercube
Miura	1987	Physics	Japan	Vectorization	Amdahl 1200
Moatti, Goldberg Memmi	1987	Physics	Israel	Concurrency	Network of Microcomputers
VanRensburg	1987	Physics	UK	Concurrency	General
Wansleben	1987	Physics	USA	Vectorization	Cyber 205
Yokozawa, Oku, Kondo, Togo	1987	Nuclear Engr.	Japan	Concurrency Vectorization	HITAC S810
Traynor, Anderson	1988	Chemistry	USA	Concurrency	General
Mori, Tsuda	1988	Physics	Japan	Vectorization	HITAC S810
Malaguti	1988	Physics	Italy	Vectorization	Cray
Vohwinkel	1988	General	USA	Vectorization	Cyber 205

Phase I research effort is to take the first steps toward parallel implementation. Toward this end we have focused herein on the basic issues of parallel implementation and have aimed to identify the special software and hardware research and development needs for large scale parallel PSM implementation.

The specific objectives of this Phase I study are:

1. Assess the current state-of-the-art of parallel processing and the adequacy of currently available technologies, architectures, and software for parallel implementation of PSM.

2. Identify the sources and multiple levels of parallelism in PSM computations that can be exploited on parallel processing computers.

3. Implement a PSM code on a parallel computer and execute fundamental example problems to identify specific implementation issues.

4. Formulate recommendations and generic specifications for development of the parallel PSM hardware and software system.

The report is organized around the basic tasks conducted to achieve these objectives. Chapter 2 contains an in-depth review of current parallel architectures and also discusses emerging technologies that are expected to impact parallel PSM. This chapter provides the basis for development of architectural concepts for the parallel PSM system. In Chapter 3 we review the sources of parallelism in PSM computations. Parallelism in both the probabilistic and structural mechanics computations are covered. Chapter 4 presents an overview of MCPAP, a parallel PSM code developed and implemented under this effort. Several example applications are also presented, including two stochastic finite element problems. Finally, in Chapter 5 we present the conclusions of our work and present generic hardware and software specifications for the integrated parallel PSM system.

An appendix is also provided that describes a numerical technique for solving systems of equations, that was developed and implemented in MCPAP. This method, the Stochastic Pre-Conditioned Conjugate Gradient method, shows excellent potential for dramatic reductions in storage requirements and computational effort in PSM problems. Both of these are critical issues for successful large scale parallel PSM implementation.

CHAPTER 2

REVIEW OF PARALLEL PROCESSING

2.1. HISTORY OF PARALLEL PROCESSING

Parallel processing seeks to improve the speed with which a computational task can be done by breaking it into subtasks and executing as many as possible of these subtasks simultaneously. This idea has a long history in computer science, but has received greater attention recently with the advent of affordable parallel computers. This section presents a brief summary of the development of the principal ideas in parallel processing (for a more detailed discussion the reader is referred to Hockney and Jesshope [1981]). Recent developments in parallel processing have led to the current situation where parallel processing can be considered to be a useful tool for many realistic applications, in particular Monte Carlo methods.

The principal means to instill parallelism into a computer architecture are:

- Pipelining (instructions as well as arithmetic operations) and
- Concurrency

Pipelining refers to the processing of data in an assembly line fashion, the concept now widely employed in vector processing computers described later. Concurrency refers to the simultaneous operation of multiple independent processors. Both concepts are of importance in parallel implementation of probabilistic structural mechanics problems.

Each of these approaches has been utilized for a number of years (decades in some cases). The earliest reference to parallel processing from the standpoint of actual computers is by Menabrea [1842], who observed that Charles Babbage's analytical engine could be used to great advantage if it were designed to perform several calculations simultaneously, thus reducing the time spent to perform the entire calculation. In addition, Babbage recognized the need to utilize "bit-parallel" arithmetic in his difference machine since serial arithmetic would have been too slow. Thus the advantages of parallel processing were identified over a century before technology had progressed to the point where it could be implemented into real hardware.

Over 100 years later, desk calculators in the 1940's exhibited some parallelism, due to the use of bit-parallel arithmetic to process approximately 12 decimal digits at the same time. Multiple functional units were introduced into computer designs quite early in the 1950's, with machines such as the IBM 704, which was modified in the latter part of that decade to include parallel I/O, and renamed the IBM 709.

Independent processors were apparently first suggested by Holland [1959], who described an assembly of processors each obeying its own instruction stream. This reference appears to be the first mention of the multiple instruction stream, multiple data stream (MIMD) class of concurrent processors, which will be discussed in more detail in a following section.

Beginning with the CDC 6600 and continuing with the CDC 7600 and IBM "Stretch" computers, functional parallelism and instruction pipelining were essential characteristics of high performance computers in the 1960's and early 1970's. Memory interleaving was also being introduced at this time, which can be described as accessing memory in parallel (perhaps memory "pipelining" might be a better description). Pipelining of the functional units was also incorporated into these machines, hence the first implementation of "vector processing" into production computers began with the CDC 6600-7600 series. However, these were still basically scalar units, and did not have features explicitly incorporated to take advantage of vector processing on a large scale. This was remedied by Seymour Cray with the introduction of the Cray-1 in 1976 at Los Alamos. Also, the CDC Star 100 and the Texas Instruments TIASC were both pipelined vector computers with earlier origins than the Cray-1. Neither of these machines, however, was a commercial success, due to relatively slow scalar units which extracted a large performance penalty, as will be noted in the section on efficiency below. The Cray-1 had an extremely fast scalar processor and this coupled with its vector processing ability made it an immediate success for large scale computation.

The notion of array processors was first mentioned in the context of a report for the "Solomon" concept [Slotnick, *et al.*, 1962], which was the conceptual seed for the class of computers known as single instruction stream, multiple data stream (SIMD), which is discussed below. The first examples to be built were the Illiac-IV, Burroughs PEPE, Goodyear STARAN (and later the MPP), and ICL DAP computers. Similar but more specialized hardware known as attached array processors were introduced by Floating Point Systems in the late 1970's with their FPS-120 series. These were attached arithmetic processors under the control of the host processor and were not technically general processing elements, but the functional characteristics of these machines are similar to a SIMD processor.

The decade of the 1980's has seen remarkable increases in computer performance, primarily due to advances in architecture rather than raw hardware gains. This will be discussed in more detail in the following sections on vector and concurrent processing.

2.2. CLASSIFICATION OF COMPUTER ARCHITECTURES

There have been several attempts to classify computer architectures, or create a taxonomy for them, but the field is sufficiently dynamic that new architectures which defy existing classifications continue to be created. In the general scheme of Flynn [1966], computers are classified as follows:

- SISD - single instruction stream, single data stream
- SIMD - single instruction stream, multiple data stream
- MISD - multiple instruction stream, single data stream
- MIMD - multiple instruction stream, multiple data stream

This scheme has made it into the general lexicon of computer science, however, difficulties arise since conceptually different architectures may fall into the same category and new machines may actually represent hybrids of more than one category. Also, the scheme does not make the important distinction, for concurrent processing computers, between shared memory and distributed memory.

For purposes of the discussions that follow, we will use Flynn's scheme in conjunction with memory architecture to assist in the descriptions. Our interest here is to examine existing architectures in order to identify those architectural characteristics that will be best suited for PSM problems. We will first examine some implementation and performance aspects of vector processing and concurrent processing machines. This will be followed by a survey of current computer architectures.

2.3. VECTOR PROCESSING

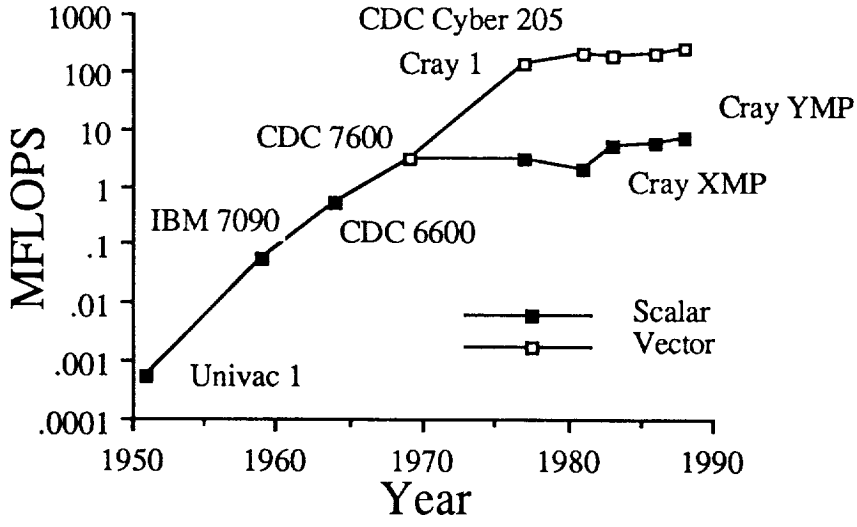
2.3.1 Description

The utilization of vector architectures in modern day supercomputers is well-established, beginning with the Cray-1 in 1977 and continuing into the foreseeable future with the Cray products as well as the IBM 3090, ETA-10, and various Japanese products, including Hitachi, NEC, and Fujitsu. The basis idea of a vector processor is that it is based on an assembly line concept -- the basic functional units (*e.g.*, add, multiply, divide) are *segmented* into many smaller units, each of which performs a very simple task. Since each task is simple, it can be done very fast, hence the clock speed can be increased to allow data to stream through the segmented unit faster than for a stand alone functional unit. This is a direct analog of Henry Ford's assembly line, with data streaming through the "factory" (the segmented functional unit), having many simple things done to it, and then exiting the assembly line at a rate which is constrained by the slowest internal function. It takes some time for a single datum to make it through the unit, but once it does, it is followed by successive data at a very fast rate. This is known as pipelining, and the idea is to amortize the time it takes to traverse the pipeline (startup time) by processing many data following the first data at a very fast rate (streaming rate). In general, it takes longer for one data to traverse the pipeline for a segmented unit (or *vector* unit) than for a conventional functional unit (*scalar* unit), and the performance depends on having a reasonably long vector of data to be processed. Depending on the

specific architecture, the number of elements in the vector to *break even*, that is the average processing speed per data element for a vector of data versus the speed for a scalar unit to process one data, may be very large. For example, for the Cray-1, the break even vector length is in the range of 7 to 10, while for the CDC Cyber 205, it is in the range of 500-1000. These differences are very important when developing algorithms for execution on these processors.

2.3.2 Impact on Performance

Figure 2-1 is a plot of computer performance measured in millions of floating point operations per second (MFLOPS) as a function of time. This figure clearly indicates that this improvement in computer performance has been due not only to advances in hardware but also to innovations in architecture, that is, how the computer is designed and organized to carry out its computational tasks. This is seen by noting the difference in the two speeds (reflecting scalar and vector performance) plotted after the CDC 7600 in 1969. The scalar speed corresponds to the speed of the conventional (scalar) CPU, whereas the vector speed is indicative of the speed attained by taking advantage of the principal innovation for large-scale computation in the 1970's and 1980's -- vector (pipelined) architectures.



Sources -

- Univac 1 and IBM 7090 - [Hockney and Jesshope, 1981]
- CDC 6600,7600 and vector data for Cray X-MP and CDC Cyber-205 - [Dongarra, 1986]
- Scalar data for Cray X-MP (8.5 ns and 9.5 ns) and CDC Cyber-205 - [Bucher and Simmons, 1985]
- Data for Cray Y-MP scaled (x1.3) from Cray X-MP

Figure 2-1. Computer Performance Versus Time

It is instructive to note that after 1970 nearly all of the increase in performance can be attributed to the vector processing capabilities of the various machines. The data in Figure 2-1 represent *actual published timing results* rather than vendor advertised peak rates, which are generally not a reliable measure of performance in realistic scientific computations. It should be noted that even this published data must be treated with caution, especially for vector calculations, because it has been obtained from relatively simple kernels and may not be representative of performance in practical applications.

2.3.3 Efficiency of Vector Processing

The task of developing or adapting an algorithm for a vector CPU is known as *vectorization*, and is essential for realizing the full potential performance of a vector CPU. If the algorithm is not vectorized, then the vector CPUs will not be utilized and there is a good chance that the overall performance will actually be worse. The effort required in modifying a scalar algorithm to run efficiently on a vector machine can be quite substantial. On the other hand, there has been substantial progress in developing optimizing compilers for vector processors in the 12 years since the Cray-1 was introduced. Today, for most intensive computational applications in science and engineering, the vectorized algorithms are well-understood and the major impediment is the implementation, or retrofitting, of old production codes originally developed for scalar processors, onto vector processors. Here the drawbacks of not following the software technology curve (upward compatible for conventional architectures) are apparent - only a fraction of the performance of a Cray (or other vector processor) can be obtained if the code is not vectorized.

The efficiency of vector processing is easily seen by a simple analysis. Consider a computer with two processing modes -- a "fast" mode and a "slow" mode. Let us define k as the ratio of the fast processing speed to the slow processing speed,

$$k = \frac{v_{fast}}{v_{slow}} \tag{2-1}$$

where for example $k = 10$ is typical of a Cray or $k = 3$ for the IBM 3090. Now define W as the total workload to be performed on the computer, α to be the fraction of the workload that is performed with the "slow" mode, and S to be the overall speedup,

$$S = \frac{\text{CPU time for slow mode only}}{\text{CPU time (both fast and slow modes)}} \tag{2-2}$$

or,

$$S = \frac{W/v_{slow}}{\alpha\ W/v_{slow} + (1 - \alpha)\ W/v_{fast}} \tag{2-3}$$

Cancelling terms yields the following expression for the speedup, sometimes known as *Amdahl's Law* [Amdahl, 1967] :

$$S = \frac{1}{\alpha + (1 - \alpha)/k} \tag{2-4}$$

It is illuminating to note that if the speed ratio, k, is *infinite* and only 50% of the workload is done in the "fast" mode, then the speedup is only two, reflecting the fact that half of the workload is still being done with the slow mode. For a vector processor, α can be viewed as the *scalar* fraction (and $1-\alpha$ the *vectorization* fraction) of the workload that is done on the vector processor. This simple expression can also be used for a concurrent processor, where one would normally use $k = N$, the number of processors, and α would represent the fraction of the workload that could be done on only one processor (and α would then be the *serial fraction*). Figure 2-2 plots the speedup versus vectorization fraction for k=10, typical of the Cray. Although plots for other values of k are not shown, suffice to say that the curves are not sensitive to k until vectorization fractions greater than 90% are reached, which are difficult to attain in practice.

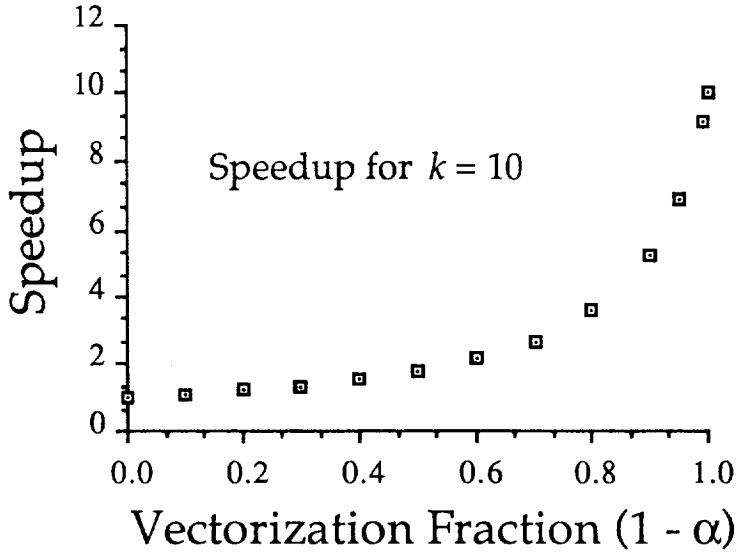


Figure 2-2. Speedup versus Vectorization Fraction for $\alpha = 10$

As an illustration, consider a code that is approximately 20% vectorized by the compiler. This yields a speedup of only 20%. For 50% vectorization, experience has shown that minor restructuring by a programmer will be needed, and the speedup of the calculation is still less than a factor of 2. To approach the 80% level will

require the attention of the methods developer, and may require substantial changes to the overall algorithm. As can be seen, the asymptotic speedup of ten (typical for a Cray) is not even approached until 90% vectorization. Unfortunately, it appears that this "law" has been ignored by more than one vendor in recent years.

In conclusion, it seems apparent that vector processors are well-understood and modern compilers do a relatively good job of extracting vectorization from a typical code. However, to obtain optimum performance, it is still necessary to structure the algorithm to take advantage of the vector architecture. For many engineering applications, this has been done, at least in principle. One reason for this state of affairs is that a vector processor is still a von Neumann architecture, in that computations are done serially. This avoids concurrent processing, which is, in general, recognized to be a much more difficult challenge to algorithm designers and programmers than vector processing. Fortunately, as we will demonstrate later, probabilistic structural mechanics algorithms are inherently parallel, and we will be able to take advantage of this.

2.4. CONCURRENT PROCESSING

2.4.1 Description

We will distinguish concurrent processors by being either distributed-memory or shared-memory. Distributed-memory parallel processors are typically regular arrays of large numbers of processors each with their own local memory. These processors are interconnected by communication links that can be used for inter-processor communication. Two processors can communicate by passing messages along a path of links that has the processors as end points (the messages may pass through intervening processors). Distributed-memory machines are also referred to as "message passing" or "loosely coupled" architectures. Examples of array topologies that have been proposed are meshes, pyramids, toroids, and hypercubes. Shared-memory parallel processors typically have fewer processors than distributed-memory machines (although there are exceptions of course, such as the RP3, the Butterfly, and the Ultracomputer, which are described in later sections), and these processors, as the name suggests, communicate through a shared memory rather than over links. They are also referred to as "tightly coupled" architectures. In the following we will use the memory-based classification scheme in conjunction with the MIMD and SIMD terms of Flynn, as discussed earlier.

The distinction of shared *vs.* distributed arises naturally when we consider programming models. The shared-memory architecture can support interprocessor communication equally well by shared variables or by message passing. The distributed-memory architecture can only support message passing with reasonable efficiency, because the time to transfer a message is several orders of magnitude slower than memory access to a shared variable. For example, interprocessor communication times in current distributed memory processors are measured in milli-seconds versus microseconds for typical memory access times in a shared

memory processor. Furthermore, the lack of shared memory frequently means that program code cannot be shared and must therefore be replicated on each processor (as well as the operating system). On the other hand, a shared memory may be a potential source of congestion, limiting the practical number of processors.

2.4.2 Impact on Performance

There is now a consensus of opinion that concurrent processing will be essential if computing speeds are to continue to increase. One of the strongest arguments for this point of view is the "speed-of-light" principle. This states that the speed of light limits the rate at which information can be transmitted and, by implication, the rate at which a single processor can perform computations. For example, Denning [1986] estimates that the speed of light limitation will prevent a single sequential processor from exceeding 1 GFLOPS (1 billion floating point operations per second). Such fundamental limitations coupled with the improvements that have occurred in hardware -- dramatic increases in levels of integration and equally dramatic reductions in cost -- have finally made concurrent processing appear to be a practical method of achieving improved rates of computation. Evidence of this is the large number of commercial systems that have appeared in recent years, many of which are relatively inexpensive. This trend is continuing, and it is not an exaggeration to describe this as a veritable revolution in computer design and manufacturing. To substantiate these claims, Figure 2-3 is a plot of cycle time over the past several decades, indicating that the speed is leveling off, consistent with the levelling off in performance for a single CPU indicated in Figure 2-1.

