CIVIL ENGINEERING STUDIES
Civil Engineering Systems Laboratory
Research [illegible]

UILU-ENG-90-2001

ISSN - 0735-2867

ARCHITECTURE INDEPENDENT ENVIRONMENT FOR DEVELOPING ENGINEERING SOFTWARE ON MIMD COMPUTERS

by

P-152

KARIM A. VALIMOHAMED
L. A. LOPEZ

Technical Report of Research
Supported by

National Aeronautics and Space
Administration – Langley Research Center

ARCHITECTURE-INDEPENDENT ENVIRONMENT
FOR DEVELOPING
ENGINEERING SOFTWARE ON MIMD COMPUTERS

BY

KARIM A. VALIMOHAMED
L. A. LOPEZ

FINAL REPORT

For Research Sponsored
by NASA Langley (CSM)
Grant #NAG-1-5-25622

ABSTRACT

Engineers are constantly faced with solving problems of increasing complexity and detail. They frequently rely upon numerical methods to solve these problems, and their insatiable appetite for improved performance from computing hardware has reached a point where the computational requirements exceed reasonable expectations of the performance of Von-Neumann (serial) computers.

Multiple Instruction stream Multiple Data stream (MIMD) computers have been developed to overcome the performance limitations of serial computers. The hardware architectures of MIMD computers vary considerably and are much more sophisticated than serial computers. Developing large scale software for a variety of MIMD computers is difficult and expensive. There is a need to provide tools that facilitate programming these machines.

The first part of this report examines the issues that must be considered to develop those tools. The two main areas of concern were architecture independence and data management. Architecture independent software facilitates software portability and improves the longevity and utility of the software product. It provides some form of insurance for the investment of time and effort that goes into developing the software. The management of data is a crucial aspect of solving large engineering problems. It must be considered in light of the new hardware organizations that are available.

The second part of the report describes the functional design and implementation of a software environment that facilitates developing architecture independent software for large engineering applications. The topics of discussion include: a description of the model that supports the development of architecture independent software; identifying and exploiting concurrency within the application program; data coherence; engineering data base and memory management.

The scope of the research is restricted to the development of the conceptual design of the environment and the implementation of a prototype system on a distributed memory system. The results obtained from developing two application problems related to finite element analysis are discussed. It is shown that the application program is indeed architecture-independent however the implementation of a shared memory prototype is left for future work.

ACKNOWLEDGEMENTS

This report is based on the dissertation submitted to the Graduate College of the University of Illinois at Urbana-Champaign by Karim A. Valimohamed in partial fulfillment of the requirements for the Doctor of Philosophy degree in Civil Engineering. The research was conducted under the supervision of Professor Leonard A. Lopez.

Financial assistance for this study was provided primarily by the Computational Structural Mechanics Group at NASA - Langley. Initial support was also provided by the Apollo Computer Company. This support is gratefully acknowledged.

The authors wish to thank Dr. Norman Knight and Mr. William Greene of NASA for suggesting the problem, and for their help and encouragement throughout the investigation. A special thanks goes to Ms. Linda Schub, Dr. Scott Schiff, and Ms. Nan Fancher for their contributions to various parts of the investigation.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 OVERVIEW OF THE PROBLEM

Engineers and scientists rely upon numerical methods to solve complex analytical problems. The analysis of such problems has been facilitated by technological advances in electronic computation. The availability of general purpose computers has influenced the development and application of numerical methods in both engineering and in the sciences. This report addresses the issues related developing large scale software for engineering applications on computing hardware known as Multiple Instruction stream Multiple Data stream (MIMD) computers. The issues examined herein include the ability to develop architecture-independent software applications and the data management aspects of engineering problems.

The finite element method (FEM) is used as a focus problem throughout the report. The method grew out of the need to solve structural analysis problems in the aerospace industry during the late 1950's. Since then, FEM techniques have evolved rapidly, and are now established as a basic method for the analysis of complex engineering problems [15]. The method is widely used in the areas of soil mechanics, heat conduction, aerodynamics, and fluid flow. In structural engineering, it has been used for linear, nonlinear, static, and dynamic analysis of complex problems. In the remainder of this paper, reference to the FEM will be made in the context of its application in structural analysis

Engineers are constantly faced with solving problems of increasing complexity and detail. Their insatiable appetite for improved computational performance from the hardware has reached a point where computational requirements often exceed reasonable expectations on the capacity of Von-Neumann type computers. The latter are generally referred to as "serial computers" herein. Although the revolution in microelectronics has resulted in tremendous advances in computing power on a single chip, the speed of light ultimately constrains the performance that can be obtained from a single chip. Therefore novel computer architectures have been developed in an attempt to satisfy the performance requirements of compute-intensive problems. These architectures incorporate multiple processors that cooperate with each other to achieve the solution to a single problem. These multiprocessor computers are also referred to as parallel computers.

History shows that the development of the finite element method has been closely linked to advances in computer technology; each new generation of hardware and software has permitted users to solve problems of increasing complexity. Recent developments in hardware technology provide parallel processing capabilities that are suited to exploit the natural parallelism inherent in the finite element method [52].