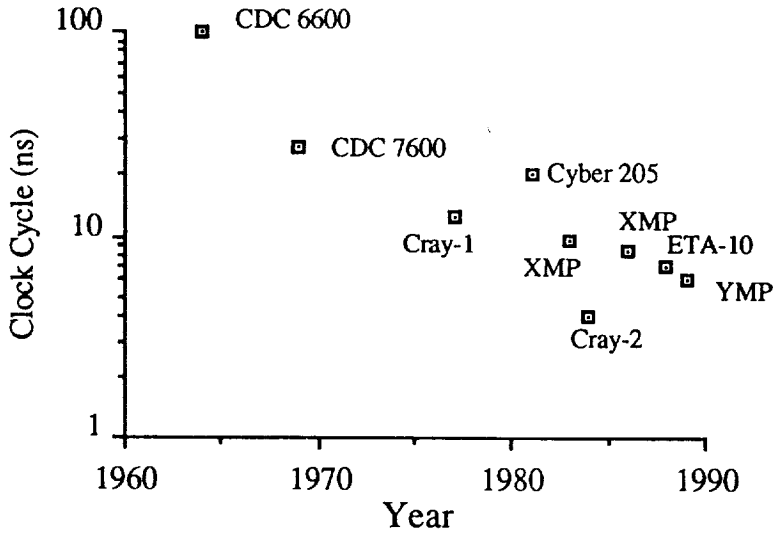
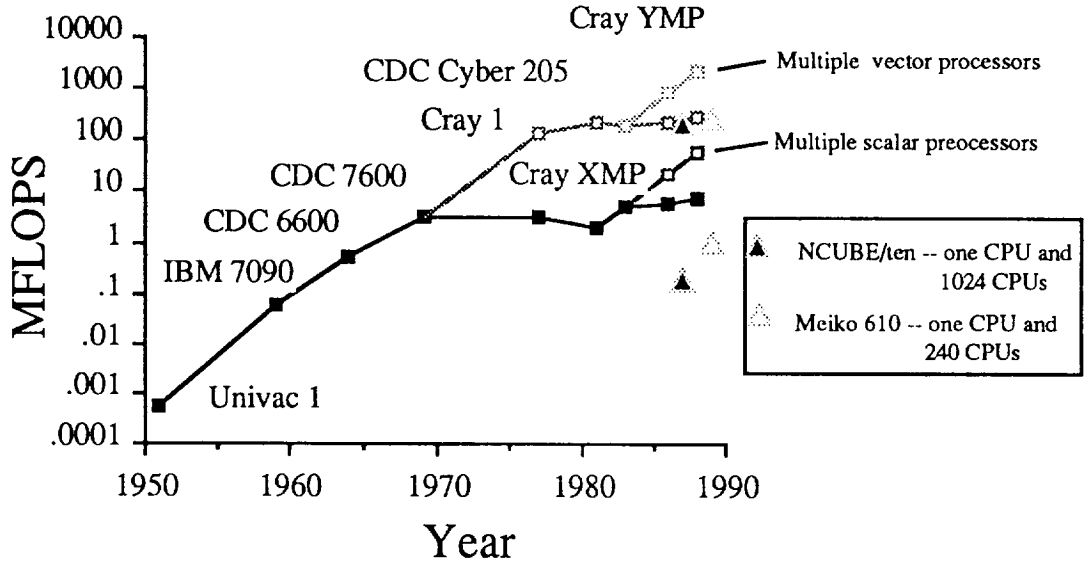


Figure 2-3. Trend in Cycle Time

To illustrate the effect of concurrent processing on computer performance, Figure 2-4 is a plot of the *potential* performance gains with parallelization for

vector/concurrent supercomputers and two MIMD computers, the Meiko 610 and the NCUBE/ten. The Meiko 610 has up to 240 Transputers and the NCUBE/ten is a hypercube parallel processor with 1024 processors. (These machines are described in more detail in a later section.) Clearly, concurrent processing offers the *promise* of dramatic increases in overall performance, but so far this promise has only been realized for a few applications, and Monte Carlo simulation is one of the successful applications areas.



Notes:

- Sources - Same as Figure 2-1
- NCUBE/ten -- .180 MFLOPS per processor (1024 processors)
- Meiko 610 – 1 MFLOP per processor (240 processors)
- Linear speedup assumed for multiple processors

Figure 2-4. Computer Performance Versus Time

2.4.3 Challenge of Concurrent Processing

The basic difficulty with concurrent processing is dealing with multiple processors all working simultaneously on a single problem. This is much more of a challenge to algorithm developers than vector processing, undoubtedly due to the fact that vector processors are basically serial processors. With concurrent processors, the programmer must be concerned with keeping the processors busy, since an idle processor represents an inefficiency that will detract from the overall performance. In addition to keeping them busy, the processors may need to communicate data, and there may be an order that must be imposed on this communication. For example, processor *A* may need to use the results of processors *B* and *C* to continue the calculation, and processors *B* and *C* might be constrained to work on consistent data (such as from the same time step). Thus, there may be a serious data verification and communication problem to ensure that processors are

communicating correctly and using the correct data. For PSM problems, which are inherently parallel, many of these problems can easily be minimized.

2.4.4 Efficiency of Concurrent Processing

The reason for concurrent processing is increased performance and we therefore need some measure of efficiency in order to gauge the relative worth of alternative concurrent processors and different algorithms. The basic speedup equation for vectorization can be used for concurrency with a change in the definition of k, which was defined for a vector processor as:

$$k = \frac{v_{fast}}{v_{slow}} \tag{2-5}$$

Noting that a concurrent computer with N processors should be N times faster than one with a single processor, gives $v_{fast} = N \bullet v_{slow}$, which leads immediately to $k = N$. Using the speedup equation for vectorization speedup to define the concurrency speedup, S, gives:

$$S = \frac{\text{wallclock time for single CPU}}{\text{wallclock time for } N \text{ CPUs}} \tag{2-6}$$

Here wallclock times are being used rather than CPU times and the assumption is made that the computer is dedicated to the job being analyzed. That is, only the job under examination is being executed on either the single processor or the multiple processors. Therefore, the wallclock time for the job to be executed on a single processor will essentially be the same as the CPU time. Using $k = N$, gives the following expression for the theoretical speedup S_N for N processors:

$$S_N = \frac{1}{\alpha + (1 - \alpha)/N} \tag{2-7}$$

where now α is the *serial fraction* -- the fraction of the workload that can only be done on one processor at a time.

The theoretical efficiency ε of a concurrent algorithm is defined as the ratio of the theoretical speedup S_N to the number of processors N,

$$\varepsilon = \frac{S_N}{N} \tag{2-8}$$

This definition is a maximum, theoretical measure and does not take into account other effects that may result in a decreased efficiency, such as:

- synchronization overhead -- extra time required to properly synchronize concurrent tasks
- task overhead -- extra time required to complete task because it is executed on a concurrent processor
- communications overhead -- extra time required due to need to communicate between parallel tasks
- idling overhead - perceived loss of efficiency due to processor idle time during concurrent tasks

Given an *observed* speedup $S_{N,obs}$ with N processors, the effective efficiency of the algorithm is defined as :

$$\varepsilon_{eff} = \frac{S_{N,obs}}{S_N} \tag{2-9}$$

which is a measure of the degree to which the concurrent portion of the algorithm was effectively implemented on the concurrent processor. The extent to which $\varepsilon_{eff} < 1$ measures the effects of overhead, such as synchronization or task overhead (or poor coding). One must be careful with these definitions of efficiency because it is possible for a *serial algorithm to yield an effective efficiency of nearly 100%* as long as the parallel portion of the workload is successfully implemented on the parallel processor. Both definitions are found in the literature, and one must be careful to determine which definition is being utilized. A generalization to explicitly include overhead is discussed below.

2.4.5. Effect of Overhead

The simplest model is one that incorporates one overhead factor, which might be called a concurrency overhead, which accounts for all degradations in performance due to the need to multitask the work across several processors. This overhead may be due to the operating system, synchronization of tasks, or communications overhead. Defining W as the total workload to be carried out, βW as the additional work done to carry out multitasking, v as the speed of a single processor (workload units per unit time), then the time necessary to finish workload W in unitasked mode is

$$\tau_1 = \frac{W}{v} \tag{2-10}$$

and the time to finish W, allowing multitasking with N processors, is

$$\tau_N = \frac{(1 - \alpha)\, W + N\beta W}{N\, v} + \frac{\alpha W}{v} \tag{2-11}$$

where α is the serial fraction. The speedup for N processors is thus given by

$$S_N = \frac{\tau_1}{\tau_N} = \frac{1}{\left[\frac{1 - \alpha + \beta N}{N}\right] + \alpha} = \frac{1}{\left[\frac{1 - \alpha}{N}\right] + \alpha + \beta} \tag{2-12}$$

which indicates that the fractional overhead β is additive to the serial fraction α. Thus, β is tantamount to an additional serial workload, except in this case all processors are busy doing the same thing (the overhead) rather than waiting for one to finish the serial work. This overhead would be negligible for "dusty deck" applications codes where the serial fraction might be expected to be significantly greater than the multitasking overhead. However, in an inherently parallel application such as Monte-Carlo, where the serial fraction α can be arbitrarily reduced by simply increasing the number of histories, this overhead can be significant. This was observed in early studies with the IBM 3090/600, where the overhead to implement multitasking was found to be 1.8% and this dominated the performance of the Monte-Carlo simulation which had a measured serial fraction of only .03% [Denning, 1986]. For example, Table 2-1 illustrates the speedup versus number of CPUs, assuming no serial fraction for the parallel code and a multitasking overhead of 1.8%.

Table 2-1. Maximum Speedup from Multitasking *vs.* Number of CPUs
($\alpha = 0$ and $\beta = .018$)

# CPUs	Maximum Speedup	# "Lost" CPUs
1	.98	.02
2	1.93	.07
3	2.85	.15
4	3.73	.27
5	4.59	.41
6	5.42	.58
7	6.22	.78
8	6.99	1.01
12	9.87	2.13
16	12.42	3.58

The impact of this "small" overhead is evident -- even for a perfectly parallel application (α=0), the overhead to implement multitasking will result in a substantial degradation of performance for a large number of processors. For example, if 16 processors are to be utilized, this results in a loss of 3.6 CPUs, hardly a

negligible loss in total system performance. More sophisticated models to include the effects of overhead have been proposed by Worlton [1987] and discussed by Johnson [1989] and Gustavson, *et al.* [1988]. The effects of overhead on the performance of parallel probabilistic structural mechanics codes is investigated further in Chapter 4.

2.5. A SURVEY OF CURRENT COMPUTER ARCHITECTURES

This section contains a survey of computer architectures currently available from commercial vendors. We first discuss current parallel architectures and then summarize some other relevant technologies. The discussion of parallel architectures is organized by memory hierarchy, that is, shared vs. distributed memory architectures. Simply stated, shared memory machines are composed of multiple processors that are all connected to a central (global) shared memory; whereas in a distributed memory machine, each processor has its own local memory. A shared memory is a potential source of congestion that, in practice, limits the number of processors in shared-memory architectures. There are a number of possible solutions including the multiple stage interconnection networks (multistage-ICNs) discussed in the next section. However, none of these solutions has been explored beyond the prototype stage. In contrast, distributed-memory machines are already available with large numbers of processors. The potential drawback here is communication among the processors, and the amount of memory available to each processor.

2.5.1 Distributed-Memory Machines

To emphasize the number of processors, these systems are often referred to as massively-parallel processors. Current commercial examples of distributed-memory machines are the Intel iPSC-2, the NCUBE-2 (8192 processors), and the Connection Machine CM-2 (65,536 processors) made by Thinking Machines, Inc. The numbers in parentheses are the largest configurations possible. All of these machines are interconnected in a hypercube topology and, with the exception of the Connection Machine, their processors are general purpose computers that can operate independently (in MIMD mode). The processors of the Connection Machine operate in lockstep (SIMD mode) with each processor obeying the same instruction. Its processors are much smaller than those in the other machines -- they are designed to operate on one data bit at a time rather than 16 or 32 bits at a time.

There have been numerous proposals for interconnecting large numbers of processors together that pre-date the commercial machines mentioned above. Examples include 2-dimensional meshes, pyramids, and numerous ICN-based machines (see the next section). A number of experimental machines have been or are being built to test out these proposals. Some notable examples are the Illiac-IV [Barnes, *et al.*, 1968] (mesh connected SIMD), the Goodyear Massively Parallel Processor [Potter, 1985] (16,536 processors, mesh connected SIMD), the pyramid machines of Tanimoto [Tanimoto, 1983], and the Cosmic Cube [Seitz, 1985], a

forerunner of the hypercubes mentioned above. Given the variety of proposals it is interesting to note that the overwhelming majority of first generation commercial distributed-memory MIMD parallel machines have been hypercube connected. There are a number of good reasons for this, but before discussing them we will clarify what is meant by a hypercube multiprocessor.

A hypercube is a generalization of the (3-dimensional) cube to spaces with higher dimension (hyperspaces). Just as a 3-dimensional cube has 2^3 corners (vertices), so an n-dimensional cube has 2^n corners. Similarly, each corner of a 3-cube has 3 edges connected to it, and each corner of an n-cube has n edges connected to it. Hypercube multiprocessors take this simple geometry and use it to define the interconnection pattern among the processors: processors are placed at the vertices of the cube and are connected by links along the edges. For the sake of consistency, lower dimensional cubes (squares, lines, and points) are also regarded as hypercubes (strictly speaking they are hypocubes). Thus the conventional uniprocessor is a 0-cube. In general, an $(n+1)$-dimensional cube can be constructed by replicating an n-cube and then connecting each vertex in the original cube with its corresponding vertex in the replicated cube (see Fig. 2-5).

There are several attractive features of the hypercube geometry. First, the geometry is "isotropic" in the sense that it appears the same from each processor. There are no edges or borders where processors may need to be treated as special cases. This isotropic property can even be extended to include I/O if, as is the case with the NCUBE machines, each processor has a separate I/O channel. Second, the geometry provides a manageable trade-off between two extremes. At one extreme a completely connected geometry to reduce communications time is desirable. Unfortunately, this requires that a multiprocessor with N processors would need $N(N - 1)/2$ communication links to interconnect the processors and that each processor would be connected to N-1 links. For a system with 1024 processors, over a half-million links would be needed and each processor would have to manage 1023 links. Even if the links were simple bit-serial channels the system would be dominated by the interconnection network and by the power required to run it. At the other extreme a small number of links between processors is desirable to keep the system cost within reason. The simplest is a ring -- each processor has only two links to deal with. Unfortunately, the communication time grows linearly with N and in the case of a system with 1024 processors some messages must travel 512 links before reaching their destination. The hypercube strikes a balance between the high-cost/high-connectivity of a completely connected geometry and the low-cost/low-connectivity of a ring geometry. It guarantees that any two processors are no more than n links (n = hypercube dimension, $N = 2^n$) apart, and that each processor is connected to only n links. For a system with 1024 processors this means no more than 10 links separate processors and that only 10 links need to be managed by each processor. It is interesting to compare this to the other two interconnection geometries that have been widely studied -- 2-dimensional meshes and pyramids. Figure 2-6 shows mesh and pyramid geometries.

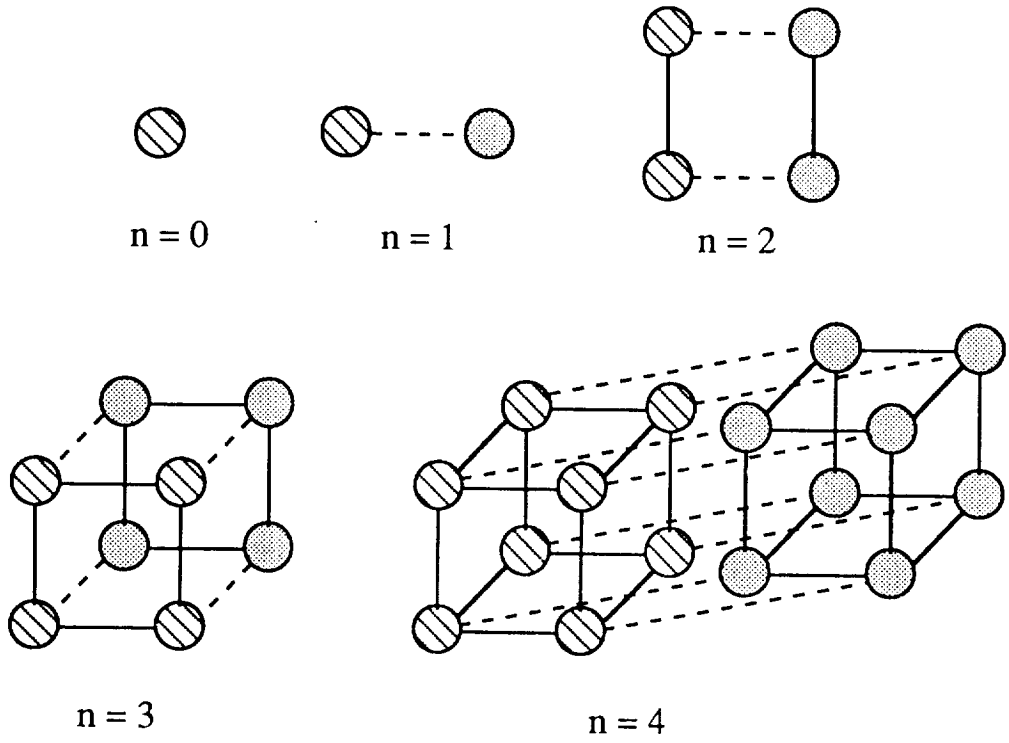


Figure 2-5. Constructing an (n)-Cube from Two (n - 1) -Cubes for n = 0, 1, 2, and 3

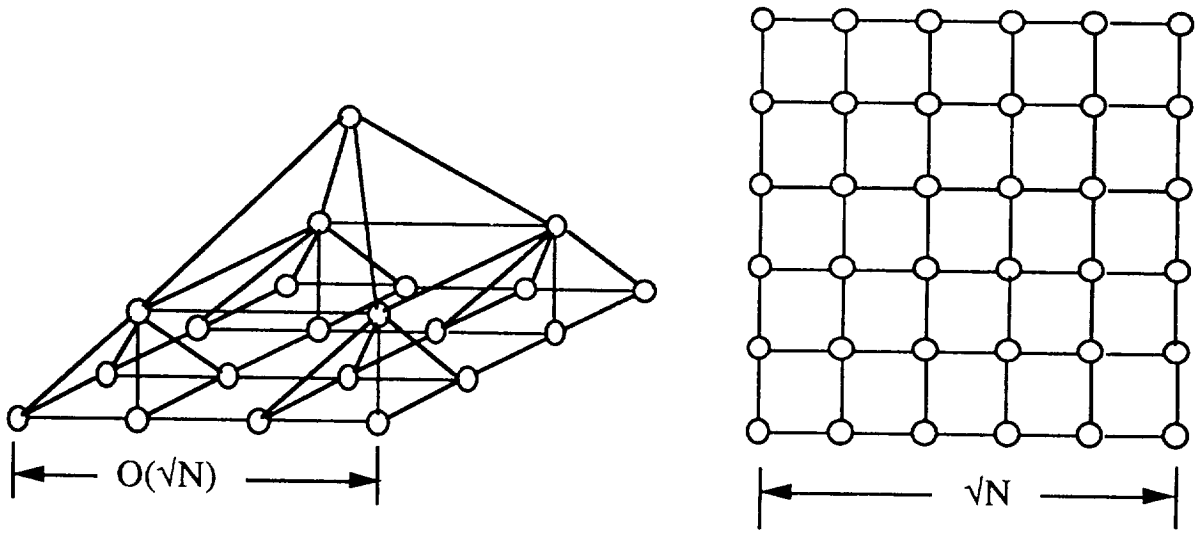


Figure 2-6. Pyramid and Mesh Geometries with N Processors

For a small class of problems the 2-dimensional mesh is ideal but, in general, interprocessor communications can be a limitation -- in the worst case, communications between processors must traverse $2N^{1/2}$-1 links and the small fixed number of links at each processor (≤ 4) is a source of congestion. The pyramid is better in many respects -- the communication delay between processors is

logarithmic as it is with the hypercubes, but the fixed number of links connected to each processor (≤ 9) is also a source of congestion.

The above points undoubtedly made hypercubes attractive to commercial interests wishing to produce massively parallel processors, but there was another ingredient important to their popularity: researchers at Caltech demonstrated that a hypercube (the "Cosmic Cube") could easily be built with off-the-shelf microprocessor components [Fox, 1985 and Seitz, 1985]. Many of the other proposals for massively parallel machines require complex custom chips. This is a more serious restriction than it might first appear, if one considers that a component count of more than a few tens of thousands of ICs puts air-cooled systems at the outer limits of reliability (regardless of the complexity of the subsystem within the ICs).

The first commercial machines were introduced in 1985/6 by Intel, NCUBE, and Ametek. The NCUBE/ten, which had the most impressive specifications of the first generation machines, was capable of a peak performance of 500 MFLOPS (1024 processors, single precision arithmetic) [Hayes, *et al.*, 1986]. In trial experiments we found that about 30% of that capability could be utilized on typical scientific codes like Linpack [Mudge, *et al.*, 1986]. However, a number of new techniques had to be developed to cope with the lack of shared-memory before this level of performance could be obtained, and, unfortunately, many of these techniques were application dependent and do not generalize.

The next generation of hypercube machines can be expected to have a peak performance that is several times that of the NCUBE. Intel has already moved in this direction with the addition of the iPSC-VX machine to their product range. This version of the iPSC has a 20 MFLOP vector processor with 8 MBytes of memory attached to each node. In addition, NCUBE has recently announced the availability of the NCUBE-2 with up to 8,192 processors, each with 8 MBytes of memory and capable of 3.3 MFLOPS, for an aggregate peak performance of 27 GFLOPS [Bacon, 1989]. In addition to improvements in raw processing power, the inter-processor communication rates can be expected to increase dramatically. This will occur through improvements on two fronts: 1) special hardware to support very high speed communications is being developed; and 2) software that works with the new hardware is being developed to replace the current store-and-forward message routing with newer techniques such as virtual cut-through and low-overhead nearest neighbor communications [Mudge, *et al.*, 1987]. The improvements may be sufficient to allow the new machines to be viewed as logical shared-memory machines, *i.e.*, access to remote shared variables will no longer be prohibitively time consuming.

As noted earlier, there is a second class of parallel architectures, SIMD parallel, which is fundamentally different from the MIMD parallel architectures discussed thus far. SIMD parallel computers are characterized by many identical processors (for example, 4096 processors for the Active Memory Technology (AMT)

Dap; 16,384 for the MasPar; and 65,536 processors for the Connection Machine CM-2 offered by Thinking Machines, Inc.). The processors in a SIMD parallel computer are generally simple. In the Connection Machine and AMT DAP, the processors are single bit processors and in the MasPar they are 4-bit processors. These machines operate in a lockstep (synchronous) fashion, controlled by a single supervisor CPU. That is, each processor performs the same instruction at the same time. This is in contrast to a parallel MIMD processor with many CPUs, each of which operates independently and asynchronously.

From a functional standpoint, there are important similarities between SIMD parallel and SIMD vector architectures. In essence, a SIMD parallel processor is still a serial processor, in the sense that one can look at a single instruction from the control processor to all the distributed processors as the analogue of a single vector instruction of a vector processor. For example, consider the AMT Dap with 4,096 single bit processor and a Cray with a 64 word (64 bits each) vector. The result is a vector, 4,096 bits in length for both machines. These machines, while originally developed for AI applications, are showing impressive performance on scientific problems due to their ability to perform vector and matrix operations.

A final type of distributed memory parallel processor is that based on the transputer. For example, the Meiko parallel processor utilizes Inmos Transputers as the nodes. A Transputer is basically an integrated chip with custom CPU and "built-on" communications connections (4 links per processor). In essence it is an"off-the-shelf" building block for parallel processors, since they can be arranged in various ways, including a hypercube (maximum order 5), ring, or grid, among others. There is no theoretical limit to the number of Transputers that can be linked together, the largest is probably at the University of Edinburgh, with over 300 Transputers. A production Monte Carlo particle transport code, MONK6, has been successfully ported to the Meiko and is packaged along with the Meiko for potential buyers. The advantage of the Meiko is its relatively large memory per processor (up to 8 MBytes) and its maturity, since it has been a commercial product for a number of years, with mature operating system and compilers.

2.5.2 Shared Memory Machines

Shared memory machines can conveniently be divided into two groups: those that use multistage-ICNs to connect their processors to the shared memory, and those that use more conventional means, such as a shared bus. We will discuss the multistage-ICN machines first, in the following subsection. It is interesting to note that, apart from a few exceptions, these machines are experimental.

Multistage-ICN Based Machines. Multistage-ICNs were developed to provide a high bandwidth connection to a shared memory without incurring the prohibitive complexity of a crossbar network. As such, they offer the best opportunity to equal the massive parallelism of distributed memory machines while retaining the ease of use of a shared memory architecture. They are intended for very large scale

programs in which there is significant intra-program parallelism. The multistage-ICN that connects the processors to the shared memory is a key feature of these architectures. Indeed, the multistage-ICN has been the subject of a considerable amount of research in its own right [Siegel, 1985]. Examples of machines that use multistage-ICNs are the University of Illinois Cedar machine [Kuck, *et al.* 1986], the Purdue PASM [Siegel, *et al.*, 1981], the NYU Ultracomputer [Gottlieb, *et al.*, 1983], and the RP3, a machine based on the NYU work that is being built by IBM [Pfister, *et al.*, 1985]. These machines are all experimental prototypes that will have anywhere from a few hundred to several thousand processors. The BBN Butterfly [Crowther, *et al.*, 1985] with up to 504 processors is currently the only commercial machine in this class. All these machines use variations on the Omega multistage-ICN first proposed by Lawrie [1975].

The practical importance of multistage-ICNs is likely to grow as we gain experience from experiments such as the RP3 and Cedar. Therefore, we will devote the rest of this subsection to discussing the operation of these networks.

Figure 2-7 shows a multistage-ICN, the shuffle-exchange network [Stone, 1987], a variation of the Omega network mentioned above. We have shown it as connecting processors to a set of memories. These memories together form the shared memory. In addition, each processor usually has its own cache or local memory. Other organizations are possible; for example, the processors could equally well be connected back onto themselves. In such a case the local memories of each processor would form the shared memory. A shuffle-exchange network that connects N processors to N memories has n (= $\log_2 N$) stages. Each stage consists of a perfect shuffle connection pattern followed by N exchange boxes. The exchanges are 2 x 2 crossbars that can connect any one of the two input ports to any of the two output ports. For a message entering an input port, one bit is sufficient to direct it to the desired output port. *A* 0 and 1 are shown on the diagram of the exchanges to indicate the output port that a message will be directed to by its routing bit. The perfect shuffle gets its name by analogy with the shuffle operation on a deck of cards. If we imagine the left side of the perfect shuffle pattern to be the positions of cards in a deck (we have an 8 card deck in Fig. 2-7), then the right hand side of the pattern shows the position of the cards in the deck after a perfect shuffle operation.

To route a message through the shuffle-exchange network, a destination address is required that identifies the memory to be accessed. At each stage M in the network the message passes through an exchange box. A bit from the destination address is used to determine which output port of the exchange box the message should be directed to. At the first stage the first bit is used, at the second stage the second bit is used, and so on. Figure 2-7 shows a message being routed from P2 to memory 110_2.

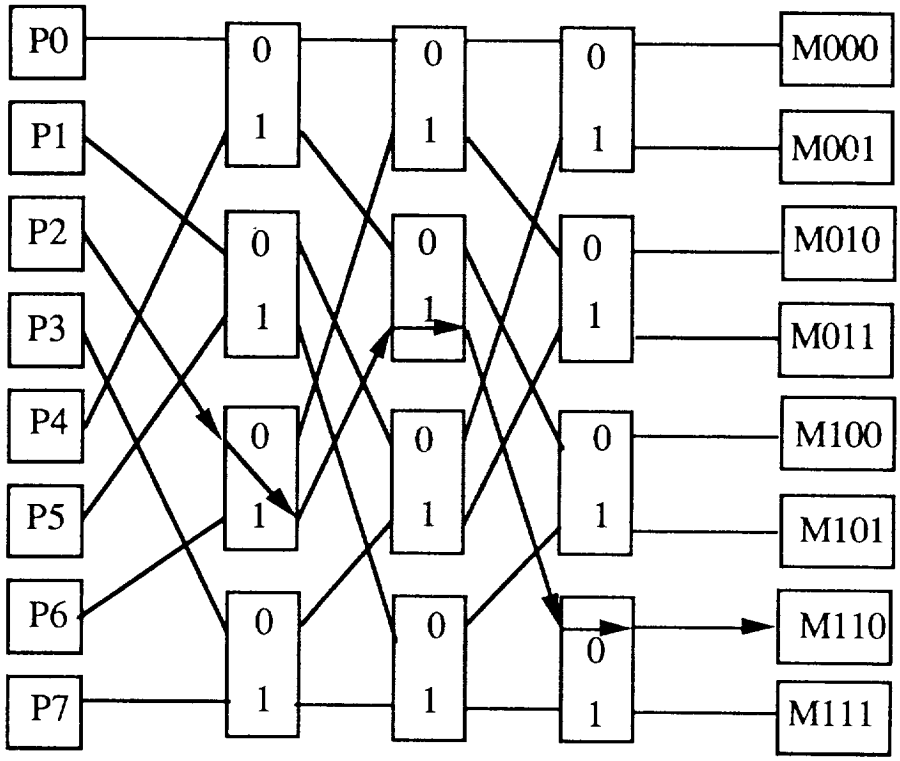


Figure 2-7. A Shuffle-Exchange ICN

Multistage-ICNs provide a high-bandwidth connection to a shared memory. However, congestion can still occur if two messages require the same output port of an exchange box. The effects of congestion can be reduced by adding buffers at the inputs of each exchange. The network can then be operated in a pipelined manner. The control and scheduling of these networks is quite complicated especially if buffering and pipelining are added. Multistage-ICNs offer the possibility of building massively parallel machines with shared memory. These would be ideal execution vehicles for AI programs, and, in particular, the kinds of strategy-level programs that will be part of any smart robot. Early work on building prototypes has been encouraging, but much work remains if these networks are to realize their potential.

Shared-bus Machines. Machines in this class are comprised of at most a few dozen processors connected to a shared memory over a high speed bus. To date, they have been by far the most successful parallel processors from a commercial standpoint. This category includes such large parallel processors as the Cray multiple processor products (*e.g.*, Cray X-MP, Cray-2, and Cray Y-MP), the IBM 3090-600, and the Alliant FX series of minisupercomputers. Figure 2-8 illustrates the architecture of the Alliant FX/80. These computers have operating system features and extended Fortran compilers which allow parallelism within a single Fortran job. In addition to these well-known examples, there are shared-bus parallel processors which are primarily intended to be used in a multiprogramming mode, where complete programs execute sequentially on a single processor, that is, there is no parallelism within each program. The processors themselves are typically 32-bit

microprocessors with a local cache, such as the Sequent and Encore computers [Dongarra and Duff, 1989].

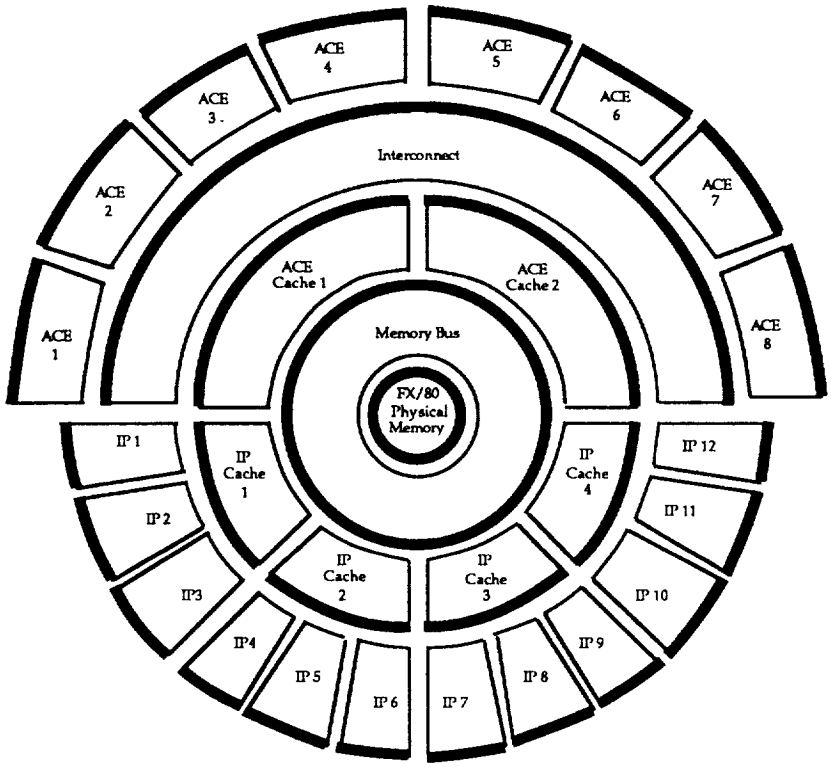


Figure 2-8. Architecture of Alliant FX/80

The design philosophy of these machines is evolutionary. They are based on off-the-shelf processors and they are integrated by a common backplane -- the shared bus. Most of the technology is well understood. The only exceptions are the extensions to the operating system required to handle multiple processing resources, and mechanisms to distribute interrupts and exceptions across several processors. This evolutionary approach makes a lot of sense from a commercial viewpoint -- there is less uncertainty (and risk). However, it does appear that the shared bus may be an insurmountable obstacle to expansion. On the other hand there have been a number of developments that will allow systems to be built with several hundred processors. These include larger caches and improved caching strategies that reduce the per processor bus traffic.

2.5.3 Other Technologies

Super-minicomputers and Minisupers. The rapid advance of computer hardware technology means that features that today are found in supercomputers are likely to be found in tomorrow's microprocessors. This will have a significant impact on distributed parallel architectures. Some important features are vector

processing and the use of very high speed circuit technologies such as ECL (emitter-coupled logic) and GaAs (Gallium Arsenide).

These machines have already impacted the design of less costly machines. A notable example that was already mentioned in the section on distributed-memory machines is the Intel iPSC-VX. This machine relies on high performance chip sets that use a combination of vector processing techniques and advanced technology for their logic circuits to achieve performance in the range 1~20 MFLOPS.

The circuit technology of these chip sets is usually a bipolar technology such as ECL, which was pioneered for supercomputer use. GaAs, which is also being pioneered for super- computers, promises a factor of ten improvement in computation speeds over that obtained using ECL. To give an illustration of the impact of these technologies, consider an arithmetic unit constructed with conventional high performance logic circuits (*e.g.* CMOS) that has a peak performance of 10 MFLOPS. If the same unit is constructed from ECL it will be capable of a peak performance of about 20-30 MFLOPS. If it is constructed from GaAs logic circuits it will be capable of a peak of about 40-50 MFLOPS. Of course, GaAs technology is still in its early stages.

Systolic machines. Systolic computers were first proposed by H. T. Kung and C. E. Leiserson of Carnegie-Mellon University [1980]. Unlike most of the other architectures described in this section, systolic machines have not been commercialized (notwithstanding a few application-specific machines). Systolic machines are a marriage of pipelining techniques with VLSI technology. The idea is to construct highly parallel computers from iterative arrays of computing elements (cells) and then to stream data through the array so that the computations are performed in a pipelined fashion. The analogy with the blood stream lead to the term systolic being adopted to describe such computations.

The general concept of constructing arrays of identical cells is an ideal match with integrated circuit technology, because it is particularly suited to the fabrication of systems which conform to a repeated pattern (memory chips are the prime example). In addition, computations performed by arrays of cells usually lend themselves to pipelining, hence high throughput rates are possible. Finally, streaming data through an array of computing cells greatly reduces the memory bandwidth requirements because intermediate results are never stored in memory, instead they are sent directly to the next cell in the array.

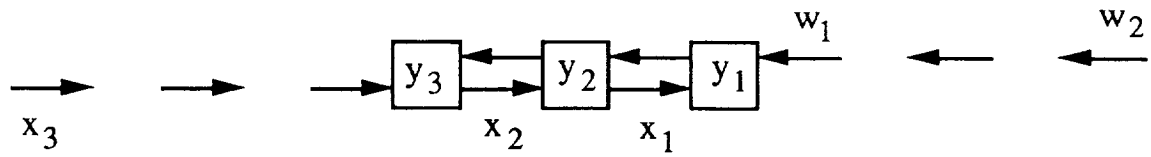


Figure 2-9. Systolic Array for Convolution

The early ideas for systolic machines were quite elaborate; for example, 2-dimensional arrays of hexagonally connected cells were proposed. More recent examples have been simpler. The best known current project, the Warp computer [Annaratone, *et al.*, 1987], is just a simple linear array, although the cells are complex computers. To illustrate the operation of a systolic machine, consider Fig. 2-9. It shows an array for computing the convolution, $y_1, \ldots, y_{n+1-k}$, of a set of weights $w_1, \ldots, w_k$ with an input sequence $x_1, \ldots, x_n$, in other words,

$$y_i = w_1 x_i + w_2 x_{i+1} + \ldots + w_k x_{i+k-1}, \qquad i = 1, \ldots, n+1-k$$

The sequences of x's and w's move through the array in opposite directions. When an x meets a w in cell i, they are multiplied and accumulated into the partial result for y_i. After the streams have completed their passage through the array the results (y_i's) are left in their respective cells. To make sure that every x_i meets every w_i the components of the two sequences are spaced two cycles apart.

Clearly, the applications for systolic machines are limited to computations that can be can be pressed as convolutions or vector operations. This includes a number of important operations as well as convolution that are useful for many signal processing applications that occur in robotics. For example, the system of Fig. 8 formed the basis for a pattern matching chip [Foster and Kung, 1980]. Systolic machines are likely to find a position as application-specific attached processors.

Dataflow Machines. Dataflow computers had their beginnings in the late 1960's and early 1970's with the work of Jack Dennis [1969]. Since their inception dataflow machines have held out the promise of high performance parallel processing. A number of experimental machines have been constructed [Dennis, 1979], but the promise has yet to be fulfilled.

The basic idea behind dataflow machines is to have the hardware detect when all the data for an executable statement (its operands) has been computed and then to schedule that statement for execution. For example, the statement,

```
A = B + C;
```

would be readied for execution as soon as the values for B and C were calculated. The execution of instructions is determined by the flow of data and not by a program counter, as is the case with the von Neumann architecture. In principle this allows instructions to execute in parallel because the only constraint on the ordering of their execution is the presence of data. The idea is an elegant one, but it is not without some problems. One that is immediately apparent is the need to build computer hardware that can efficiently detect when the input data of a statement has been computed. This requires a substantial change in the way computers are constructed, ruling out the use of current hardware. Even if this is achieved, the

degree of parallelism that can be obtained will be no better than can be extracted by modern compiler techniques.

It would appear that dataflow machines have "missed the boat". The reasons for this are complicated. One factor is undoubtedly that they represented too great a departure from the status quo. A dataflow computer would be incompatible with conventional computers, and to use them appropriately would require a revolution in the computer world, from the standpoint of operating systems, compilers, and programming. Such a revolution would only be attractive if the improvements in performance were likely to be considerable; however, the potential for a quantum jump in performance has yet to be convincingly demonstrated.

2.5.4 Concluding Remarks

In this section, we have tried to err on the side of surveying too many architectures. On the other hand we have not included anything about the relatively new field of neural networks. This is because neural networks are, strictly speaking, a model of computation, not an architecture, and architectures inspired by this line of research have yet to be defined.

In the supercomputer category it has not been uncommon for machines to have more than one processor, and, since the days of the CDC-6600, multiple functional units have been common. However, as we have seen from the survey, only recently have a number of more "affordable" parallel processors appeared. These range in type from massively parallel cubes to relatively conventional shared-bus architectures. This relatively sudden emergence of a wide range of modestly priced commercial parallel processors has opened up tremendous possibilities for research into parallelism that goes beyond the paper studies of the past, and is going to have a profound impact on the kinds of things that we will be able to compute in the near future. Nowhere is this likely to have more impact than in the computational problems that are encountered in science and engineering.

There is another class of "parallel processors" that should be mentioned. This is a network of relatively cheap workstations (*e.g.*, from Apollo, Sun, DEC, IBM, etc.) which can be combined to work in parallel on a single task. For example, the PAX-1 system from VXM Technologies can be used to transform a network of DEC VAXes into a parallel processing computer. The potential of these individual machines is such that this alternative mode of parallel computation must be considered. For example, the new System 6000 workstations from IBM are faster than a Cray XMP (in scalar mode), and nearly as fast as the Cray YMP. Since these workstations cost a fraction of a Cray XMP or YMP, the price-performance ratio of these computers is outstanding. This has been noted in a recent article [1990] in the New York Times about the so-called "killer micros", pertaining to the use of many low-cost microcomputers to perform scientific computation at speeds greatly in excess of current-day supercomputers. Thus, a candidate "parallel computer" for scientific computation may very well be networks of advanced function workstations.

2.5.5. Commercially-Available Vector and Concurrent Computers

Table 2-2 contains a list of vector and parallel processors that are commercially available at the present time. This table does not include many of the experimental machines mentioned above since they are either special purpose or else not in a commercial category. In general, advanced function workstations have not been added (*e.g.*, the IBM System 6000, the Apollo DN10000, the "Stardant", or the DEC SparcStations), although these machines are beginning to blur the distinctions between workstations and supercomputers, as noted above.