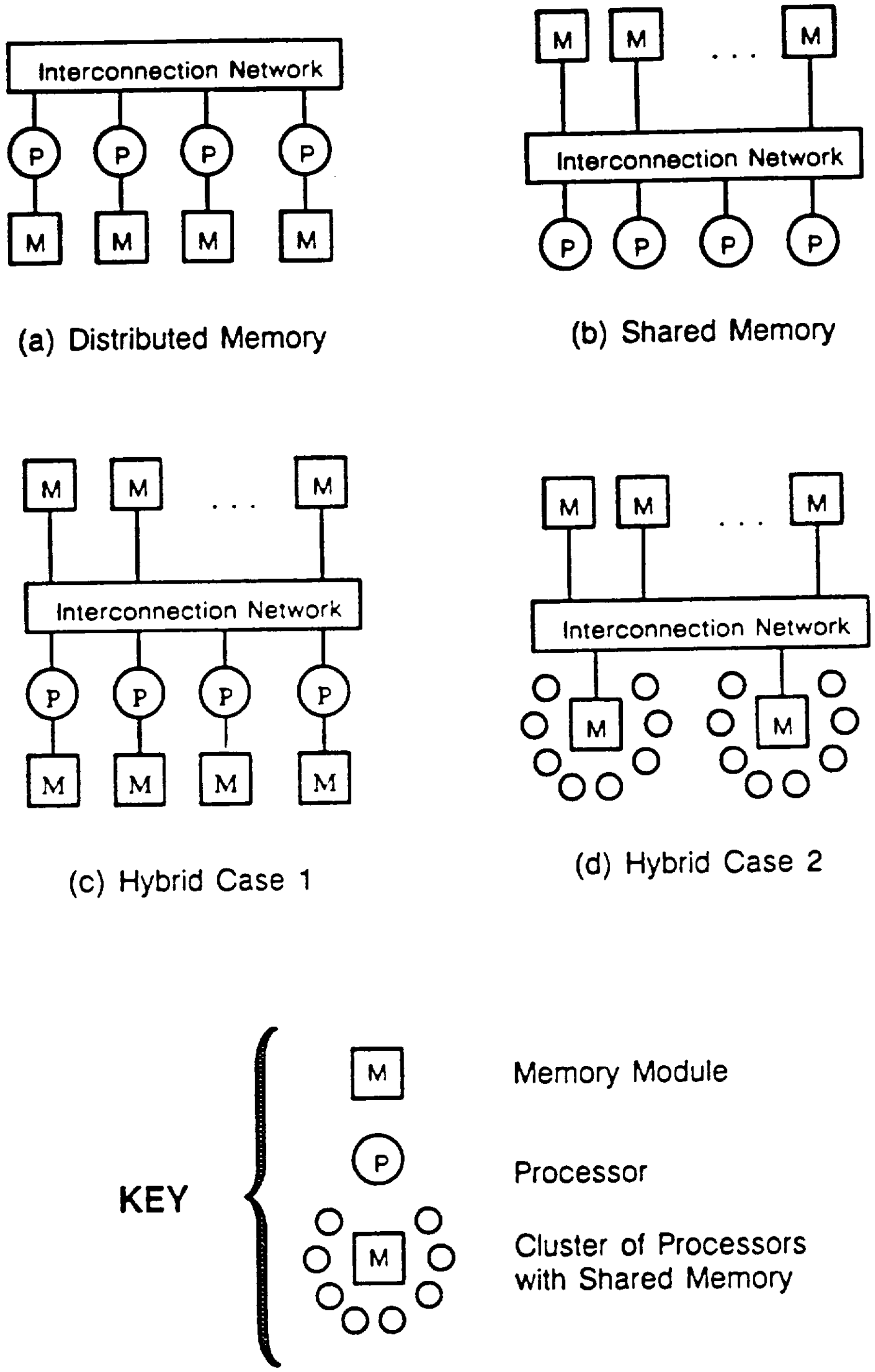
The idea of exploiting parallelism is not new, and various forms of parallelism exist to differing degrees in all computers at the arithmetic and and storage levels. Historically, parallelism at the processor level was an integral part of the early ILLIAC IV design but was obviated by progress is microelectronics [15]. Early parallel machines like ILLIAC IV were not very popular with engineers because they were too expensive, too cumbersome to program, and were not available for general use. Today, parallel computers are commercially available and, in some instances, may be a worthwhile investment.

1.2 MIMD ARCHITECTURES

The number and type of processors, memory modules, and their interconnecting network are among the many distinguishing features that are used to classify the architectures of parallel computers. Various other taxonomies[49] [39] [38] [30] [29] [40] and type architectures [76] have been developed in order to classify machines that have similar salient features.

In this paper, only those architectures referred to as Multiple Instruction Multiple Data stream computers are considered. This group of machines is based on a a taxonomy developed by Flynn. It classifies machines by the number of instruction and data streams. Despite its generality, Flynn's taxonomy is commonly used in the scientific and engineering communities.

The term MIMD provides a rather obscure definition for the classification of computers. Therefore, for the purposes of identifying the machine architectures that will be referred to in this report, a simple scheme proposed by Karp [45] is used. It consists of three types of machines, namely shared memory, message-passing, and hybrid systems. Their features are described below and their schematics are illustrated in figure 1.2-1.



(a) Distributed Memory

(b) Shared Memory

(c) Hybrid Case 1

(d) Hybrid Case 2

Figure 1.2-1. MIMD Architectures

Distributed–memory or loosely coupled systems, are networks of computing elements each having its own memory. No memory is globally accessible. Communication and data transfer is accomplished by "message passing". Examples of homogeneous message passing systems include the Hypercube [71] and Intel iPSC. General networks containing nonhomogeneous computing elements also fall into this category. The processing elements are combined by a multicomputer interconnection network that can be classified as asynchronous, decentralized, or packet–switched (fig. 1.2–1.a) [13].

Shared–memory or tightly coupled systems, have a single global memory that is shared by all the processing elements. Each processor may have its own local memory, such as vector registers on the Cray X–MP [50] or the cache on the IBM 3090 [83]. The use of such local memory is transparent to the user. Communication is accomplished via shared variables. An example of such a computer is the Alliant FX/8 [4] . The processors and memory modules are combined using a multiprocessor interconnection network that can be classified as being shared buses, crossbars, or multi–stage (fig. 1.2–1.b) [13].

Hybrid systems exist between the two extremes. They are systems which have properties of both shared memory and message–passing systems (figs. 1.2–1.c, 1.2–1.d). They are programmed as shared–memory machines but can have data access delays like message passing systems. These delays are much smaller than on message passing systems, but they can be significant [45]. Memory hierarchies can be found in some architectures like the ETA 10 [27] and Cedar [48], while other machines such as the IBM–RP3 [67] and Flex–32 [81] can be configured to suit the needs of the application.

The majority of commercially available MIMD computers are either shared or distributed memory type architectures. Hybrid machines (Cedar and RP3) are still in the experimental stages of development. MIMD machines provide a significant improvement in peak performance over serial machines, and therefore attract many large scale engineering applications that are inherently amenable to parallel processing.

MIMD computers are best utilized when they exploit both the concurrency and the vectorization that exists within the application code. Concurrency is the ability to execute multiple stream of instruction on multiple streams of data, whereas, vectorization is the ability to work on multiple stream of data for one stream of instructions. A sophisticated processor on an MIMD machine (e.g. Cray–XMP, Alliant–FX80) is usually equipped with a vector facility, thereby providing the capability to exploit both vectorization and concurrency within the application. Typically, such potential exists within do-loop type programming constructs of procedural languages. The finite element method has much potential for improved performance on MIMD computers because of the inherent concurrency and vectorization in the procedure. How can we exploit this technology?

1.3 CONCURRENCY IN THE FINITE ELEMENT METHOD

The finite element method algorithm for a linear static problem using the direct stiffness method consists of three major stages. In the preprocessing stage, the problem is defined by identifying the elements, the boundary conditions or constraints, and the applied loadings. In the processing stage, the stiffness matrices and internal load vectors for all of the elements are generated and are assembled into a global structure stiffness matrix and the global load vector. Given the load vector and stiffness matrix, the displacements of the structure are obtained by solving a linear system of equations (see equation 1.3-1). Finally, in the postprocessing stage, element quantities such as stresses and strains for some or all of the elements are computed.

$$[K]\{u\} = \{P\} \qquad \text{(equation 1.3-1)}$$

where: [K] is the global stiffness matrix,
{u} is the displacement vector, and
{P} is the load vector.

There is potential for concurrency and vector processing in all three stages of the method. The time spent in both the pre and post processing stages is dominated by the user. In the preprocessing stage, engineers develop the FEM model by using advanced graphics software on workstations. In the near future, graphics systems will be augmented with knowledge based systems that guide the engineer in developing the model. In the post processing stage, the engineer interacts with the system to interpret the results. This requires the use of advanced graphics/visualization software. Although there is a significant amount of computation in the pre and postprocessing stage, the processor time used in these areas is overshadowed by the time spent in the processing stage.

The processing stage is the most computation intensive part of the finite element problem.There is very little interaction with the user (with the exception of nonlinear solutions), and there is a tremendous potential for the use of concurrent and vector processing. Generating the individual stiffness matrices and load vectors for the elements can be performed concurrently since the computation of one element quantity is independent of all other elements. A large portion of the assembly of the global stiffness matrix can also be done in parallel. In systems employing the direct stiffness method, the solution to the system of equations can begin once the global stiffness matrix and load vector have been assembled. Direct methods of solving such equations exist in many of the existing large scale systems. The assembly and solution procedures are described in more detail in the following subsections.

1.3.1 The Assembly Process

An examination of the details of the global stiffness assembly process can help explain some of the aspects of parallel computing that should be considered.

A structure can be composed of a hierarchy of substructures or simply an array of a distinct element types. The elements can vary from a special purpose element, to a complex 3D shell element, to a simple truss component. There are several algorithms that can be used to compute the global stiffness matrix. Carey [14] has identified three categories, namely: (1) subdomain splitting; (2) substructure techniques; and (3) splitting.

In the first method, the domain is subdivided into overlapping regions. The original problem is decomposed into one that involves the solution of boundary value problems on the subdomain. In the second approach, the structure is divided into substructures. Each substructure is treated as a separate domain. The third approach, splitting, can be used to reduce the number of dimensions in the split problem by replacing a two-dimensional problem by a series of one-dimensional problems.

Substructure techniques are most commonly used for the FEM because they map directly to the procedures for describing problems and for solving problems. The process of describing a complex structure by the use of components (which in turn can be defined using sub-components) naturally lends itself to the substructure approach. Static condensation of substructures is used to reduce the size of stiffness matrix and to produce "super elements" [14]. In parallel processing, the calculation of the stiffness matrix for a substructure may be an ideal task unit. Furthermore, a substructure may be defined such that its interaction with other substructures is minimal, hence it is well suited for concurrent processing.