Table 2-2. Vector and Parallel Computers Currently Available

Name	Type	CPU	Memory	MBytes	*N*	Clock	MFLOPS
Alliant FX-80	MIMD	Scalar/vector	Shared	256	12	170	188.8
Alliant FX/2800	MIMD	Scalar/vector	Shared	1,024	14	---	1,000
Amdahl VP-1400E	SIMD-V	Scalar/vector	Shared	1,024	1	7	1,714
AMT DAP 610	SIMD-P	Bit-serial	Shared	64	4,096	100	--
Ardent Titan	MIMD	Scalar/vector	Shared	128	4	62.5	64
BBN Butterfly TC2000	MIMD	Scalar	Shared	16,096	504	---	10,080
Convex C-210	MIMD	Scalar/vector	Shared	4,000	4	40	200
Cray-2	MIMD	Scalar/vector	Shared	2,048	4	4.1	1952
Cray-3	MIMD	Scalar/vector	Shared	4,096	16	2	16,000
Cray-XMP	MIMD	Scalar/vector	Shared	128	4	8.5	940
Cray-YMP	MIMD	Scalar/vector	Shared	256	8	6	4,000
Elxsi M6460	MIMD	Scalar	Shared	2,000	10	31.25	100
Encore 320	MIMD	Scalar	Shared	128	20	67	50
Hitachi S820	SIMD-V	Scalar/vector	Shared	512	1	4	3,000
IBM 3090/600S	MIMD	Scalar/vector	Shared	512	6	16	800
Intel iPSC/2	MIMD	Scalar	Distributed	1,024	128	--	80
Intel iPSC/2 VX	MIMD	Vector	Distributed	1,024	128	100	2,560
International Parallel IP-1	MIMD	Scalar	Shared	264	33	--	600
MasPar	SIMD-P	4 bit-serial	Shared	256	16,384	--	600
Meiko	MIMD	Scalar	Distributed	4,000	800	50	1,000
NCUBE	MIMD	Scalar	Distributed	524	1,024	125	300
NCUBE-2	MIMD	Scalar	Distributed	65,536	8,192	50	27,000
NEC SX-2A	SIMD-V	Scalar/vector	Shared	1,024	1	--	1,300
Sequent Symmetry S81	MIMD	Scalar	Shared	240	30	62.5	390
Thinking Machines CM-2	SIMD-P	Bit-serial	Shared	512	65,536	--	31,000
Unisys ISP 1100/90	MIMD	Scalar/vector	Shared	70	4	30	67

Notes:
(1) Generally only maximum configuration per vendor is given
(2) *N* = maximum number of processors
(3) MBytes = maximum memory configuration in megabytes
(4) SIMD-V = SIMD (vector); SIMD-P = SIMD (parallel)
(5) Clock = cycle time (ns)
(6) Only commercially-available products included
(7) List does not include attached processors or "one-of-a-kind" products
(8) MFLOPS = "peak" 64-bit performance
(9) Primary source of data : [Dongarra and Duff, 1989]

CHAPTER 3

PARALLELISM IN PROBABILISTIC STRUCTURAL MECHANICS

3.1 INTRODUCTION

Probabilistic structural mechanics problems are inherently parallel. This parallelism makes these problems well suited for investigation of parallel processing implementation. In this chapter we first briefly review probabilistic structural mechanics methods and then identify the sources of parallelism.

Several levels of parallelism in PSM problems may need to be exploited in order to achieve optimal speedup on a parallel processing computer. Two macro-scale levels of parallelism are illustrated in Figure 3-1. The top level parallelism results from parallelism associated with the probabilistic aspects of the problem, and the lower level parallelism results from the structural mechanics aspects of the problem. In addition to these macro levels of parallelism there are many levels of micro-scale parallelism. These are additional levels of parallelism associated with the structural mechanics aspects of the problem, including both concurrency and vectorization. These will be further described in the following sections.

3.2 PARALLELISM IN PROBABILISTIC COMPUTATIONS

3.2.1 Overview of PSM Methods

Before identifying the parallelism inherent in the probabilistic computations required in a PSM problem, we first present a brief overview of PSM methods. The purpose here is to provide conceptual descriptions to aid in the discussions of parallelism that follow. In depth treatment and mathematical details of PSM methods can be found in several texts [*e.g.*, Ang, A. and Tang, W., 1984; Madsen, H., Krenk, S., Lind, N., 1986].

Simply stated, in a PSM problem it is necessary to determine the probability distribution of structural response or damage. For example, Figure 3-2 shows the results of a PSM analysis for the second stage turbine blade of the Space Shuttle Main Engine [Newell, *et al.*, 1989]. The PSM results are given by the plot of the response cumulative distribution function (CDF) shown in the figure. The CDF gives the probability that various effective stress levels will not be exceeded.

Mathematically, the problem is to solve for the probability distribution of a response variable, w, that is a function, g, of a vector of variables, $\mathbf{x}$. The function, $g(\mathbf{x})$, is commonly called the performance function. The vector of variables, $\mathbf{X}$, represents all the problem variables, both random and deterministic, such as member dimensions, material properties, boundary conditions, and loadings. The response cumulative distribution function (*e.g.*, Figure 3-2) is obtained by evaulating

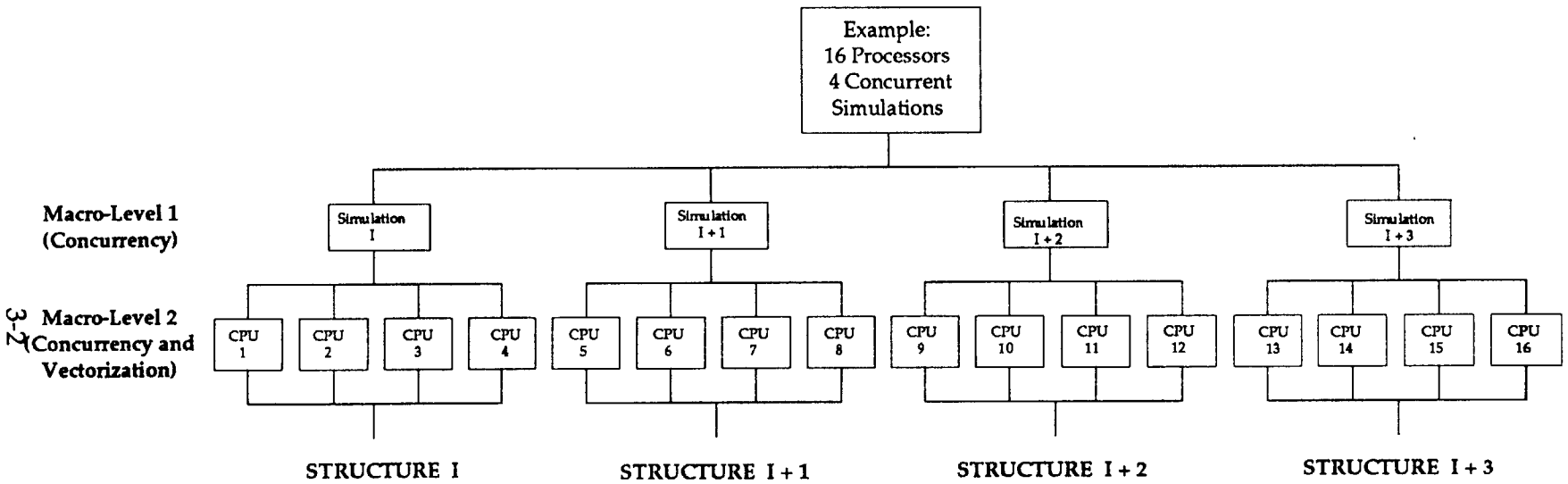
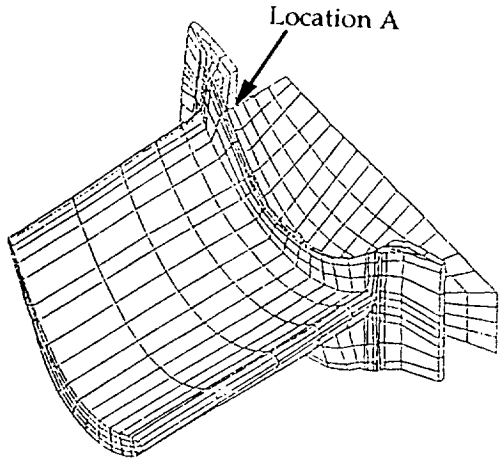
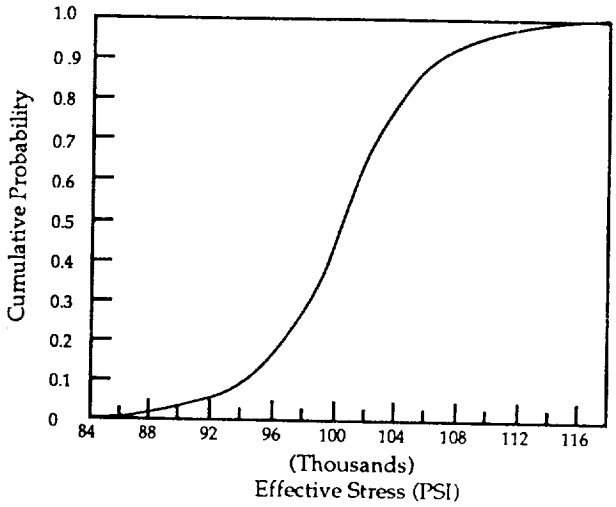


Figure 3-1. Multi-Level Parallelism in Probabilistic Structural Mechanics



Turbine Blade Finite Element Model Used for PSM



Cumulative Distribution Function for Effective Stress at Location A

Figure 3-2. Example PSM Analysis

$$P[g(x) \leq y] \tag{3-1}$$

(where P denotes probability) for all values of y. We note that, in general, we are interested in multiple stochastic response variables, representing response at various points in the structure.

For PSM problems, evaluation of the performance function requires solution of the structural mechanics problem. For problems of practical interest this may entail solving systems of linear equations, systems of nonlinear equations, or time variant dynamical systems.

A number of methods have been developed to evaluate the response probability functions. All are based on one of two basic approaches: (1) partial derivative analysis, or (2) pseudo-random sampling. Pseudo-random sampling methods include, for example, Monte-Carlo simulation, Monte-Carlo simulation with variance reduction (*e.g.*, Importance Sampling, Stratified Sampling), and Experimental Design methods. Partial derivative methods include, for example, First Order Reliability Methods (FORM), Second Order Reliability Methods (SORM), First Order Second Moment Methods, and Second Order Second Moment Methods. In addition, hybrid methods are possible wherein, partial derivative methods, sampling methods, or perturbation methods are used to develop a first or second order response surface and Monte-Carlo methods are then used to evaluate the response uncertainty.

For purposes of our discussions, we present below the basis of the most common partial derivative method, FORM, and the Monte-Carlo simulation method. We also present here, the transformation required for treatment of correlated variables, which is also a source of parallelism in PSM computations and has been implemented in the computer code developed under this effort.

FORM. In the First Order Reliability Method, the performance function, g, is approximated by a linear hyperplane. Since, the performance function will, in general, be a nonlinear hypersurface, the approximate hyperplane is fit as a tangent to the hypersurface at a single point. This point is selected to be the most probable failure point on the surface. Once the hyperplane is fit, it is straightforward to evaluate Eq. 3-1 for the linearized performance function.

The most probable failure point is given by:

$$x'^{*}_i = \frac{-\left(\frac{\partial g}{\partial x'_i}\right)_* \beta}{\sqrt{\sum \left(\frac{\partial g}{\partial x'_i}\right)^2_*}} \tag{3-2}$$

(for all n random variables) where the random variables have been transformed into the standard normal space, *i.e.*, with zero mean and unit variance;

$$x'_i = \frac{x_i - \mu_{x_i}}{\sigma_{x_i}} \tag{3-3}$$

and are assumed to be uncorrelated (treatment of correlated variates is described later); and β is the distance to the most probable failure point in the standardized normal space, *i.e.*,

$$\beta = \sqrt{x'^{2}_i + \cdots + x'^{2}_n} \tag{3-4}$$

Note that the most probable failure point is the point of minimum distance from the origin to the failure surface in the standardized space. This distance is often referred to as the reliability index and is itself a measure of structural reliability.

A number of methods have been proposed for evaluation of the most probable failure point. These will not be presented here. It is important to note, however, that in most approaches the solution will require evaluation of the derivatives of the performance function. These derivatives must typically be evaluated for each random variable, and at several points on the failure surface

(since the solution methods are iterative in nature). The derivatives are most commonly evaluated by using finite differences since the performance function will normally not be available in close form. Hence, the use of the FORM method will require repeated evaluations of the performance function with perturbed values of the problem variables. This is clearly an inherently parallel problem.

Monte-Carlo Simulation. In Monte-Carlo simulation, the performance function is solved repeatedly for different values of the problem variables and the results scored in order to obtain an estimate of the probability given by Eq. 3-1. Mathematically, this probability can be estimated as the statistical expectation of the event, $g(\mathbf{x}) < 0$, and calculated as:

$$P_g = \int_x I(g(x))\, f_x(x)\, dx = E[\, I(g(x))\,] \tag{3-5}$$

where $I(\bullet)$ is the indicator function and $E[\bullet]$ the expected value (statistical mean). The indicator function $I(\bullet)$ is defined in accordance with the event under discussion. That is, for the event $g(x) \le 0$, then $I(\bullet)$ is defined as $I(g(x) \le 0) = 1$ for $g(x) \le 0$ and 0 otherwise.

If the density function $f_x(x)$ exists and independent sample vectors of X can be generated, the estimation of P_g can be calculated as the sample mean of $I(g(x_i))$, *i.e.*,

$$P_g \approx \hat{P}_g = \frac{1}{N} \sum_{i=1}^{N} I(g(x_i)) \tag{3-6}$$

where N is the total number of samples, and x_i the i^{th} sample vector of X. The estimator $\hat{P}_g$ itself has an uncertainty, due to the finite number of samples (or simulations). In theory, this uncertainty can be quantified by the variance of the estimator which is calculated by

$$\begin{aligned} \mathrm{Var}\left(\hat{P}_g\right) &= \frac{1}{N} \mathrm{Var}\left(I(g(\mathbf{x}))\right) \\ &\approx \frac{1}{N(N-1)} \sum_{i=1}^{N} \left[\left(I(g(\mathbf{x}_i))\right) - \hat{P}_g\right]^2 \end{aligned} \tag{3-7}$$

The coefficient of variation or relative statistical error of $\hat{P}_g$ is then evaluated by

$$\delta\left(\hat{P}_g\right) = \frac{\sqrt{\mathrm{Var}\left(\hat{P}_g\right)}}{E\left[\hat{P}_g\right]} \tag{3-8}$$

Eq. 3-8 can be used to estimate the required sample size for a prescribed relative statistical error.

Therefore, the basic steps in numerical Monte Carlo simulation are: (1) generating sample variables based on the density function $f_x(x)$; (2) evaluation of the performance function $g(x)$ using the generated sample variables; (3) calculating $I(g(x))$; and (4) performing the scoring procedures as outlined in Eqs. 3-6 and 3-7.

Since the g function must be repeatedly evaluated for independent sample values of the random variables, the numerical procedure can be carried out in parallel. That is, the g function can be evaluated for the different sample sets on independent processors. Since this method will be implemented on a parallel processor, as described in the next chapter, we present some additional relevant details below.

If $\boldsymbol{X}$ consists of independent random variables $(x_1, x_2, x_3, \bullet\bullet\bullet, x_m)$, the sample variables can be generated by direct inversion of marginal cdf's of individual random variables, *i.e.*,

$$x_1 = F_{x_i}^{-1}(u_i), \qquad i = 1, 2, 3, \bullet\bullet\bullet, m \tag{3-9}$$

where $F_{x_i}^{-1}(\bullet)$ is the inverse cdf of random variable x_1 , and u_i is a generated random number from a uniform distribution.

Without losing generality, it can be assumed that all random variables are standardized, *i.e.*, with zero mean and unit variance. Moreover, if all the random variables are of normal distribution, Eq. 3-9 can be rewritten as

$$x_1 = \Phi^{-1}(u_i), \qquad i = 1, 2, 3, \bullet\bullet\bullet, m \tag{3-10}$$

where Φ^{-1} is the inverse standard normal distribution function.

Treatment of Correlated Variates. In the case where the x's are correlated standard normal with a correlation coefficient matrix $\boldsymbol{R}$,[1] it is possible to find a transformation such that the x's are transformed into a set of independent standard normal variables z's. One such transformation is as follows:

[1]The correlation coefficient matrix is obtained from statistical analysis of data for the problem input variables. Examples are given in Sues and Twisdale, 1988.

$$z = T(x) = L^{-1} x \tag{3-11}$$

where L is the lower triangular matrix of the Cholesky decomposition of the correlation coefficient matrix, *i.e.*, $R = L\,L^T$ The sample vector x can thus be generated by

$$\begin{aligned} x &= T^{-1}(z) \\ &= L\,z = L\,[\,\Phi^{-1}(u_1), \Phi^{-1}(u_2), \Phi^{-1}(u_3), \bullet\bullet\bullet, \Phi^{-1}(u_m)\,]^T \end{aligned} \tag{3-12}$$

The procedures described above are valid only for normal variates. To generate samples from non-normal random variables, additional transformations are necessary. The Rosenblatt transformation [Rosenblatt, 1952; Hohenbichler and Rackwitz, 1981] is one such transformation. However, it requires the calculation of conditional probabilities which are often complex. Among other transformations, the Nataf-model transformation, proposed by Nataf [1962] and enhanced by Liu and Der Kiureghian [1986], has more merit in practical application.

The Nataf-model transformation first defines the marginal transformation for each individual random variable as

$$z_1 = \Phi^{-1}[\,F_{x_i}(x_i)\,], \qquad i = 1, 2, 3, \bullet\bullet\bullet, m \tag{3-13}$$

Thereby, following the rules of probability transformation, the joint pdf of X can be written as

$$f_x(x) = f_{x_1}(x_1)\, f_{x_2}(x_2) \bullet\bullet\bullet f_{x_m}(x_m)\, \frac{\phi_m(z,R')}{\phi(z_1)\,\phi(z_2) \bullet\bullet\bullet \phi(z_m)} \tag{3-14}$$

where $\phi(\bullet)$ is the standard normal pdf and $\phi_m(z,R')$, the m-dimensional joint pdf of standard normal variables z with correlation coefficient matrix R'. R' is a modified correlation coefficient matrix with element ρ'_{ij} defined in terms of the original correlation coefficient ρ_{ij} via

$$\rho_{ij} = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} x_i x_j \, f_{x_i}(x_i) \, f_{x_j}(x_j) \, \frac{\phi_2(z_i, z_j, \rho'_{ij})}{\phi(z_i)\,\phi(z_j)} \, dx_i \, dx_j$$

$$= \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} x_i x_j \quad \phi_2(z_i, z_j, \rho'_{ij}) \quad dz_i \, dz_j \tag{3-15}$$

For each pair of marginal distributions with given ρ_{ij}, iterative procedures are necessary in order to solve Eq. 3-15. To avoid these calculations, empirical formulae have been developed for some commonly used distributions [Liu and Der Kiureghian, 1986]. For arbitrary types of distributions, a method based on the truncated expansion of $x_i x_j$ with respect to z_i and z_j was also developed [Wu, *et al.*, 1988]. In the latter approach, however, if nonlinear terms are retained, numerical procedures are still needed for computing higher order derivatives and solving a nonlinear algebraic equation.

In short, a practical procedure for generating random samples from correlated non-normal random variables can be described as follows,

(1) Transforming the original correlation coefficient matrix R into R' through the Nataf-model transformation ;

(2) Using the Cholesky decomposition $R' = L\,L^T$ and Eq. 3-12 to generate samples of z_i's;

(3) Applying the inverse transformation of Eq. 3-13 to obtain the samples for x_i's.

3.2.2 Identification of Parallelism

From the overview of PSM presented above, there are several fundamental sources of parallelism in PSM computations. The major source of parallelism results from the required multiple evaluations of the performance function. In addition, the Nataf-model transformation, for transformation of correlated variates, is also inherently parallel. This is because the calculation of the modified correlation coefficient for any pair of random variables can be carried out independently. Hence, the elements of the modified correlation coefficient matrix can be calculated concurrently. Other sources of parallelism are also present that are specific to the PSM method being used, as described below.

Partial Derivative Methods. For the partial derivative methods, multiple evaluations of the performance function (g function) are required for calculation of the partial derivatives (see Eq. 3-2). These partial derivatives are typically evaluated using finite differences, since the performance function is not usually available in close form. For example, the performance function may be in terms of the stress of a particular member, which must be obtained from a finite element analysis.

The partial derivative of the performance function with respect to each random variable must be evaluated on each iteration. Hence, the number of performance function evaluations is proportional to the product of the number of random variables and the number of iterations. If two-sided finite differences are used, then evaluation of each partial derivative requires two performance function evaluations; whereas, if one-sided finite differences are used (which will be more expedient but less accurate) then each partial derivative requires one performance function evaluation.

Since the iterations must be carried out serially, only the performance function evaluations for a single iteration can be carried out in parallel. Hence, the degree of parallelism is related to the number of random variables. The greater the number of random variables, the greater will be the percentage of the calculations that can be carried out in parallel. If the number of random variables is not significantly larger than the number of processors, it will be difficult to keep all processors busy during the concurrent operations. There are several reasons for this: (1) it is unlikely that the number of random variables will be an even multiple of the number of processors; (2) the length of time required to compute the individual partial derivatives may be different for the different variables; and (3) it is not possible to start a new iteration until all partial derivative evaluations from the current iteration are complete. Hence, for some problems it may be difficult to achieve high efficiency by taking advantage of this parallelism. Note, however, that if second order methods are used, the number of performance function evaluations per iteration will increase significantly, thereby, increasing the parallelism. Thus, second order methods may be more suitable for parallel implementation.

There are two other related sources of parallelism in partial derivative methods that can be utilized to achieve greater speedups. These occur in evaluating different performance functions, and in developing the cumulative distribution functions (CDF).

When developing the complete CDF, it is necessary to compute the non-exceedance probability for different response levels. This is illustrated in Figure 3-3. Here, we could allocate different groups of processors to different response level calculations. Within each group of processors, different processors perform the finite difference evaluations for different variables. In such cases the degree of parallelism is significantly increased, and the processor idling time identified above can be reduced.

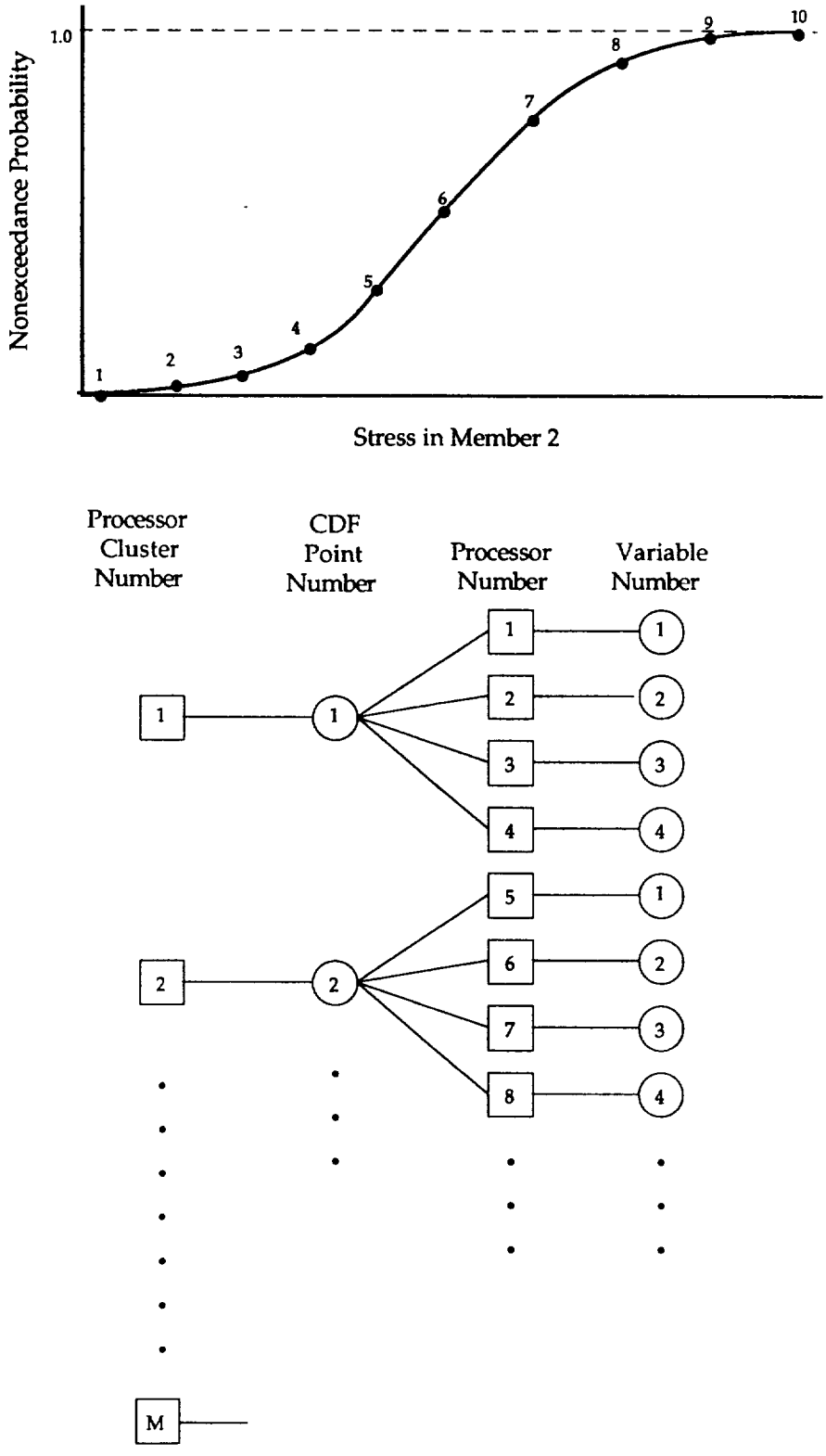


Figure 3-3. Implementation of Multi-Level Parallelism for Partial Derivative Based PSM Method (M Processor Clusters)

An example of a case where different performance functions are evaluated would be when evaluating the CDF for stresses in more than one part of a structure. Another example would be a case in which different failure modes must be evaluated (*e.g.*, demonstration of reliability for stress criteria, deflection criteria, or stability criteria). In effect this is identical to the problem of multiple response level calculations, and multiple processors are assigned to solve each performance function.

Monte-Carlo Simulation. Parallelism in Monte-Carlo simulation results from the repeated independent evaluations of the performance function. The number of performance function evaluations required will be controlled by the accuracy required for the tails of the response distribution and can typically range from the order of thousands to tens of thousands. If only first and second order response statistics are required (mean and standard deviation) then the number of performance function evaluations required will be significantly reduced (typically less than 100 will be required).

We note that the other sources of parallelism that arise in the partial derivative methods for evaluation of the entire CDF and different performance functions are not present here. In Monte-Carlo simulation, the additional work required to obtain one point on the CDF, or the entire CDF, is negligible. It is only necessary to keep track of the score for each desired point as each history in the simulation is completed. Similarly, it is a trivial matter to evaluate the response CDF for different parts of the structure by scoring the response at different locations. An exception would be when multiple failure modes must be considered and when these failure modes require different types of analyses. For example, member checking for overstress *vs.* checking for stability, *vs.* checking for a frequency shift.

Other Methods. There are several other methods that can be used for PSM problems, as mentioned earlier. These methods exhibit parallelism similar to that described above, but to different degrees. For Monte-Carlo simulation with variance reduction, the degree of parallelism is reduced since the number of performance function evaluations required is reduced. However, with some variance reduction methods, such as importance sampling, when different performance functions must be evaluated, different sampling strategies, and hence, different simulations must be performed (analogous to the extra effort involved in partial derivative methods for multiple performance functions). Thus, the parallelism of Figure 3-3 is relevant (while, as pointed out above, it is not relevant for direct Monte-Carlo simulation). For hybrid methods wherein either sampling, partial derivative, or perturbation methods are used to develop a response surface, essentially all of the sources of parallelism described above are present. Development of the response surface by any of the methods will require multiple evaluations of the performance function. Also, the use of different performance functions will usually require a different response surface evaluation.

Summary. Table 3-1 summarizes the sources of parallelism in various PSM methods.

Table 3-1. Sources of Parallelism in Various PSM Methods

Method	Repeated Performance Function Evaluations for Perturbed Inputs	Multiple CDF Values	Multiple Failure Mode Analysis	Different Structural Response Locations of Interest
FORM/SORM	X	X	X	X
Direct Monte-Carlo	X		X[1]	
Monte-Carlo w/ Variance Reduction	X	X	X	X
Hybrid	X	X	X	X

[1]Only when different analysis model or method is used for different failure modes

3.3 PARALLELISM IN STRUCTURAL MECHANICS COMPUTATIONS

Many sources of parallelism exist in structural mechanics computations; and techniques to take advantage of this parallelism have been the subject of a significant amount of research for the past several years. We briefly present here some of the strategies developed for taking advantage of parallelism in structural mechanics computations. It is likely that it will be necessary to use these strategies in conjunction with those presented above to optimally exploit parallel processing for PSM problems.

There are essentially three strategies that have been used for implementing structural mechanics problems, in particular finite element methods, on parallel processing computers. These include: (1) substructuring; (2) domain decomposition; and (3) operator splitting.

Substructuring. The substructuring approach has been common in finite element analysis for some time [Przemieniecki, J., 1963]. In this approach the structure is broken down into substructures and solved as an assemblage of superelements that relate forces and displacements at the boundaries of the superelement or substructure. Substructuring techniques were developed in order to break large structural problems into smaller, more manageable problems (of particular importance when memory is limited) and to take advantage of cases when structures are composed of replicating units. The application for parallel processing is evident. Independent processors can work on the independent substructures and development of the superelements. Once the superelements are formed the structure must be assembled, and then solved on a single processor or solved on multiple processors using an operator splitting technique (see below).

Substructuring is relevant for parallel implementation of PSM problems since memory requirements are large for concurrent performance function evaluations. Taking advantage of both substructuring parallelism and the probabilistic parallelism will make large PSM problems more manageable and increase the parallel processing efficiency.

Domain Decomposition. Domain decomposition is similar to substructuring in that the structure is broken down into sub-regions or sub-domains. It differs, however, in that superelements are not created and the complete structure is not actually assembled. Rather, each sub-domain is solved as an independent initial/boundary-value problem. An example of domain decomposition, using an approach to decompose an irregular grid to achieve a balanced workload while minimizing the number of interface nodes is shown in Figure 3-4 [Farhat, *et al.*, 1987]. Since the solution at the sub-domain interfaces is unknown, the individual sub-domain solutions must be iterated until the interface solutions converge. The advantage of this method for parallel implementation is that it is not necessary to finally assemble the entire structure for solution on a single processor. A recent review of domain decomposition methods can be found in Chan, *et al.* [1989].

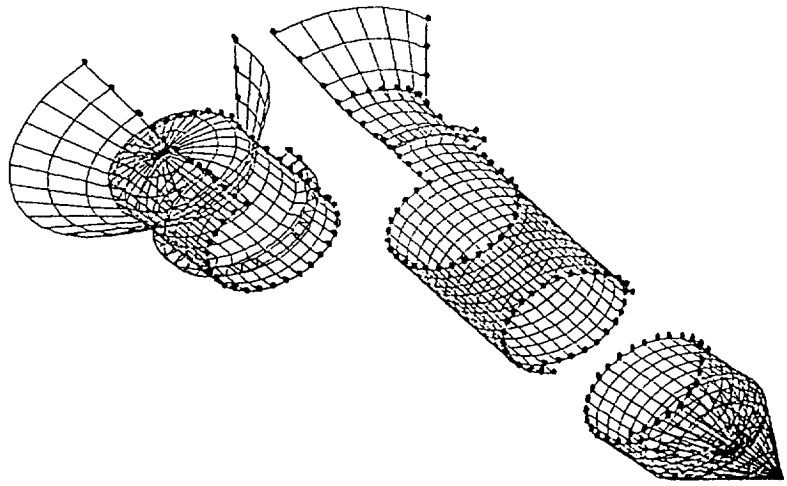


Figure 3-4. Decomposition of Irregular Grid Into Three Subdomains (after Farhat, *et al.* [1987].

Operator Splitting. Operator splitting, in general, refers to the reduction of the solution of a complex problem into the solution of several simpler problems. Hence, essentially any parallel processing implementation is a form of operator splitting. We use the term operator splitting herein to refer to numerical splitting techniques that may not have a physical interpretation, as for the strategies previously presented.

Iterative equation solvers, wherein only matrix multiplications are required during the solution process [Fox, *et al*, 1988], are one such approach (*e.g.*, the conjugate gradient method). In general iterative solvers will not be as fast as direct method solvers (*i.e.*, Gauss elimination or Cholesky decomposition), however, since

only matrix multiplications are required it is straightforward to break up the numerical effort among available processors. Iterative equation solvers solve the structural equations using an initial, assumed solution that is updated on subsequent iterations until convergence is achieved. These techniques are also well suited to PSM problems, wherein the structure must be repeatedly solved with slightly perturbed input values. Hence, they are of particular interest herein and are further discussed in Chapter 5 and in Appendix A.

It is also possible to use a number of techniques to decompose direct solvers for application on parallel processing architectures. A recent application of the Cholesky decomposition method by Agarwal, Storaasli, and Nguyen [1990] achieved speedups ranging from 5 to 7 using 8 processors on a Cray Y-MP.

3.4 SUMMARY

There are a number of sources and levels of parallelism in both the probabilistic computations and the structural mechanics computations for PSM problems. The sources of parallelism in the probabilistic computations were summarized in Table 3-1 and it was shown that a high degree of parallelism is inherent in all commonly used PSM methods. Techniques for taking advantage of the parallelism in structural mechanics computations were briefly reviewed in Section 3.3.

Due to the limited scope of this Phase I effort, the review of parallelism in the structural mechanics computations covered only currently used techniques. A thrust of the Phase II effort will be identification of additional levels of parallelism and how these can be implemented in parallel PSM applications. For example, in many structural applications, different degrees of modeling detail are required for different parts of a structure, such as in a crack propagation problem. Similarly different levels of detail in the treatment of uncertainties are also required. Assigning different processors to work on different parts of the structure and the corresponding uncertainties presents a new challenge in parallel processing.

CHAPTER 4

MCPAP: A MONTE-CARLO SIMULATION CODE FOR PSM PROBLEMS ON A MULTIPROCESSOR COMPUTER

4.1 INTRODUCTION

In order to demonstrate the feasibility of implementing a PSM code on a parallel processing computer, and to study the speedups and efficiencies obtainable, a parallel PSM code was developed and implemented on an Alliant FX/80. The Alliant FX/80 is a shared memory parallel processing computer with eight 64-bit vector pipeline processors. Its architecture was described in Chapter 2. The code developed for this effort, MCPAP, employs the Monte-Carlo simulation method described in the previous chapter. Monte-Carlo simulation was selected for this demonstration since it is the most readily adapted method to the parallel processing environment (see Chapter 3). Monte-Carlo simulation is also the method of choice in many instances (*e.g.*, when the number of problem variables is large and the first few statistical moments of the response are of interest, and when multiple performance functions must be evaluated). Current and future developments in parallel processing have the potential to make Monte-Carlo simulation a very practical PSM method.

In this chapter, we present an overview of MCPAP and discuss some of the coding required for parallel implementation. In particular we treat the problem of random number generation and scoring in parallel. This is a unique problem for parallel processing since these operations are not independent from simulation trial to trial (in contrast to evaluation of the performance function, which is independent from trial to trial). The programming details on a parallel processing computer are not presented; for more information the reader may consult any one of a number of texts on this subject (*e.g.*, [Fox, *et al.*, 1988]). The results of three example applications are also given: (1) a cantilever beam problem with a closed form solution; (2) a two-tier truss finite element problem; and (3) a 3-D space truss finite element problem.

4.2 OVERVIEW OF MCPAP

MCPAP is a direct Monte Carlo simulation shell for structural mechanics applications with a library of random variable generators for ten commonly used distributions. Generation of correlated random variables is handled through the Nataf model transformation described in the previous chapter. Multiple scoring is also facilitated, allowing the code to analyze multiple performance functions simultaneously.

Two program modules are supplied by the user: XLIMIT is for calculating the values of performance functions for a given set of sample variables, and PLIMIT is for I/O and preprocessing the necessary data for those calculations. The latter will be

executed only once during the simulation run, while the former will be called repeatedly for as many times as the number of simulation trials. A finite element code is often incorporated in these two modules for PSM. As such, the system configuration including the information on nodal points, elements, and applied forces, and necessary information on storage addresses for assembling the system stiffness matrix are input and prepared in PLIMIT. The actions for assembling the system stiffness matrix and carrying out the responses for a given set of sample variables in each simulation trial are performed in XLIMIT.

The input data for MCPAP contains the statistical descriptions of the problem random variables including means, standard deviations, bounds, and correlation coefficients, and simulation control parameters such as number of trials, number of performance functions, and seed for random number generations. The output contains the event probability and statistical error of simulation for each performance function. Figure 4-1 illustrates the general flow chart for MCPAP. The vertical parallel lines indicate segments of the code that are executed in parallel, that is, concurrently using all available processors.

4.3 PARALLEL IMPLEMENTATION ON THE ALLIANT FX/80

To implement a computer code on a parallel computer, it is first necessary to identify the parallelism in the code, and then determine which parts will be automatically parallelized by the compiler and which parts require recoding. The Fortran compiler on the Alliant FX/80 will attempt to automatically optimize DO loops for concurrency, and array operations for vectorization. This automatic parallelization is briefly reviewed below, followed by a discussion of the specific parallelization of MCPAP.

4.3.1 Parallel Code Construction

DO loops will be automatically optimized for concurrency (that is, executed simultaneously), if calculations in different iterations of the same loop can be executed independently. In order to do this, the compiler first checks for data dependency between loop iterations. If a dependency is found, the loop will not be executed in the concurrent mode. Also, if a DO loop contains certain statements, such as a subroutine call, the compiler will not automatically optimize the loop. One reason for this is that it is not possible for the compiler to determine if the subroutine call in a certain loop iteration requires data from a previous iteration (which could be executing at the same time on another processor). Similarly, in order for array operations within a DO loop to be vectorized, there should be no dependencies between iterations.

A number of programming techniques have been developed to avoid data dependencies to allow automatic optimization to occur. Also, it is possible to invoke parallelization through the use of optimization directives embedded in the

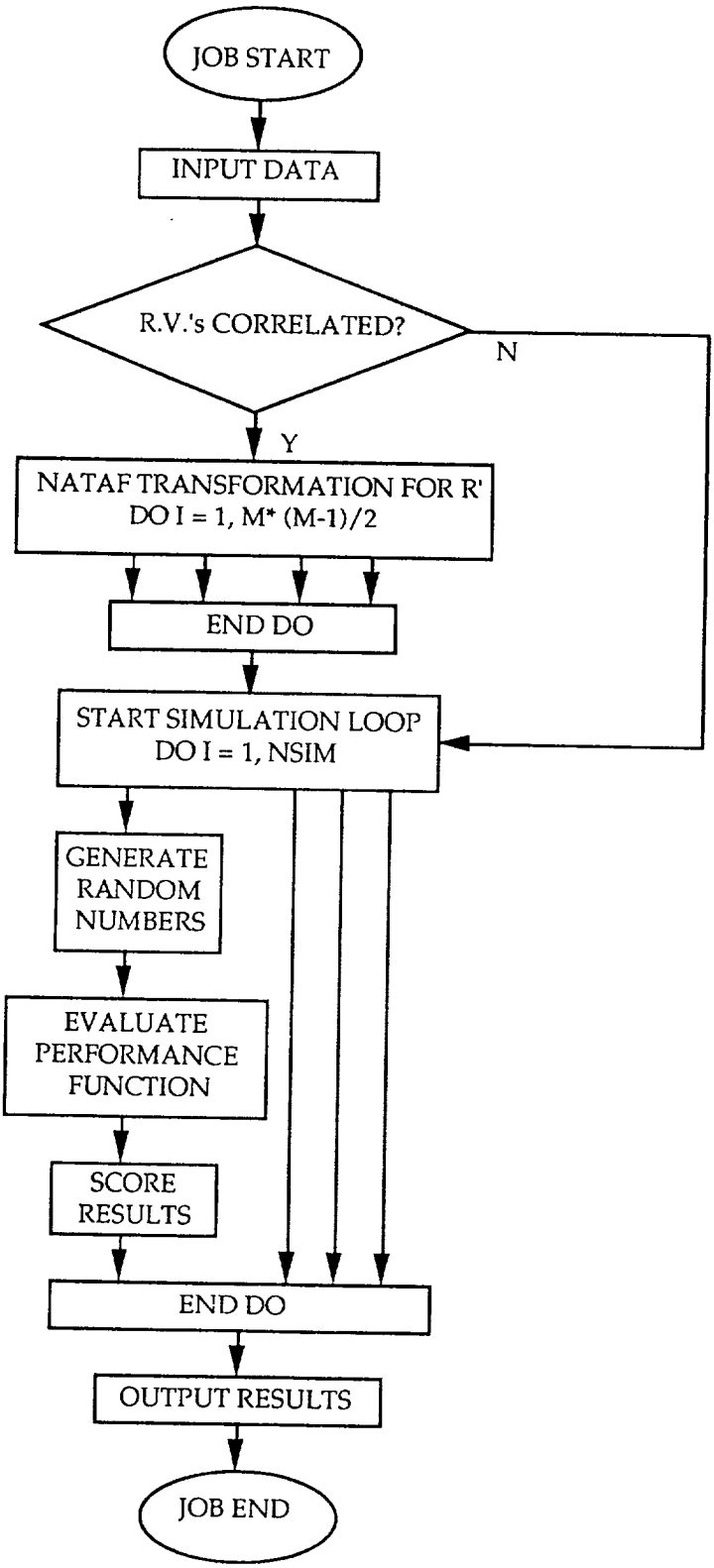


Figure 4-1. Code Implementation on Alliant FX/80

code. Discussions of these techniques can be found in a number of texts on parallel processing, and we will only describe here one particular technique, of key significance to the parallel implementation of Monte-Carlo simulation. If a portion of a code is subject to a number of independent replications, it can be gathered into a subroutine and called from inside a DO loop. The DO loop can then be optimized for concurrent operation by preceeding the loop with a concurrent call directive, and declaring the subroutine to be recursive. For example,

```
      PROGRAM MAIN
      • • •
CVD$  CNCALL
      DO I = 1, N
        CALL SUB ( A, B, C)
      END DO
      • • •
      END

      RECURSIVE SUBROUTINE SUB ( A, B, C)
        •
        •
        •
      END
```

where CVD$ CNCALL is the concurrent call directive for the Alliant. By declaring the subroutine to be recursive, each time the subroutine is called, storage is allocated for a unique copy of the subprogram's local variables. Conversely, variables passed through the argument list or in a common block are treated as shared variables. For each iteration of the loop, a processor will be allocated to execute the operations inside the subroutine. The Alliant FX/80 dynamically allocates the processors for concurrent loop operations, so that as soon as a processor has completed execution of the subroutine for one loop iteration, it is assigned to execute the subroutine for another iteration.