1.3.2 Solution of the System of Equations

Traditionally, direct methods such as the Gaussian Elimination and Cholesky decomposition have been used to solve the system of equations associated with the FEM. Iterative methods are also used but to a lesser degree. The latter are inherently concurrent, and were used in the past when there were severe limitations on the amount of RAM available. However, since design problems in structural analysis frequently involve many right-hand side vectors (e.g. various load combinations), factorization of the system matrix followed by repeated sweeps is both preferable and possible with today's large memory configurations. As a result, direct methods have become the prevalent approach and are used widely with such sparse solution schemes as banded solvers, envelope (or profile) solvers, frontal solvers, and hypermatrix solvers.

For problems with a single load vector, iterative techniques may be a viable alternative, particularly when very large problems are considered, and when memory limitations become a significant constraint. Iterative methods are most effective when the matrix is symmetric, positive-definite, and diagonally dominant. Generally this is not the case for nonlinear problems.

In many cases the success of iterative methods depends on special preconditioning strategies. In such cases, the method may fail or not be practical, and the current trend back toward iterative methods because of memory limitations on MIMD computers may be a serious mistake. It may be better to examine other methods of using the memory that is available.

In the analysis of time-dependent and nonlinear problems using implicit integration, sparse linear system solution techniques are still required, and the above considerations apply. Such solution techniques are well suited to machines with large fast memories.

1.4 APPLICATIONS OF FEM ON MIMD MACHINES

The advent of parallel processing has again escalated the research efforts on computer methods related to finite element techniques. The work associated with implementing the finite element method on MIMD machines has progressed on various topics. They include: (1) developing new algorithms for various components of the problem such as equation solving; (2) the development of special purpose software for a particular type of MIMD machine; (3) development of special purpose hardware; and (4), the transfer of existing finite element software systems to MIMD machines.

1.4.1 Hardware Approaches

- Finite Element Machine [1] developed at NASA-Langley: consists of a lattice architecture with a global bus connecting all the processing elements.
- FEM-2 [69] developed at NASA-Langley: incorporated a different approach to developing a finite element machine. The main feature was the use of several layers of virtual machine concepts developed using a top-down rather than a bottom-up approach.
- Parfem [61] developed at the University of Calgary: consists of several special purpose hardware modules used to generate element data, assemble the stiffness matrix, and solve the system of equations.
- Macropipelining approach proposed by Teng and Hwang [80]: intended to improve overall system performance by matching the processor bandwidth of all the component subsystems. The subsystems include: a host computer; preprocessor, linear system solver, and the postprocessor.

1.4.2 Software Approaches

Since the processing stage is the most compute intensive portion of the finite element procedure, most of the research related to exploiting the concurrency in the finite element method has been focused on solving the system of equations. They include:

- strategies for exploiting concurrency on both the assembly and solution processes. In particular, conjugate gradient multitasking [58], subdomain splitting [9], multicolor iterative methods [10], adaptive grids [1], gaussian elimination [2], and cholesky factorization [12] have been examined. An overview of other developments in this area can be found in [9].

- an applications library known as the uniform system. It has been developed on the Butterfly [3],[2],[16], and has been used for the development of CoFEM, a special purpose finite element program developed at BBN laboratories.
- PISCES [70]. This system was developed at the University of Virginia in cooperation with NASA-Langley, and is an environment to support parallel computing for scientific applications. An implementation exists on both the Flex-32 machine and on the Hypercube.

1.4.3 Observations on Existing FEM Applications on MIMD Computers

None of the hardware solutions that were just described has been completely developed and implemented. Furthermore, the author has not heard of any new proposals for developing special purpose finite element machines.

A large proportion of the software research has been focused on addressing very limited aspects of the finite element problem - developing and implementing better algorithms. However, these algorithms are suited to solving a restricted sets of problems, or focus on a relatively small aspect of the problem.

With the exception of PISCES, the projects identified above have not addressed the data management issues related to the solution of very large problems and, although, the concept of "windows" was proposed in the PISCES concept, the implementation of such a scheme has not been accomplished.

To develop general purpose solutions, all aspects of the problem must be examined. Very little work has been done on the development of complete general purpose systems. Examples of such systems for serial computers are ASKA, ADINA, ABAQUS, ANSYS, FINITE, and NASTRAN. These systems incorporate the concept of element libraries, various material models, and linear, nonlinear, and dynamic analysis capabilities; they operate on a wide variety of computers that embrace the FORTRAN model of computation.

Several commercial FEM programs have been ported to MIMD machines, but they have achieved limited success. Although larger problems are being solved faster, for the most part, the improvement is due to the faster clock speed and vector processing. When two or more processors are used, the improvement in performance seems to peak at a small number of processors [79]. This is because the conversion of these codes is often a simple compile combined with a few changes to obtain the best performance possible from the matrix operator packages. In essence, the codes have been vectorized by making use of the automatic vectorization algorithms built into the compilers. They have not been re-written to any extent so that the explicit concurrency inherent in the finite element method is exploited. It is not easy to exploit the latter in the context of just the FORTRAN model.

1.4.4 Tools to Facilitate Software Development for MIMD computers

Programmers have found writing software for these machines to be difficult and expensive [74]. Unlike serial computers, there are many variables to consider when developing software for MIMD machines.

Several software options have been proposed to facilitate programming. They include: extending existing languages to express concurrency and synchronization; developing compilers that will identify the concurrency in programs; adding a "language layer" on top of existing languages to describe the desired concurrency and necessary synchronization, while allowing the basic applications program to remain "relatively" unaltered; defining new languages and compiler systems [60]. In addition, efforts are being directed towards developing systems capable of restructuring existing FORTRAN programs [17]. Currently, multiprocessor computers offer a rather restricted set of software tools with which to develop parallel programs [45], and for many applications, the full potential of parallel computing has not been realized yet [74].