In MCPAP local DO loops have been optimized and two major parts of the code have also been optimized for concurrency by grouping them into recursive subroutines. Within each subroutine, the code is vectorized, adding an additional level of parallelism to maximize efficiency. Also, within the subroutines, automatic compiler concurrency is suppressed, since there is no advantage to executing a concurrent operation within a subroutine when the subroutine is executing on other processors concurrently. This is particularly true for MCPAP on the Alliant FX/80 since the granularity of the subroutine is large and it is executed many more times than the number of available processors.

The first parallelized code segment is the procedure for the Nataf space transformation. In this transformation, the calculations of the modified correlation coefficient for any pair of random variables can be carried out independently.

Hence, elements of the modified correlation coefficient matrix R' can be calculated concurrently.

The second parallelized code segment is for the execution of the simulation loop as shown in the flow chart. Procedures including sampling, performance function evaluation, and scoring are all controlled by one master subroutine which is declared to be recursive. The repetitive executions of this subroutine are dynamically allocated to the multiple processors. That is, as soon as a processor is free a new simulation history is allocated to the processor. This continues until all simulation histories have been allocated. For example, if 1,000 histories are to be performed, histories are dynamically allocated to the processors until the $1{,}000^{th}$ history is *begun*. This strategy minimizes processor idle time without biasing results. A slightly more efficient strategy would be to continue allocation until the $1{,}000^{th}$ history is *complete*. This strategy is not used, however, since it would bias results to shorter executing histories.

Note that for parallel implementation, all random variables must be defined as local variables so that a unique copy of these variables will be maintained for each concurrently executing subprogram. For example, when the problem is a finite element analysis wherein the structure properties are random, the stiffness matrix is defined as a local array by dimensioning it within the recursive subroutine. In this way each processor will operate on a unique copy of the structure stiffness matrix. For large structures, maintaining multiple copies of the stiffness matrix can put a heavy demand on available memory as will be discussed later in this chapter.

Conversely, deterministic problem variables can be passed through the subroutine argument list or maintained in a common block, to be shared by all concurrently executing processes in order to minimize memory requirements, and maximize computational efficiency. For example, if in a particular problem only the loading variables are random, the structure stiffness matrix need only be formed once, and one copy may be shared by all concurrent processes.

4.3.2 Parallel Random Number Generation and Scoring

In direct Monte-Carlo simulation, evaluation of the performance function (Figure 4-1) is independent from trial to trial. However, generation of random numbers and scoring are not. Hence, special strategies are required to enable parallel implementation.

Pseudo-random numbers are generated in MCPAP by the mixed congruential method (see, *e.g.*, Knuth, 1973). It is defined by

$$X_{n+1} = (aX_n + c) \bmod m \tag{4-1}$$

$$R_{n+1} = X_{n+1} / m \tag{4-2}$$

in which X_i is the i^{th} random number of the sequence (or stream), R_i is the output random number which is uniformly distributed, and a, c, and m are chosen in order to make the random numbers as random as possible. As can be seen from Eqs. 4-1 and 4-2, each random number generation has to rely on the previously generated number in the same stream. Concurrent execution of such recursive procedures on multiple processors can cause duplication of random number sequences. Several strategies for random number generation are possible to avoid this problem and maintain parallel independent streams of random numbers.

The first possible strategy is to generate the whole set of random variables for the total number of simulation trials prior to the simulation loop. However, memory requirements for storing the entire set of random samples can easily become prohibitive for practical PSM problems. To prevent this an alternative is to generate the random variables for a fraction of the total number of simulations, execute these simulations, and then generate another set of random variables. This approach makes memory requirements more practical, at the expense of increased overhead due to processor idling.

A second strategy involves generating the random numbers within the simulation DO loop so that they are generated concurrently on different processors. An approach that results in generation of exactly the same stream of random numbers as would be generated on a serial computer is presented by Fox, *et al.* [1988], based on work of Frederickson [1983], Brown [1983], and Barkai [1984]. Employed in this approach is the fundamental relation between the $n+k^{th}$ and n^{th} random numbers for the congruential random number generator:

$$X_{n+k} = (AX_n + C) \bmod m \tag{4-3}$$

with

$$A = a^k \tag{4-4}$$

and

$$C = 1+a+a^2+ \cdots +a^{k-1} = \frac{a^k-1}{a-1} \tag{4-5}$$

Note that A and C need to be calculated only once and stored. Then, if there are NP processors, the sequence of, for example, the first three random numbers generated on each processor can be listed as (subscripts denote position in the random number sequence, superscripts denote processor number, and Y denotes the random number that would be generated on a serial computer):

$$X_0^{(0)} = Y_0$$
$$X_0^{(1)} = (aY_0 + c) \bmod m = Y_1$$

$$\vdots$$

$$X_0^{(NP-1)} = (a\, Y_{NP-2} + c) \bmod m = Y_{NP-1},$$

$$X_1^{(0)} = Y_{0+NP}$$
$$X_1^{(1)} = Y_{1+NP}$$
$$\vdots$$
$$X_1^{(NP-1)} = Y_{NP-1+NP},$$

and

$$X_2^{(0)} = Y_{0+2NP}$$
$$X_2^{(1)} = Y_{1+2NP}$$
$$\vdots$$
$$X_2^{(NP-1)} = Y_{NP-1+2NP}$$

The method is illustrated in Figure 4-2 for the case of $NP = 4$. As the figure demonstrates, the parallel processors use a staggered start and then leapfrog using Equation 4-3. As mentioned above, this strategy is advantageous since it generates exactly the same stream of random numbers as a serial random number generator and the Monte-Carlo results are repeatable. Since individual simulation histories in PSM problems can take different lengths of time to execute, all random numbers should be generated at the beginning of the history and prior to the beginning of any structural mechanics computations. This ensures that when the k^{th} history is begun the random numbers for the $k - NP^{th}$ history have already been generated. The use of local variables in the structural mechanics computations, ensures that the k^{th} history calculations cannot corrupt the values used in the $k - NP^{th}$ history.

A third strategy developed and implemented herein is to establish an independent stream of random numbers for each processor. Random seeds are generated for each processor using a different random number generator. On each invocation of a simulation trial, a function call to the Alliant's system library function LIB_PROCESSOR_NUMBER is first executed which returns the processor number of the processor that is allocated to the particular trial. The processor number is used to fetch the previous set of random numbers generated by this processor, which is then used to generate the next set of random numbers. By doing so, in the concurrent process of the simulation trials, each processor only picks up and works on its own sample stream and therefore no synchronization among the concurrent trials is necessary.

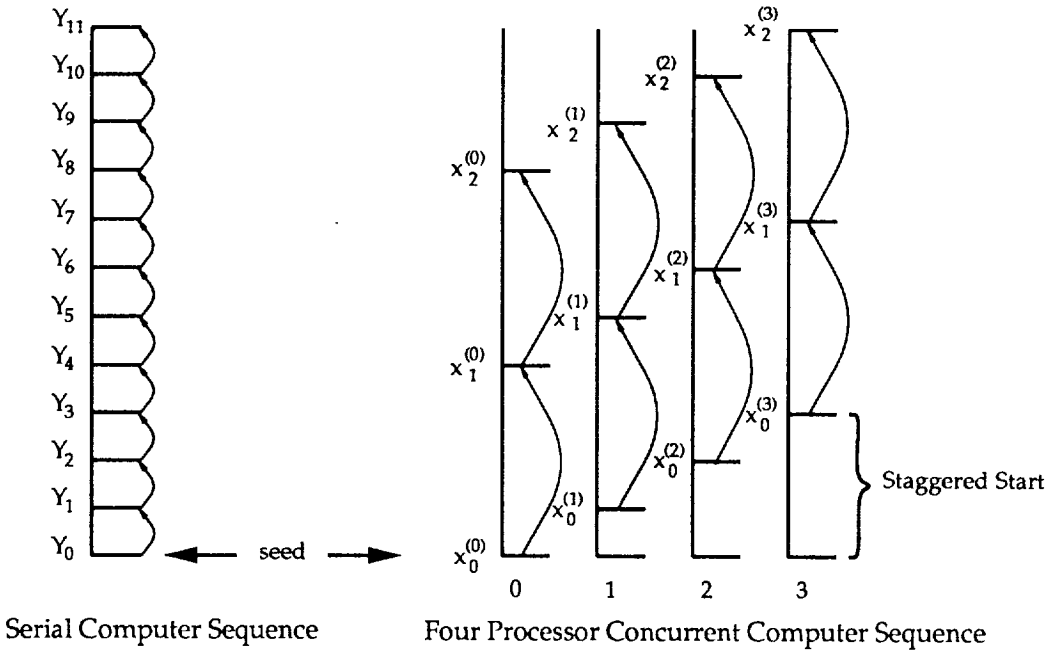


Figure 4-2. Strategy for Generating an Identical Sequence of Random Numbers on Sequential and Concurrent Computers

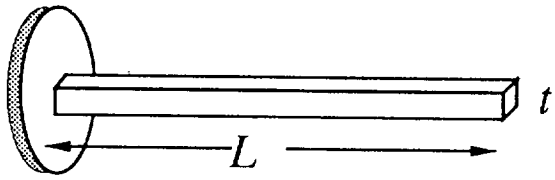
As mentioned earlier, another numerical dependence between simulation trials is in the simulation results scoring. In the conventional serial algorithm, a carry-around scalar is used in the simulation loop to keep track of the scoring in each simulation trial. That is, the running sum of the results of the simulation trial (0 for safe, 1 for failure) is stored in the scoring variable and modified after completion of each history. Scoring in this manner on a parallel processor can lead to an erroneous score since different processors may attempt to update the score at the same time. To resolve this, a similar strategy to that of random number generation is employed in our code. That is, each processor has its own designated score board (*i.e.*, a carry-around scalar). The final total score is then the sum of the score on each processor.

4.4 CANTILEVER BEAM EXAMPLE

Figure 4-3 illustrates this simple first example. The selected problem is to evaluate the cumulative distribution function (CDF) of the fundamental frequency of the cantilever beam, wherein the properties of the cantilever beam are random variables. The fundamental frequency of the rectangular beam is given by

$$\omega = 3.52 \sqrt{\frac{E t^2}{12 \rho L^4}} \qquad (4\text{-}1)$$

where E is the modulus of elasticity, ρ is the material density, t is the beam depth, and L is the beam length. Each of the beam properties is assumed to be a lognormal random variable with median and coefficient of variation given in Table 4-1.



E = modulus of elasticity
ρ = material density
t = thickness
L = length

Figure 4-3. Cantilever Beam Example

Table 4-1. Random Variables for Cantilever Beam Example

Variable	Type	Median	COV
E	lognormal	10^7 (*psi*)	0.03
ρ	lognormal	2.5×10^{-4} ($lb\text{-}sec^2/in^4$)	0.05
t	lognormal	0.98 (*in*)	0.05
L	lognormal	20.0 (*in*)	0.05

The lognormal distribution is selected for this illustration since this allows for an exact closed form evaluation of the CDF for the fundamental frequency. When the beam properties are lognormal, the fundamental frequency is also a lognormal random variable and the median frequency (denoted here by ^) is given by

$$\hat{\omega} = 3.52 \sqrt{\frac{\hat{E}\,\hat{t}^2}{12\,\hat{\rho}\,\hat{L}^4}} \tag{4-2}$$

Also, the logarithmic standard deviation (denoted as ξ) of the fundamental frequency is given by

$$\xi^2_{\omega} = (1/2)^2 \left[\xi^2_E + 2^2\,\xi^2_t + \xi^2_{\rho} + 4^2\,\xi^2_L \right] \tag{4-3}$$

assuming independence among the problem variables. From the relationship between the logarithmic standard deviation and the coefficient of variation for a lognormal random variable, the coefficient of variation for the beam fundamental frequency can be obtained as

$$\delta_w = 1 - e^{\xi^2_w} \tag{4-4}$$

The CDF for the beam frequency, $F(w)$, is given by

$$F(W) = \Phi\left(\frac{\ln w - \ln \widehat{w}}{\xi_w}\right) = \Phi\left(\frac{\ln \frac{w}{\widehat{w}}}{\xi_w}\right) \tag{4-5}$$

where Φ is the standard normal CDF.

For the simulation, the CDF for the beam fundamental frequency was evaluated by coding the frequency expression, Eq. 4-1, into the performance function subroutine of MCPAP. On each simulation history, the frequency is evaluated and compared with several fixed values. The number of times each fixed value is exceeded (over all the simulation histories) is scored by MCPAP to determine the exceedance (or non-exceedance) probability of each fixed value. Note that for each simulation history the frequency function need only be evaluated once. This is one advantage of the simulation approach when the performance function is complex as in most practical problems.

For this analysis 10,000 Monte-Carlo histories were used to obtain the CDF. This large number of histories ensures reliable results for the range of probability levels considered. Figure 4-4 shows the results of the analysis using 1, 4, 6, and 8 processors of the Alliant FX/80. As can be seen the comparison between the exact solution and the Monte-Carlo simulation is quite good.

Note that the results using different numbers of processors vary slightly. This is because, as explained in the previous chapter, each processor uses an independent stream of random numbers which is begun with a different random seed. Although it is not possible to detect from the figure, there is no systematic trend to the results with the number of processors. That is, the results do not systematically increase or decrease with the number of processors. This is to be expected and is heuristic evidence of the fidelity of the random number generation approach. The difference in results, for different numbers of processors, becomes more evident in the tail of the distribution which is to be expected in Monte-Carlo simulation. The results will converge to the exact solution as the number of histories is increased.

Table 4-2 lists the speedup and efficiency obtained for 1, 4, 6, and 8 processors, while Figure 4-5 displays these results graphically. For this problem with 10,000 histories, the fraction of the computations that cannot be performed concurrently (α) is 1.38%. This is essentially program I/O. The theoretical speedup (*i.e.*, speedup assuming no concurrency overhead) is obtained from Eq. 2-7 and the actual speedup is the observed speedup on the Alliant. It may be noted that the maximum theoretical speedup from Eq. 2-7 (assuming an infinite number of processors) is 72.5.

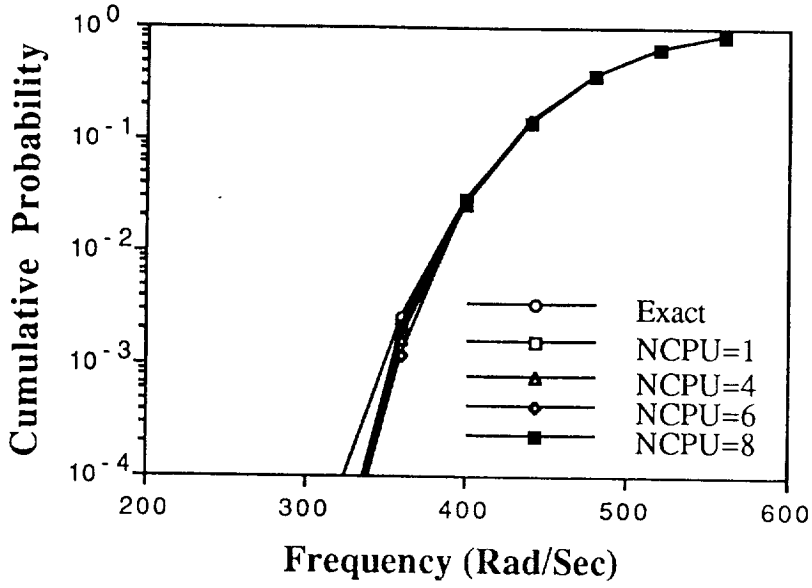


Figure 4-4. CDF for Cantilever Beam Fundamental Frequency

Table 4-2. Speedup and Efficiency for Cantilever Beam Example (Sample Size = 10,000, $\alpha = 0.0138$)

N (CPU)	Theoretical Speedup	Actual Speedup	Overhead Factor	Efficiency (%)
1	1.0	1.0	0.0	100.0
4	3.84	3.76	0.0056	97.9
6	5.61	5.40	0.0070	96.3
8	7.30	6.81	0.0098	93.3

The overhead factor is obtained from Eq. 2-12 and, as defined in Chapter 2, is the additional processor time required to support the concurrent operations, expressed as a fraction of the time required to solve the problem on a single processor.

The results show that the efficiency is high, exceeding 93% for 8 processors. The observed decrease in efficiency as the number of processors increases is because the overhead increases with the number of processors as expected. Actually, since the performance function computation for this problem is so trivial (*i.e.*, Eq. 4-1), there is a very small amount of computing done on each Monte-Carlo history (*i.e.*, the problem granularity is fine). Thus, only a relatively small amount of time passes before the processor completes a history and another history must be allocated to the processor. For this reason the overhead factor may be smaller for more complex problems. This is explored in the examples that follow. This phenomenon affects the number of processors that can effectively be used, and impacts decisions regarding optimal hardware configurations for PSM problems.

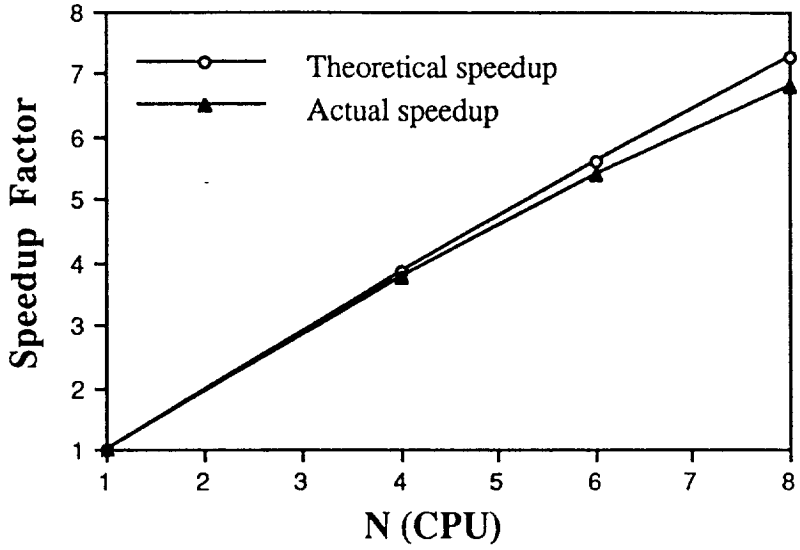


Figure 4-5. Theoretical and Actual Speedups for Cantilever Beam Example

4.5 FINITE ELEMENT EXAMPLES

As described earlier, a finite element code was implemented into MCPAP in order to study speedup and efficiency for more complex problems. Two example problems were conducted, a two-tier 2-dimensional truss and a 3-D space truss.

4.5.1 Two-Tier Truss

Figure 4-6 shows the two tier truss problem, and Tables 4-3 and 4-4 describe the material properties. The problem random variables are the member elastic moduli, the member cross sectional areas, the initial strain in the members and the loadings S_1 and S_2. The truss has ten members and six independent material types. The members that have the same material type have perfectly dependent properties. Values shown in the table are the mean properties for each material type. Note that while the mean moduli are the same for different material types, they are independent random variables, so that on a given Monte-Carlo history the values will not be the same. The modulus of elasticity, cross sectional area, and loading are modeled as lognormal random variables with the coefficients of variation (δ) as shown in Table 4-3. The initial strain random variable accounts for possible initial deformation in members due to fabrication and construction tolerances, and is modeled as a normal random variable with zero mean and standard deviation of 10^{-4}. As for the material moduli, the initial strains in members of different material types will not be the same on a given Monte-Carlo history.

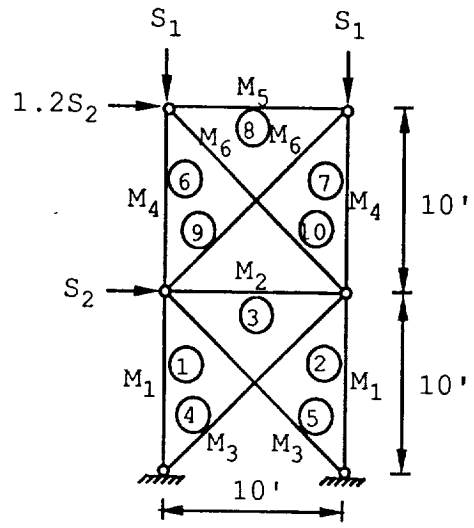


Figure 4-6. Two-Tier Truss Example

Table 4-3. Material Property Random Variables for Two-Tier Truss Example

Material Type	E (δ=0.15) (*ksi*)	A (δ=0.10) (*sq. in*)	ε_{int} (σ=10^{-4})
M1	29000.	3.3334	0.
M2	29000.	0.3334	0.
M3	29000.	1.7778	0.
M4	29000.	0.7778	0.
M5	29000.	0.5556	0.
M6	29000.	1.1112	0.

E = mean modulus of elasticity (lognormal r.v.)
A = mean bar cross-section area (lognormal r.v.)
ε_{int} = mean initial strain in bar element (normal r.v.)
δ = coefficient of variation
σ = standard deviation

Table 4-4. Loading Random Variables for Two-Tier Truss Example

Load	Type	mean (*kips*)	δ
S1	Lognormal	20.0	0.30
S2	Lognormal	60.0	0.30

Figure 4-7 shows the results of the analysis using 1, 4, 6, and 8 processors of the Alliant FX/80. As for the cantilever example, the results using different numbers of processors vary only slightly, and result from each processor using an independent stream of random numbers. Also, it is again observed that there is no systematic trend to the results with the number of processors, as expected.

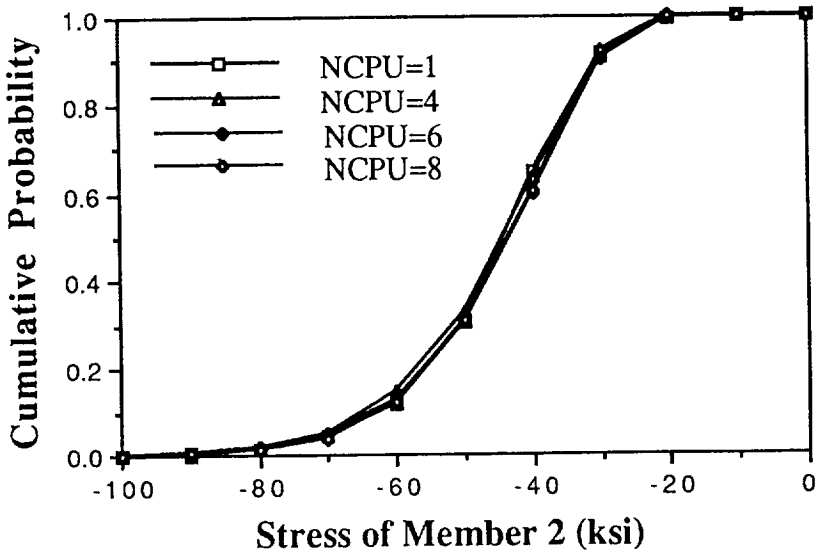


Figure 4-7. CDF for Stress in Member 2

The speedup and efficiency obtained for 1, 4, 6, and 8 processors was evaluated as shown in Table 4-5 and Figure 4-8. For this problem 1,000 Monte-Carlo histories were performed and the fraction of the computations that cannot be performed concurrently (α) was evaluated to be 3.37%. Note that α is larger here than in the previous example, because of the lesser number of Monte-Carlo histories. If the number of histories were increased, the fraction of the computations (measured by execution time) that can be performed would increase and α would decrease. Again, the theoretical speedup (*i.e.*, speedup assuming no concurrency overhead) is obtained from Eq. 2-7 and the actual speedup is the observed speedup on the Alliant. The overhead factor, obtained from Eq. 2-12, is also shown in Table 4-5. Note that the overhead factors are significantly smaller, and the efficiencies higher, for this problem as compared with the cantilever beam example. This is to be expected, since, for this problem, the computation time for each Monte-Carlo history is much larger (*i.e.*, the granularity of the problem is larger). Thus, a larger amount of computation time passes before another history must be allocated to the processor.

Table 4-5. Speedup and Efficiency for Two-Tier Truss Example (Sample Size = 1,000, $\alpha = 0.0337$)

N (CPU)	Theoretical Speedup	Actual Speedup	Overhead Factor	Efficiency (%)
1	1.0	1.0	0.0	100.0
4	3.63	3.62	0.0010	99.7
6	5.13	5.05	0.0033	98.4
8	6.47	6.26	0.0052	96.8

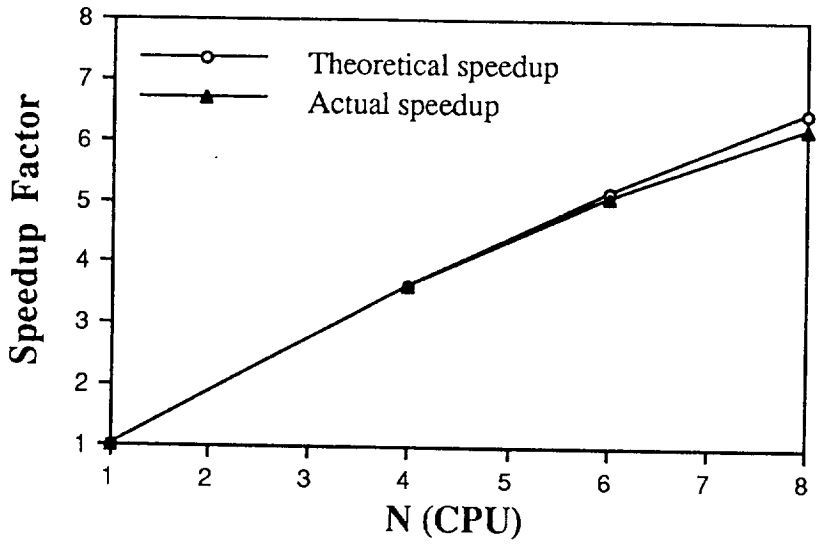


Figure 4-8. Theoretical and Actual Speedups for Two-Tier Truss Example

4.5.2 3-D Space Truss

The next example considered is a 3-D space truss composed of 99 members with 72 degrees of freedom. The purpose of this problem is to examine the effect on the overhead and efficiency as the problem size increases. This has important practical significance since the overhead factor limits the maximum speedup that is achievable. Figure 4-9 shows the front panel and section details of the truss. The truss is made up of three panels so that the cross section is an equilateral triangle. It is simply supported at three points on the bottom and is capped by a pyramid at the top. The apex of the pyramid is circumferentially constrained so that only vertical movement is possible. Three loads are applied to the structure, a vertical load at the apex and two horizontal loads The problem random variables are the same as for the two-tier truss: the member elastic moduli, the member cross sectional areas, the initial strain in the members and the loadings S_1, S_2, and S_3. Satistical descriptions of the random variables are given in Tables 4-6 and 4-7.

The speedup and efficiency in computing the CDF for element stress, obtained for 1, 4, 6, and 8 processors was evaluated as shown in Table 4-8. As for the two-tier truss, 1,000 Monte-Carlo histories were performed and the fraction of the computations that cannot be performed concurrently (α) is evaluated to be 0.5%. Note that α is much smaller here than in the two-tier truss example. This results because the time to evaluate the performance function (*i.e.*, solve the structure) which is the concurrent part of the Monte-Carlo simulation, has significantly increased. As for the earlier examples, the theoretical speedup (*i.e.*, speedup

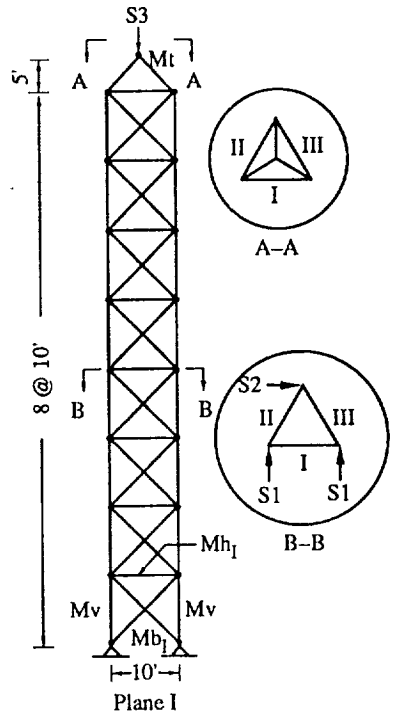


Figure 4-9. Front Panel of 3-D Space Truss Example

Table 4-6. Material Property Random Variables for 3-D Truss

Material Type	E (δ=0.10) (*ksi*)	A (δ=0.10) (sq. *in*)	ε_{int} (σ=10^{-4})
Mv	29000.	1.590	0.
Mh_I	29000.	1.590	0.
Mh_{II}	29000.	1.590	0.
Mh_{III}	29000.	1.590	0.
Mb_I	29000.	0.938	0.
Mb_{II}	29000.	0.938	0.
Mb_{III}	29000.	0.938	0.
Mt	29000.	1.590	0.

E = mean modulus of elasticity (lognormal r.v.)
A = mean bar cross-section area (lognormal r.v.)
ε_{int} = mean initial strain in bar element (normal r.v.)
δ = coefficient of variation
σ = standard deviation

Table 4-7. Loading Random Variables for 3-D Truss

Load	Type	mean (*kips*)	δ
S_1	Lognormal	10.0	0.25
S_2	Lognormal	10.0	0.25
S_3	Lognormal	500.0	0.25

assuming no concurrency overhead) is obtained from Eq. 2-7 and the actual speedup is the observed speedup on the Alliant. The overhead factor, obtained from Eq. 2-12, is also shown in the Table. Again, the efficiencies achieved are quite high.

It is interesting to note that the overhead is generally larger, and the efficiency smaller, for this problem than the two-tier truss, although the granularity of this problem is larger than that of the two-tier truss. One possible explanation is that there is significantly more memory access required for solving the 3-D truss problem. Hence, as the number of processors is increased memory contention can become more likely. It is also important to note that, for many practical problems, attempting to use all eight processors to solve Monte-Carlo histories concurently will increase the memory requirements beyond the size of the physical memory. Hence, for these problems some means of secondary storage will have to be used (disk paging on the Alliant). This will result in a significant increase in the apparent overhead factor and severely limit the concurrency speedup. Clearly alternative strategies will be necessary here which are discussed further in the next Chapter.

Table 4-8. Speedup and Efficiency for 3-D Space Truss (Sample size = 1,000, α = 0.0050)

N (CPU)	Theoretical Speedup	Actual Speedup	Overhead Factor	Efficiency (%)
1	1.0	1.0	0.0	100.0
4	3.94	3.89	0.0033	98.7
6	5.85	5.70	0.0046	97.4
8	7.73	7.43	0.0052	96.1

CHAPTER 5

SUMMARY, CONCLUSIONS, AND RECOMMENDATIONS

5.1 SUMMARY AND CONCLUSIONS

The objectives of this effort were to identify the special software and hardware research and development needs for implementing probabilistic structural mechanics problems (PSM) on parallel processing computers, and to demonstrate the feasibility and potential advantages of such an implementation. In order to meet these objectives three basic tasks were conducted and are reported on herein. First, currently available parallel processing hardware was reviewed in detail (Chapter 2) in order to be able to assess the adequacy of these architectures for PSM problems. Second, the sources of parallelism in PSM problems were identified (Chapter 3) to assess the required software strategies for implementation and to what extent parallelism in PSM problems can be exploited. Third, several example implementations were carried out (Chapter 4) in order to demonstrate the feasibility and potential advantages of the parallel implementation. This implementation was limited, for this Phase I effort, to two levels of parallelism in PSM, that is, the repeated performance function evaluations of direct Monte-Carlo simulation and vectorization in the structural mechanics computations; and one hardware architecture, that is, a shared memory vector/concurrent multiprocessor.

While the example implementations performed herein were highly successful, it is concluded that new hardware and software strategies will be required to achieve massive parallel implementations for many practical problems. This is a result of two factors - concurrency overhead and storage requirements

First, concurrency overhead must be kept extremely small to achieve reasonable efficiency for massively parallel applications. For example, even with the relatively small overhead factors of approximately 0.005, obtained in the examples, the maximum speedup is limited to 200 for an infinite number of processors. Hence, we will need to strive for parallel implementations with even smaller overhead than this.

Second, multiple levels of parallelism need to be exploited since storage requirements can easily exceed available memory if only one level of parallelism is implemented. Structural mechanics problems solved by the finite element method are memory intensive. It is not uncommon for a single problem to require memory in excess of several million 64-bit words (1 word = 8 Bytes). When multiple concurrent solutions are performed in a parallel PSM code, so that each processor is assigned an independent performance function evaluation, the memory required is multiplied by the number of processors. For example, say a 3-D analysis of a turbine blade requires 40 MBytes of storage (assuming banded matrix storage). Then on an 8 processor computer we would require on the order of 320 Mbytes of total storage, for

the parallel PSM implementation. This will overwhelm the physical memory available on all but a few supercomputers and this example has used only a small number of processors. Also, currently available distributed memory machines support no more than 8-16 Mbytes per processor (node).[1]

The key conclusions of this study can be summarized as follows:

1. A wide range of parallel architectures have been developed and are currently available, however, none seem to be ideally suited for parallel implementation of PSM.

2. Minimization of concurrency overhead is crucial to effective implementation of parallel PSM on a large scale.

3. There are several levels of parallelism in PSM problems that may need to be taken advantage of in order to fully exploit the potential of parallel processing computers.

4. Very high efficiency (greater than 96% for 8 processors) can be achieved for parallel Monte-Carlo PSM codes.

5. Specially adapted numerical techniques will be required for efficient parallel implementation of many practical problems in order to reduce memory requirements and processor idling.

6. Existing hardware technologies can be applied to develop a computer architecture that is ideally suited for parallel PSM.

7. Availability of parallel computers with properly adapted software show excellent potential for practical turn around time on large scale PSM problems.

5.2 RECOMMENDATIONS

Based on the work performed herein we can define optimal, generic hardware and software specifications for parallel processing of PSM problems.

1. Distributed memory (as opposed to shared memory) is preferable. This is because PSM problems involve a large number of independent calculations. Although there are communication advantages for a shared memory system, the overhead cost associated with shared memory is not justified for PSM problems. For a shared memory machine, access to the shared memory can become a bottleneck,

[1] This is expected to increase over the next 6-12 months by a factor of 4, when 4 megabit memory chips become widely available, although at significant expense.

particularly for the large number of processors desired for PSM problems.

2. Local memory at each distributed node should be greater than that on currently available distributed memory computers. As was demonstrated above, practical structural mechanics problems are memory intensive. Since it will not be practical or economical to provide sufficient memory at each node to solve most structures, alternative software strategies will need to be implemented (see below).

3. Each distributed processor should be capable of performing the numeric floating point operations required in structural mechanics problems with high speed and would ideally have vector pipeline capability.

4. A host controller processor is required to track the simulation history number being performed at each distributed processor. This will allow for dynamic history allocation and unbiased results. This processor need not be very powerful or have large memory, since each distributed processor can handle its own random number generation, performance function evaluation, and scoring. Communication between this processor and the distributed processors will be limited and infrequent, but the connection topology should be such that the host processor is closely linked with each individual processor.

5. The communication topology must be flexible enough to allow for direct communication among small clusters of the processors. A topology defined by clusters of low dimension hypercubes or low dimension pyramids may satisfy this specification. This will allow more efficient handling of large problems wherein multiple levels of parallelism will need to be implemented.

6. Controlling software must be developed to optimally allocate the multiple levels of parallelism among the processors to minimize processor idling and achieve maximum speedup. The task of assigning different processors to different tasks must be handled by this software. As a simple example, if 40 Mbytes of storage are required to solve a structure and only 8 Mbytes are available at each processor node, 5 processors at a minimum must be assigned to solve a single structure. Decomposition among these 5 processors must then be accomplished. For Monte-Carlo simulation on a machine with, say 100 processors, 20 simulation histories would then be processed concurrently. Or, if a partial derivative method is used, each cluster of 5 processors is assigned to one partial derivative. If the number of random variables is less than 20 the clusters could be grouped, introducing another level

of parallelism, with different groups of clusters working on different parts of the cumulative distribution function (see Chapter 3).

7. Special numerical techniques will be required that are adapted to this architecture. These numerical techniques must minimize storage requirements, be able to be implemented concurrently, and minimize the computational effort in the PSM computations. Under this Phase I effort the initial development of one possible technique was begun. This technique, the Stochastic Pre-Conditioned Conjugate Gradient (SPCG) method is described in Appendix A, and an example implementation is presented. Dramatic reductions in storage and computational effort are possible with this approach. In addition, the method can also be implemented concurrently. Hence, it shows the potential to satisfy the requirements of this specification.

These generic specifications form the basic recommendations for a system that can achieve practical computational time for large scale PSM computations. Based on our review of currently available hardware (see Chapter 2 and Table 2-2) current architectures do not meet these specifications. However, the basic technologies do exist. Specific, practical approaches toward meeting these specifications are outlined in our Phase II proposal, along with a specific research plan for developing the system.

CHAPTER 6

REFERENCES

Agarwal, Tarun K., Storaasli, Olaf O., and Nguyen, Duc T., "A Parallel-Vector Algorithm for Rapid Structural Analysis on High-Performance Computers," Presented at the *AIAA/ASME/ASCE/AHS 31st Structures, Structural Dynamics and Materials Conference*, Long Beach, California, April 2-4, 1990.

Amdahl, G., "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," *Proceedings of the Spring Joint Conference of AFIPS*, 1967.

Ang, Alfredo H-S., and Tang, Wilson H., *Probability Concepts in engineering Planning Design, Vol. II: Decision, Risk, and Reliability*, John Wiley & Sons, New York, 1984.

Annaratone, M., *et al.*, "The Warp Computer: Architecture, Implementation, and Performance," *IEEE Transactions on Computers*, C-36(12): 1523-1538, December 1987.

Bacon, Ben, "NCUBE 2 and Thinking Machines CM-2a Aimed at University Market," *Computers in Physics*, pages 8-9 (July-August, 1989).

Barkai, D., Moriarity, K. J. M., and Rebbi, C., "A highly optimized vectorized code for Monte Carlo simulations of SU(3) lattice gauge theories," *Computer Physics Communications*, 32 (1984), 1.

Barnes, G. H., *et al.*, "The Illiac IV Computer," *IEEE Trans. on Computers*, C-17(8):746- 757, August 1968.

Brown, F. B., "Vectorization of 3-D General-Geometry Monte Carlo", *Trans. Am. Nucl. Soc.* **53**, 283 (1986).

Brown, F. B., "Vectorized Monte Carlo methods for reactor lattice analysis," *Conference on Monte Carlo Methods and Future Computer Architectures*, Brookhave National Laboratory, April 2-5, 1983.

Brown, F.B., and Martin, W.R., "Monte Carlo Methods For Radiation Transport Analysis on Vector Computers", *Progress in Nuclear Energy* **14**, 269 (1985).

Bucher, I.Y., and Simmons, M.L., "Performance Assessment of Supercomputers," LA-UR-85-1505, Los Alamos National Laboratory (1985).

Chan, T. F., Glowinski, R., Periaux, J., Widlund, O., *Domain Decomposition, Methods for Partial Differential Equations*, SIAM, Philadelphia, PA, 1989.

Crowther, W., *et al.*, "Performance Measurements on a 128-node Butterfly Parallel Processor," In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 531-540, August 1985.

Davis, D.B., "Parallel Computers Diverge," *High Technology*, 16-22, February 1987.

Denning, P.J., "Parallel Computing and its Evolution," *Communications of the ACM*, 29(12):1163-1167, December 1986.

Dennis, Jack B., "Programming Generality, Parallelism and Computer Architecture," In *Information Processing* 68, pages 484-492, North-Holland Publishing Co., 1969.

Dennis, Jack B., "The Varieties of Data Flow Computers," In *Proceedings of the 1-st International Conference on Distributed Computing Systems*, pages 430-439, October 1979.

Dongarra, J. J., "Performance of Various Computers Using Standard Linear Equations Software in a Fortran Environment," Tech. Mem. 23, Argonne Nat. Laboratory (1986).

Dongarra, J.J., and Duff, I.S., "Advanced Architecture Computers," *ANL/MCS-TM-57*, Rev. 2, Argonne National Laboratory, September 1989.

Duncan, R., "A Survey of Parallel Computer Architectures," *IEEE Computer*, 5-16, February 1990.

Farhat, C. H., Felippa, C. A., Park, K. C., "Implementation Aspects of Concurrent Finite Element Computations," *Parallel Computations and Their Impact on Mechanics*, Proceedings Winter Annual Meeting ASME, December 1987.

Flynn, M. J., "Very High-speed Computing Systems," *Proc. IEEE*, 54(12):1901-1909, December 1966.

Foster, M. J., and Kung, H. T., "The Design of Special-purpose VLSI Chips," *Computer*, 13(1): 26-40, January 1980.

Fox, Geoffrey C., *et al.*, *Solving Problems on Concurrent Processors*, Volume I, "General Techniques and Regular Problems," Prentice Hall, Englewood Cliffs, New Jersey.

Fox, Geoffrey, *The Performance of the Caltech Hypercube in Scientific Calculations*, Technical Report CALT-68-1298, California Institute of Technology, Pasadena, Calif., April 1985.

Frederickson, P., Hiromoto, R., Jordan T., Smith, B., Warnock, T., "Pseudo-random trees in Monte Carlo," *Proceedings, Conference on Monte Carlo Methods and Future Computer Architectures*, Brookhaven National Laboratory April 2-5, 1983.