The approaches described above are machine dependent - they are not totally portable. Generally, such tools are available only for tightly coupled multiprocessor systems in which compilers can generate the synchronization and scheduling instructions based on a data dependency analysis. Such features are not available for loosely coupled systems. This is a crucial issue when considering the longevity of software and the investment associated with its' development. A case in point is the failure of ETA systems. Developing software for one type of machine can be very expensive! There are some developments in environments and programming tools [70][24][64] that assist programmers in developing portable code. However, these efforts have been restricted primarily to the control aspects of the program; i.e., describing to the computer when and how to do things in parallel.

The objective of developing these tools was to obtain improved performance by solving existing problems as fast as possible. These efforts have encountered various degrees of success. In some cases the approaches were focused on a rather small aspect of the total problem. In others, they were prohibitively expensive for general use because they incorporated special purpose hardware or software which was not portable. These approaches have not addressed the crucial data problems encountered when trying to solve very large problems. A key element in developing successful engineering systems is successful management of the large data spaces. Very little work has been done in this crucial area on MIMD machines.

In order to make progress in the development of engineering software on MIMD computers, it is necessary to restructure both the data and algorithms at a high level, as well as to vectorize the code at the lowest level [52]. This would simpiy be history repeating itself in the case of the finite element method. The development of the finite element method and its use for the analysis of large structural engineering problems has been directly linked with advances in computer technology. As computers changed, the algorithms changed.

The increased storage capacity of existing hardware has permitted larger problems to be analyzed - however this new technology is capable of doing much more. Thus, the philosophy employed herein is to find methods to solve the largest problem possible within a given time frame rather than trying to solve existing problems faster. Typical expected problem sizes will require data spaces of between 40 and 100 Gigabytes. This may seem large by today's standards, but not unreasonable; one Gigabyte problems are being solved on today's mini-supercomputers.

The solution of large problems must incorporate methods to manage data efficiently. Data management is the dominant system issue with serial computers. On MIMD computers, process control, load balancing, granularity of tasks, data access, and data coherence, must all be addressed along with data management and the finite element problem itself.

These considerations are further complicated when portability between shared and distributed memory computers is considered.

In serial computation, the FORTRAN model of computing provided a mechanism for developing portable code. Portability is still a major issue in parallel computing. With rapidly changing architectures, it would be foolish to develop codes based on one MIMD configuration. Unfortunately, there is no simple model of all MIMD computers that has been mapped into a simple portable implementation language.

1.5 RESEARCH OBJECTIVES

The need to solve large structural engineering problems in the near future is the motivating factor for this investigation. This need calls for the development of a system that is capable of serving as an environment in which large scale engineering problems can be solved on a variety of MIMD machines. The objectives of the research are to develop a prototype system that provides engineering programmers with an easy and efficient means of developing large scale engineering software. It is imperative that the resulting engineering systems be portable, easy to program and maintain, and capable of exploiting the computational power offered by the next generation of computers. It is expected that such software systems will last for at least one or two decades.

The overall goal of this research is not simply to develop a computer program that will solve problems faster. The main emphasis is focused on developing a new approach to applying advanced computing technology to solve real engineering problems. The objectives are in keeping with the recent observations and recommendations of the Scientific Supercomputer Subcommittee [74]. They identified the need for high-level representation of algorithms with no machine-dependent semantics. Furthermore, it was noted that certain aspects such as the details of memory hierarchy management - vector registers, cache, virtual memory - should not be part of the user's program, but instead be optimized by the software itself. These notions have been the foundation of this research.

The scope of this research is restricted to the development of the conceptual design of the environment and the implementation of a prototype on a distributed memory system. A network of Apollo workstations was selected for the implementation of the prototype because of accessibility to the hardware and existing soflware tools.

1.6 OUTLINE OF REPORT

The report is divided into ten chapters. The first has been an introduction to the problem from an engineering point of view. However, the content of the report is interdisciplinary in nature. Hopefully, the result of the work will be used by engineers, but the concepts for achieving the results necessarily have a strong computer science flavor. Many engineers are not familiar with the problems associated with developing systems on MIMD computers. Hence, chapter two provides the reader with an introduction to the related issues, concepts, and terminology. Chapter three describes the evolution of a certain class of software environments that have been used successfully in the past on serial computers, and then develops the requirements for a new software environment for MIMD computers. Chapter four develops the general overview for an MIMD architecture independent environment that satisfies the needs expressed in chapter three. The environment calls for a three part system consisting of a Client program, multiple Server programs, and multiple Disk Manager programs. Chapters five through seven describe the functional design of each type of program in the environment, how they work, how they relate to each other, and,to how they will relate to the application programs that they will support. Chapter eight addresses the functional details, and the reasoning used to arrive at the details of some of the major components of the three programs. Special emphasis is placed on the development of a virtual address space for the distributed memory architectures and how it can also be mapped to shared memory architectures. Chapter nine presents the details of two example applications problems and the corresponding programs that were used to solve them using a prototype of the environment. Finally, chapter ten presents a summary of the findings and identifies the potential for continued work in this area.

1.7 ACRONYMS

The following is a list of acronyms that are used in this report. A description of each term is provided where it is first referenced in the text.