Gottlieb, A., *et al.*, "The NYU Ultracomputer -- Designing a MIMD, Shared Memory Parallel Machine," *IEEE Transactions on Computers*, C-32:175-189, February 1983.

Gustavson, J.L., Montry, G.R., and Benner, R.E., "Development of Parallel Methods for a 1024-node Hypercube," *SIAM J. Scientific and Statistical Computing* **9**, July, 1988.

Hayes, J. P., Mudge, T. N., Stout, Q. F., Colley, Steve and Palmer, John, "A Microprocessor-based Hypercube Supercomputer,"*IEEE MICRO*, 6-17, October 1986.

Hestenes, Magnus R., and Stiefel, Eduard, "Methods of Conjugate Gradients for Solving Linear Systems," *Journal of Research of the National Bureau of Standards*, Vol. 49, No. 6, December 1952.

Hockney, R. W. and Jesshope, C. R., *Parallel Computers*, Adam Hilger, Ltd., Bristol, England, 1981.

Hohenbichler, M. and Rackwitz, R., "Non-normal Dependent Vectors in Structural Safety," *J. of Eng. Mech.*, ASCE, 107(6), 1981, pp. 1227-1241.

Holland, J. H., "A Universal Computer Capable of Executing an Arbitrary Number of Subprograms Simultaneously," *Proc. East. Joint Comput. Conf.* **16**, 108-113, 1959.

Johnson, G.M., "Exploiting Parallelism in Computational Science," *Future Generation Computer Systems* **5**, 319-337, 1989.

Knuth, D. E., *Seminumerical Algorithms, The Art of Computer Programming*, Vol. 2, Addison-Wesley, Readung, MA, 1973.

Kuck, D. L., Davidson, E. S., Lawrie, D. H., and Sameh, A. H., "Parallel Supercomputing Today and the Cedar approach. *Science*, 231:967-974, February 1986.

Lawrie, D. H., "Access and Alignment of Data in an Array Processor," *IEEE Trans. on Computers*, C-24(12):1145-1155, December 1975.

Liu, P-L. and Der Kiureghian, A., "Multivariate Distribution Models with Prescribed Marginals and Covariances," *Probabilistic Engineering Mechanics*, Vol. 1, No. 2, 1986, pp. 105-112.

Madsen, H. O., Krenk, S., and Lind, N. C., Methods of Structural Safety, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1986.

Malaguti, F., "A Vectorized Monte-Carlo Code to Simulate Charged-Particle Motion Within Crystals," *Nuc. Inst. and Meth. in Phys. Res.*, Vol. B34, 1988.

Martin, W. R. and Brown, F. B., "Present Status of Vectorized Monte Carlo for Particle Transport Analysis," *International Journal of Supercomputer Applications* **1**, 32(1987).

Martin, W. R., Nowak, P. F., and Rathkopf, J. A., "Monte Carlo Photon Transport on a Vector Supercomputer" *IBM Journal of Research and Development* **30**, 193 (1986).

Martin, W. R., T. C. Wan, T. Abdel-Rahman, and T. N. Mudge, "Monte Carlo Photon Transport on Shared Memory and Distributed Memory Parallel Processors," *International Journal of Supercomputer Applications* **1**, No. 3, 57-74 (1987).

Martin, W.R. and Brown, F.R., "Status of Vectorized Monte Carlo for Particle Transport Analysis," *The International Journal of Supercomputer Applications*, Vol. 1, 1987.

Martin, W.R., Wan, T-C., Abdel-Rahman, T.S. and Mudge, T.N., "Monte Carlo Photon Transport on Shared Memory and Distributed Memory Parallel Processors," *The International Journal of Supercomputer Applications*, Vol. 1, 1987.

Mead, Carver, and Conway, Lynn, *Introduction to VLSI Systems*, Addison-Wesley, 1980.

Menabrae, General L. F., "Sketch of the Analytical Engine Invented by Charles Babbage," *Bibliothéque Universelle de Genéve*, October 1842.

Miura, K., "EGS4V: Vectorization of the Monte Carlo Cascade Shower Simulation Code EGS4," *Computer Physics Communications*, Vol. 45, 1987.

Moatti, A., Goldberg, J. and Memmi, G., "Parallel Monte Carlo Calculations with Many Microcomputers," *Computer Physics Communications*, Vol. 45, 1987.

Mori, M., Tsuda, Y., "Vectorized Monte Carlo Simulation of Large Ising Models near the Critical Point," *Physical Review* B, Vol. 37, 1988.

Mudge, T. N., Buzzard, G. D., and Abdel-Rahman, T. S., "A High Performance Operating System for the NCUBE," in M. T. Heath, editor, *Hypercube Multiprocessors 1987*, pages 90-99, 1987, Society for Industrial and Applied Mathematics, (Proceedings of the 1986 Conference on Hypercube Multiprocessors).

Nataf, A., "Determiniation des Distribution dont les Marges sont Donnees," *Comptes Rendus de l'Academie des Sciences*, Paris, 1962, **225**, pp. 42-43.

Newell, J. F., Rajagopal, K. R., and Ho, H., "Probabilistic Structural Analysis of Space Propulsion System Turbine Blades," presented at the *30th AIAA Structures, Structural Dynamics, and Mechanics Conference*, 1989.

Nour-Omid, B., "A Preconditioned Conjugate Gradient Method for Solution of Finite Element Equations," *Innovative Methods for Nonlinear Problems*, Ed. by Liu, Belytschko, Park, Pineridge Press, U. K., 1984.

Pfister, G. F., Brantley, W. C., George, D. A., Harvey, S. L., Kleinfelder, W. J., McAuliffe, K. P., Melton, E. A., Norton, V. A., and Weiss, J., "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture," in *Proceedings of the 1985 International Conference on Parallel Processing*, pages 764-771, August 1985.

"Physicists may be on the threshold of revolutionizing software," *New York Times*, page E2 of the *Flint Journal*, May 13, 1990.

Poole, Eugene L., "Comparing Direct and Iterative Equation Solvers in a Large Structural Analysis Software System," *Proceedings of the 4th Conference on Iterative Methods*, Copper Mt., CO, April 1990.

Potter, J. P., editor, *The Massively Parallel Processor*. MIT Press, Cambridge, Mass., 1985.

Przemieniecki, J. S., "Matrix Structural Analysis of Substructures," Journal of the American Institute of Aeronautics and Astronautics, Vol. 1, No. 1, 1963.

Rosenblatt, M., "Remarks on a Multivariate Transformation," *The Annals of Mathematical Statistics*, Vol. 23, 1952, pp. 470-472.

Seitz, C. L., "The Cosmic Cube," *Communications ACM*, 28(1):22-33, January 1985.

Siegel, H. J., *Interconnection Networks for Large-Scale Parallel Processing: Theory and Case Studies*. Lexington Books, Lexington, Massachusetts, 1985.

Siegel, H. J., Siegel, L. J., Kemmerer, Jr., F. C., Mueller, Jr., P. T., Smalley, H. E., and Smith, S. D., "PASM: a Partitionable SIMD/MIMD System for Image Processing and Pattern Recognition," *IEEE Transactions on Computers*, C-30:934-947, December 1981.

Slotnick, D. L., Borck, W. C., and McReynolds, R. qC., "The SOLOMON Computer," *AFIPS Conf. Proc.* **22**, 97-107, 1962.

Stone, Harold S., *High-Performance Computer Architecture*, Addison-Wesley, Reading, Mass., 1987.

Sues, R.H., and Twisdale, L.A., "Probability-Based Design Factors," Chapter 8, DNA Manual for the Design of Hardened Underground Facilities, ARA 5955, Draft Final, April 1988.

Tanimoto, S. L., "A pyramidal approach to parallel processing," In *Proceedings of the 10-th Annual International Symposium on Computer Architecture*, pages 372-378, June 1983.

Traynor C.A., Anderson, J.B., "Parallel Monte-Carlo Calculations to Determine Energy Differences Among Similar Moecular Structures," *Chemical Physics Letters*, Vol. 147, 1988.

Vohwinkel, C., "A Fast Method to Gather Neighbors in Vectorized Monte-Carlo Simulations," *Computer Physics Communications*, Vol. 51, 1988.

Wansleben, S., "Ultrafast Vectorized Multispin Coding Algorithm for the Monte Carlo Simulation of the 3D Ising Model," *Computer Physics Communications*, Vol. 43, 1987.

Wolfe, A., "Is Parallel Software Catching Up with the Hardware At Last?", *Supercomputing Review*, 29-33, March 1989.

Worlton, J., "Towards a Science of Parallel Computation," *Computational Mechanics - Advances and Trends*, AMD **75**, 23-35, ASME, New York, 1987.

Wu, Y.-T., Burnside, O. H., and Cruse, T. A., "Probabilistic Methods for Structural Response Analysis," *Proceedings* of Applied Mechanics and Engineering Sciences Conference, ASME/SES Summer Meeting, UC Berkeley, June, 1988.

Yokozawa, M., Oka, Y., Kondo, S. and Togo, Y., "Development of Vectorized Monte Carlo Calculation and Application of Stratified Sampling," *Journal of Nuclear Science and Technology*, Vol. 24, 1987.

APPENDIX A

DEVELOPMENT OF THE STOCHASTIC PRE-CONDITIONED CONJUGATE GRADIENT METHOD

The memory required to perform performance function evaluations concurrently on a massively parallel processor can easily exceed the available resources. Hence, for many practical problems it will be necessary to take advantage of the parallelism in both the probabilistic computations and the structural mechanics computations as identified in Chapter 3.

One approach to take advantage of the parallelism in the structural mechanics computations discussed in Chapter 3 is the use of operator splitting techniques. We investigated herein one type of operator splitting technique in the form of an iterative equation solver, the Pre-Conditioned Conjugate Gradient (PCG) Method. There were several motivating factors for this investigation:

1. The method is easily implemented in parallel since all computations are essentially matrix and vector multiplications.

2. It requires minimal storage since "fills" do not occur during solution and sparse storage methods are effective for large problems.

3. The method can take advantage of "knowledge" gained from the "mean-value" solution obtained at the beginning of the simulation.

4. Computational effort can be reduced by reducing required precision.

These four factors make the PCG method a promising candidate as the equation solver in a parallel Monte-Carlo simulation code. Not only is it straightforward to take advantage of concurrency and vectorization on parallel computers, but storage requirements are also reduced from direct solvers. In fact, Poole [1990] demonstrated reductions in storage requirements ranging from factors of 4 to greater than 10 for general practical problems. As was presented in Chapter 5, storage requirements will be a key factor in achieving massively parallel implementations of PSM. In addition, as will be shown below, the computational effort for each simulation history can be minimized through the use of the reduced preconditioning matrix that is available in Monte-Carlo simulation at essentially no extra cost, and by taking advantage of the reduced precision that may be possible in Monte-Carlo simulation.

The PCG method, which is obtained by combining a suitable preconditioning matrix with the basic Conjugate Gradient Method [Hestenes and Stiefel, 1952], has been shown to be a very powerful approach for solving large systems of equations

[Poole, E., 1989; Nour-Omid, 1984]. In the PCG method the standard linear structural analysis problem:

$$\mathbf{K}\mathbf{x} = \mathbf{f} \tag{A-1}$$

where **K** is the stiffness matrix, **f** is the loading vector, and **x** the structure unknown displacements, is modified by pre-multiplying both sides of the equation by a preconditioning matrix, **M**, to obtain:

$$\mathbf{M}^{-1}\mathbf{K}\mathbf{x} = \mathbf{M}^{-1}\mathbf{f} \tag{A-2}$$

The preconditioning matrix, **M**, is a symmetric positive definite matrix and is chosen to approximate **K**. The PCG algorithm to solve this system of equations then proceeds by first selecting an initial solution guess, $\mathbf{x}_o$ and computing

$$\mathbf{r}_o = \mathbf{f} - \mathbf{K}\mathbf{x}_o$$
$$\mathbf{h}_o = \mathbf{p}_o = \mathbf{M}^{-1}\mathbf{r}_o$$

The following steps are then repeated until convergence is achieved.

1. $$\alpha_i = \frac{\mathbf{r}_i \bullet \mathbf{h}_i}{\mathbf{K}\mathbf{p}_i \bullet \mathbf{p}_i}$$

2. $$\mathbf{x}_{i+1} = \mathbf{x}_i + \alpha_i \mathbf{p}_i$$
$$\mathbf{r}_{i+1} = \mathbf{r}_i - \alpha_i \mathbf{K}\mathbf{p}_i$$

3. $$\mathbf{h}_{i+1} = \mathbf{M}^{-1}\mathbf{r}_{i+1}$$

4. $$\beta_i = \frac{\mathbf{r}_{i+1} \bullet \mathbf{h}_{i+1}}{\mathbf{r}_i \bullet \mathbf{h}_i}$$

5. $$\mathbf{p}_{i+1} = \mathbf{h}_{i+1} + \beta_i \mathbf{p}_i$$

The convergence criteria can be taken as

$$\frac{\mathbf{r}_{i+1} \bullet \mathbf{h}_{i+1}}{\mathbf{r}_o \bullet \mathbf{h}_o} < \eta$$

where η is the convergence tolerance.

The preconditioning serves to improve the rate of convergence but at the cost of the additional computations required in step 3. A considerable amount of

research has been conducted in recent years to develop approaches for obtaining optimal preconditioning. That is, preconditioning wherein the minimum computational effort is required by achieving maximal rate of convergence with minimal extra computational effort in Step 3. The closer the preconditioning matrix approximates the stiffness matrix, the faster the rate of convergence, but the greater the additional computational expense (this derives from the obvious fact that for $\mathbf{M} = \mathbf{K}$, convergence is in one step).

For PSM problems, a candidate preconditioning matrix is clearly the mean stiffness matrix. The mean stiffness matrix will be a good approximation to the stiffness matrix of any particular simulation history and it need only be solved once (*i.e.*, inverted or cholesky factored) prior to the commencement of the simulation loop. In addition, the solution obtained using the mean stiffness matrix and mean load vector can be used as the initial guess.

We can, therefore, define the Stochastic Preconditioned Conjugate Gradient method (SPCG) as follows:

1. Select the preconditioning matrix to be the mean stiffness matrix, that is, $\mathbf{M} = \overline{\mathbf{K}}$, and;

2. Select the initial guess to be:

$$\mathbf{x}_o = \overline{\mathbf{K}}^{-1} \overline{\mathbf{f}}$$

In order to investigate this approach, the SPCG solver was implemented in MCPAP and applied to the 3-D Space Truss problem presented in the Chapter 4. Table A-1 shows the results of this analysis.

Table A-1. Stochastic Preconditioned Conjugate Gradient Method (SPCG) Applied to 3-D Space Truss Problem*

Pre-Conditioning Matrix	Initial Solution Guess	Tolerance η	Number of Iterations		Solution Time (sec)	Comments
			Mean	Standard Deviation		
$\mathbf{K}_D$	$\overline{\mathbf{K}}^{-1}\overline{\mathbf{f}}$	0.01	42	6.0	638	
$\overline{\mathbf{K}}$	$\overline{\mathbf{K}}^{-1}\overline{\mathbf{f}}$	0.01	4.2	0.48	101	Tenfold reduction in no. iter, 60% increase in time per iteration.
$\overline{\mathbf{K}}$	$\overline{\mathbf{K}}^{-1}\overline{\mathbf{f}}$	0.10	3.2	0.41	86	Accurate to 3 decimals for CDF values [0.01, 0.999].

* See Figure 4-9 , Table 4-6, Table 4-7 (Truss has 99 Members and 72 Degrees of Freedom, 1000 Monte-Carlo simulations were used).

As shown in the table three example analyses were conducted. For each example, 1000 Monte-Carlo simulation histories were performed, and the number of iterations required to achieve a converged solution for each history was retained. The table shows the mean and standard deviation of the number of iterations for each example, along with the total solution time.

Dramatic improvement in the mean number of iterations for convergence can be seen with the SPCG method. In the first example a simple and common preconditioning strategy, often referred to as Jacobi preconditioning was used. Here the preconditioning matrix is selected to be the main diagonal of the stiffness matrix (that is, of the stiffness matrix formed within the particular simulation trial). As shown in the table this method required, on average, 42 iterations to achieve a converged solution. Next the SPCG method was used. The preconditioning matrix was selected to be the mean stiffness matrix and the initial guess obtained as described above and as shown in the table. The mean number of iterations required to achieve a converged solution reduced by an order of magnitude to 4.2. The solution time is, however, reduced by a smaller factor of 6.3 due to the additional computational effort required by the SPCG method. Note that the reduction in number of iterations can be attributed to the preconditioning since the initial guess is identical in both cases.

The effect of reducing the convergence criteria was next investigated. The purpose here was to see if the number of iterations could be further reduced without significantly affecting the accuracy of the Monte-Carlo results. The tolerance was reduced by an order of magnitude from 0.01 to 0.1 and as shown in the table, the mean number of iterations reduced by another 25%. This reduction in tolerance had no appreciable affect on the Monte-Carlo results. For the cumulative distribution function values calculated for the example, which range from 0.01 to 0.999 (see Section 4.5), there was no change in results for three decimal places. The significance of these results is that it may be possible to reduce the tolerance required in Monte-Carlo simulation from that which is normally required in a deterministic evaluation.

Further work will be needed to determine what convergence criteria should be used in Monte-Carlo simulation. In fact, for Monte-Carlo a "smart" convergence criteria should be developed that reflects the basic problem uncertainties. Clearly, a high degree of precision should not be required when the problem uncertainties are very large. This can be formalized as follows:

Let Y = Random Response Variable

Then $Y = \hat{Y} + \varepsilon$

Where ε = Numerical Error

And $\overline{Y} = \overline{\hat{Y}} + \bar{\varepsilon}$

$$\sigma^2_Y = \sigma^2_{\hat{Y}} + \sigma^2_\varepsilon$$

From the above it is seen that the convergence criteria should be adjusted so that $\bar{\varepsilon}/\overline{Y}$ and $\sigma_{\bar{\varepsilon}}/\sigma_{\overline{Y}}$ are acceptable. That is, since the Monte-Carlo simulation results are an average over a large number of trials, the numerical solution procedure should be adjusted so that the mean and standard deviation of the numerical error do not significantly affect the Monte-Carlo results, that is the mean and standard deviation of the random response variable Y (where Y may actually represent the response probability).

NASA
National Aeronautics and Space Administration

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA CR-187162		

4. Title and Subtitle	5. Report Date
Probabilistic Structural Mechanics Research for Parallel Processing Computers	August 1991
	6. Performing Organization Code

7. Author(s)	8. Performing Organization Report No.
Robert H. Sues, Heh-Chyun Chen, Lawrence A. Twisdale, and William R. Martin	ARA-5589
	10. Work Unit No. 505-13-00

9. Performing Organization Name and Address	11. Contract or Grant No.
Applied Research Associates, Inc. 6404 Falls of Nuese Road, Suite 200 Raleigh, North Carolina 27615	NAS3-25824
	13. Type of Report and Period Covered Contractor Report Final

12. Sponsoring Agency Name and Address	14. Sponsoring Agency Code
National Aeronautics and Space Administration Lewis Research Center Cleveland, Ohio 44135-3191	

15. Supplementary Notes

Project Manager, P.L.N. Murthy, Structures Division, NASA Lewis Research Center, (216) 433-3332.

16. Abstract

Aerospace structures and spacecraft are a complex assemblage of structural components that are subjected to a variety of complex, cyclic, and transient loading conditions. Significant modeling uncertainties are present in these structures, in addition to the inherent randomness of material properties and loads. To properly account for these uncertainties in evaluating and assessing the reliability of these components and structures, probabilistic structural mechanics (PSM) procedures must be used. Significant advances have taken place in PSM over the last two decades. Much of this research has focused on basic theory development and the development of approximate analytic solution methods in random vibrations and structural reliability. Practical application of PSM methods has been hampered by their computationally intense nature. Solution of PSM problems requires repeated analyses of structures that are often large, and exhibit nonlinear and/or dynamic response behavior. A single deterministic solution of such structures can strain available computational resources. These methods, however, are all inherently parallel and ideally suited to implementation on parallel processing computers. While there has been research into parallel implementation of Monte-Carlo methods in physics and nuclear engineering (see Table 1-1), no research has been conducted in PSM. A need exists to systematically study implementation of these methods on parallel architectures and identify the optimal hardware and software specifications. New hardware architectures and innovative control software and solution methodologies are needed to make solution of large scale PSM problems practical. The next decade of research in computational PSM and the promise of parallel computing may open up a whole new class of nonlinear finite element and dynamics problems to probabilistic structural analysis.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
Monte Carlo; Truss-type structures; Computational advantages; Pilot program	Unclassified - Unlimited Subject Category 39

19. Security Classif. (of the report)	20. Security Classif. (of this page)	21. No. of pages	22. Price*
Unclassified	Unclassified	80	A05

NASA FORM 1626 OCT 86

*For sale by the National Technical Information Service, Springfield, Virginia 22161