CPS	Concurrent Programming System
CPU	Central Processing Unit
DBMS	Data Base Management System

DDL	Data Definition Language
DM	Disk Manager
DOF	Degree Of Freedom
GB	Gigabytes
GSS	Guided Self Scheduling
HLL	High Level Language
I/O	input and Output
IL	Intermediate Level
KB	Kilobytes
MB	Megabytes
MIMD	Multiple Instruction Multiple Data
MM	Memory Manager
OS	Operating System
P.A.	Page Allocation
RAM	Random Access Memory
UDP	User Datagram Protocol

CHAPTER 2

BACKGROUND

This chapter contains an overview of the computer related issues that arise when developing applications like the finite element method on MIMD computers. The purpose of this chapter is to familiarize non-computer-science readers with both the terminology used herein, and some of the more important programming issues.

2.1 SPEEDUP

The *actual speedup* is an indication of how well a given application performs on an MIMID machine. It is defined as the ratio of the time to execute an efficient serial program for an application, to the time to execute a parallel program for the same application on *n* processors identical to the serial processor. The wall-clock time is used in the speedup calculation because it measures the actual time required to run the application - it includes the time for I/O and any overhead that is incurred in order to execute concurrently.

This definition for actual speedup specifies the best serial time rather than the time required to run the application on one processor, because sometimes there is a disparity between the two values. For example, a given algorithm may be very inefficient when applied on one processor. If such an algorithm is used to establish the serial time, the computation of the actual speedup will be misleading because the actual speedup value may appear to be very good despite the algorithm being very poor in reality.

The equation for determining the *theoretical speedup* provides a very useful indication of the potential limits on performance. It is not unusual for the uninitiated to believe that increasing the number of processors working on an application will automatically decrease the wall-clock time for that application. Yet, the sustained rates for some applications tends to be between 5 and 15 percent of peak performance [48]. An examination of the equation for theoretical speedup gives an indication why that happens.

The potential for speedup in a given application is significantly influenced by the amount of code present that must be executed serially, and by the system overhead. Amdahl's law characterizes the possibility for improved performance, or speedup, that can be exploited for a given application. The law very simply states that if a machine has two modes of operation; a slow and fast mode, the overall performance will be heavily influenced by the slower mode. In the context of MIMD computers, the slow mode corresponds to the execution of serial parts of the code on a single processor, while the fast mode corresponds to concurrent execution of those parts of the code that can be done in parallel on multiple processors. A simplified version of Ware's equation (eqn 2.1-1) below, defines the theoretical speedup of an application given the percentage of code that can be executed concurrently and the number of processors used.

$$S(n,f) = \frac{1}{1-(1-\frac{1}{n})f} \qquad \text{(eqn 2.1-1)}$$

where; S = speedup,
f = fraction of the number of instructions executed in parallel,
n = number of processors.

Equation 2.1-1 is overly simplistic and optimistic because it excludes the overhead required to operate with n processors. However, the point of this equation is that the amount of code executed in the serial mode must be minimized if any gains are to be made. For example, if the fraction of parallel code (f) is 50%, the maximum speedup will approach a factor of 2 as n approaches infinity. When the fraction of parallel code (f) is 90%, the speedup is only 5.3 for 10 processors, and it approaches 10 in the limit. This is illustrated in figure 2.1-1.

2.2 TASKS, DATA, AND GRANULARITY

A *task* is a unit of work that consists of one or more computer instructions. An example of a task is the instructions that make up the body of a DO LOOP. DO LOOPs that are defined so that each iteration can be executed concurrently are called *parallel do loops* herein. A *data object*, or just *object*, can consist of anything from a single word of data , to a matrix, to an entire tree structure. The term *granularity* refers to the size of a task or a data object, and the terms *Fine, Medium and Coarse* are often used to characterize the granularity.

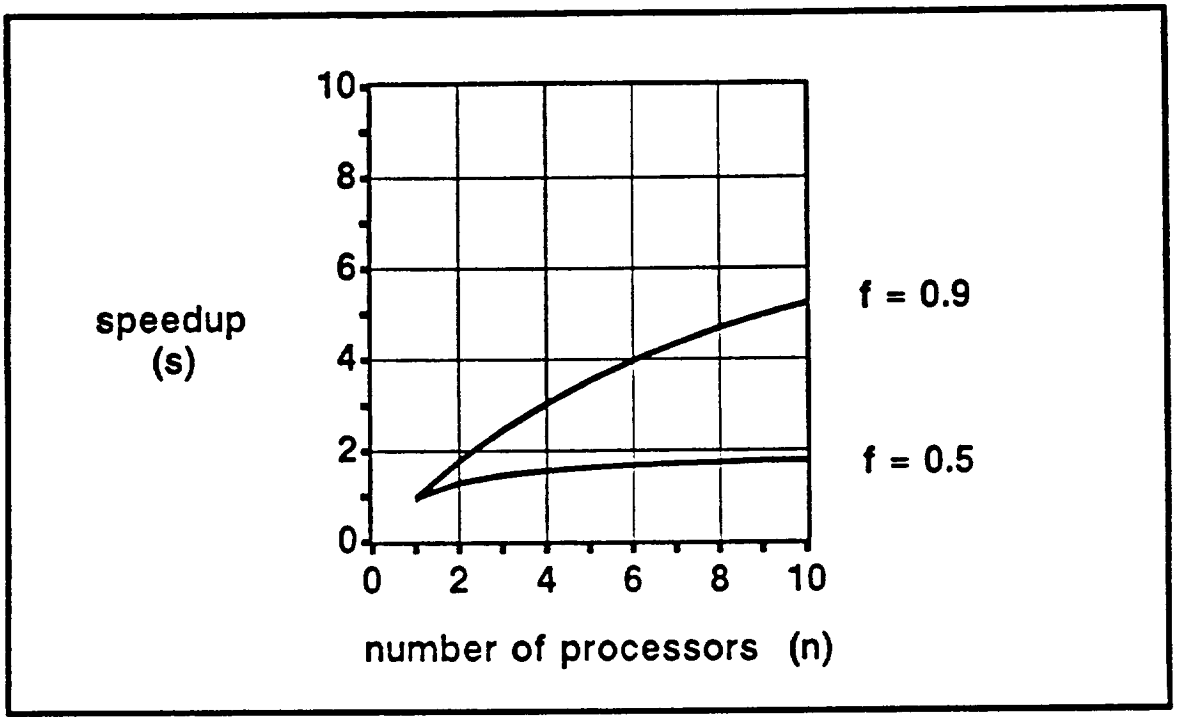


Figure 2.1-1 Illustration of Ware's Equation

Both data and task granularity influence the performance of an application program. The granularity of tasks is an important factor to consider when examining the issues related to load balancing, task scheduling, and task synchronization. Further details of these subjects are described in section 2.3. Data granularity affects the issues related to the access and modification of data. The performance issues affected by data granularity are discussed in section 2.4.

2.3 SCHEDULING, SYNCHRONIZATION AND LOAD BALANCING

The stiffness matrix of a structural model can be generated by concurrently executing tasks at various granularity levels. For example, the tasks can consist of computing the contributions to the matrix at the: substructure level (coarse grained tasks); element level (medium grained tasks); or integration point level (fine grained tasks). Alternatively, a task can consist of computing two or more substructures, elements or integration points. How does a software engineer decide which is optimum? The answer depends on the number and type of processors that will execute the job. Tasks consisting of large substructure calculations are best suited to architectures with a few very sophisticated processors (e.g. Cray-XMP). At the other extreme, computing the stiffness contribution at the integration point level is best suited to a machine that has thousands of simple processors (e.g. Connection Machine). For most commercially available multiprocessor computers (Alliant, Intel-iPSC), an ideal task would consist of one or more element calculations - the ideal number of elements is a function of the computational complexity of the element and the speed of the processors.

The *Efficiency* of an application is defined as the actual speedup of the application divided by the number of processors used for that application. *Processor Utilization* is the average percentage of processor time used. By definition, optimum efficiency is obtained when the speedup is maximized. The case of optimum efficiency may not coincide with the situation in which the system has the highest value of processor utilization or when optimum load balancing occurs. Generally, optimum efficiency occurs when minimal time is spent waiting for processors to complete their tasks and synchronize with each other. Hence, the efficiency of an application program on a given MIMD system depends on the ability to schedule and synchronize the execution of tasks effectively and efficiently.

Load Balancing is the ability to assign tasks to all the processors so that they are utilized evenly. Optimal load balancing of an algorithm generally does not guarantee the best system utilization because of the overhead and uncertainties associated with task scheduling and synchronization.

The process of allocating tasks to the processors is known as scheduling. There are two general methods of determining how to schedule tasks. *Static scheduling (*also called pre-scheduling) means that all decisions are done at compile time. *Dynamic scheduling* (also called self-scheduling) means that all decisions are made at run time.

There are many questions that need to be answered in order to schedule tasks effectively. For instance, when allocating iterations of a parallel do-loop to the processors on an MIMD system, how many iterations should each processor get? What happens when the number of instructions executed increases or decreases with every iteration? If there are conditional statements within the loop body, how much time will be spent processing an iteration? How should the scheduling algorithm incorporate disparities in processor performance? From a performance point-of-view, is it better to avoid the overhead of scheduling and synchronization by executing the loop serially or on a limited number of processors?

The programming aspects of synchronization that occur during the assembly procedure appear to be easy. The concurrent computation of the elements or substructures can be specified via simple parallel do loops. Each outer loop can do a substructure, each second level loop an element, and each third level loop an integration point on the element. The implication of the end of a parallel inner loop is that all processors working on the loop must complete their tasks prior to resuming the corresponding next higher level loop. To accomplish this, a synchronization operation known as a *barrier* is established. Synchronization mechanisms are often automatically generated by the compiler using the hardware of the machine. However, this feature is not available on all machines, especially loosely coupled processor systems. Therefore, the programmer has to explicitly define synchronization operations within the program.

Barriers are also used at a higher level. For example, when using direct methods to solve the stiffness equations, it is necessary to wait until the contribution of the last element has been made in the stiffness matrix before beginning to solve the system of equations. That wait constitutes a barrier.

Scheduling the tasks needed to solve the system of equations is much more difficult than in the assembly process because of the complex data dependencies involved with decomposition, forward elimination, and back substitution. Various types of *task related barriers* and *object related barriers* are needed. The latter are barriers that prevent a task from accessing a data object before it should. This is done to insure that correct results will be obtained. The details of this concept are discussed in chapter nine.

2.4 DATA STRUCTURES AND COHERENCE

In order to ensure that correct results are obtained, *data coherence* must be maintained during the execution of a job. Data coherence means that multiple copies of a datum must be consistent when referenced by multiple processors. This implies that a datum cannot be modified by two or more processors simultaneously. In engineering programs, a data object often consists of more than one word. Typically, the objects are vectors and matrices. There are two general ways of maintaining coherence:

a) objects to be modified are only available to one processor at a time -- object protection;

a) programmers write their code so that processors operate on mutually exclusive sections of the same object — programmed protection.

The ability to perform operations concurrently is a function of how well conflicts related to access of shared data objects can be avoided. A data dependency analysis may determine how to synchronize tasks and operations so that conflicts are avoided. Sometimes, the dependency analysis cannot yield conclusive results at compile time because the nature of the conflict is a run-time problem; i.e., it is a function of the data used in the program. For example, when a global stiffness matrix is generated, the contributions of a given element will often coincide with those of adjacent elements.

Figure 2.4-1 illustrates the conflicts that can occur when assembling the stiffness matrix. If one assumes that the contribution of one of the four elements in the model on the left of the figure is handled by a different processor, the figure on the right side shows areas of the matrix accessed by two or more processors. In order to avoid conflicts the programmer

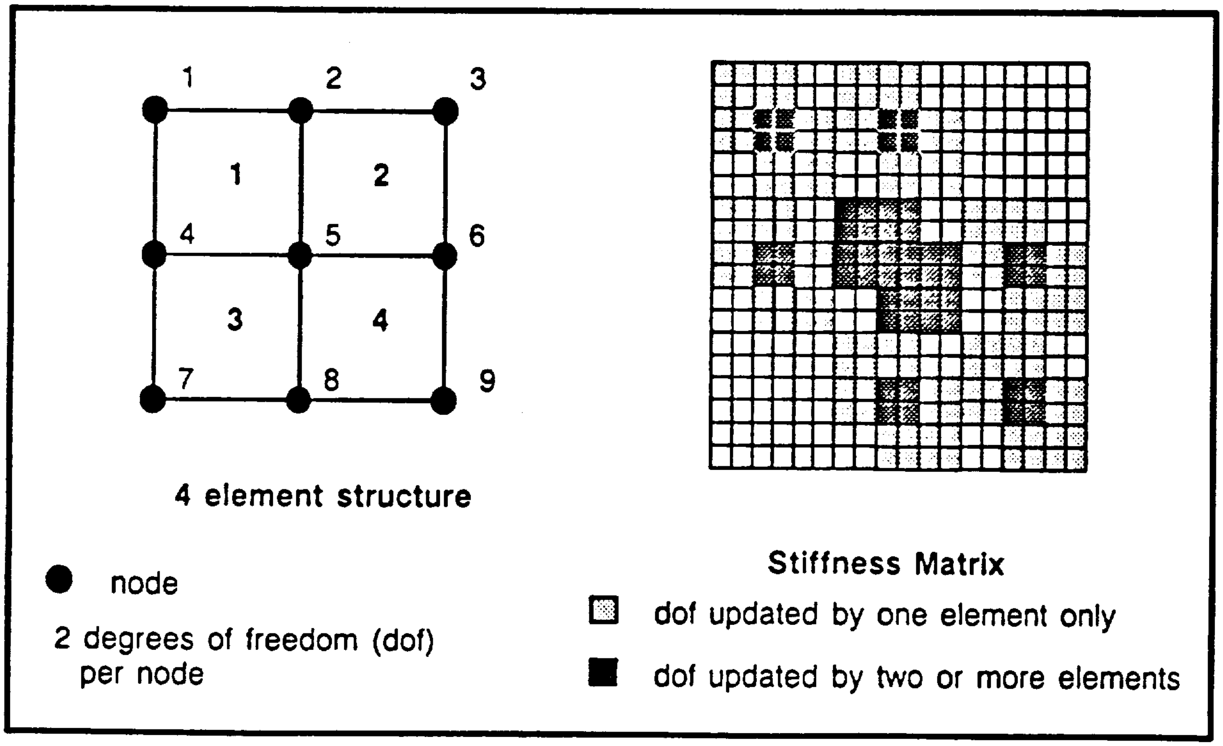

1
2
3
1
2
4
5
6
3
4
7
8
9
4 element structure
● node
2 degrees of freedom (dof) per node
Stiffness Matrix
▨ dof updated by one element only
■ dof updated by two or more elements

Figure 2.4-1 Data Conflicts During Stiffness Matrix Generation

would need a priori knowledge of the structure or develop an algorithm that sorts the elements into groups of non-conflicting element calculations.

To further illustrate this problem, consider how the program would be written. If the stiffness matrix is defined as a single physical data object, and if any processor can make an arbitrary modification to the matrix via an element contribution, then only one processor can be allowed to modify the matrix at any given time to ensure data coherence. If other processors wish to update it they must wait their turn; there is very little room for concurrency.

On a shared memory machine, a *critical section* can be used in the program when the stiffness matrix is updated. A critical section is analogous to a lock; one processor obtains a lock on the object thereby preventing all other processors from accessing the object. On a distributed memory machine, two alternatives exist; a) pass the stiffness matrix around from processor to processor or b), all of the processors can be programmed to send their element contributions to a given processor that will update the global matrix. In general, both of these processes degrade overall performance considerably.

If the stiffness matrix can subdivided into smaller physical objects like submatrices, the potential for concurrency is improved because multiple processors can simultaneously update different parts of the stiffness matrix. This approach does not eliminate the data coherence problem because potentially there will always be submatrices that must be accessed by more than one processor.

In distributed memory systems, concurrent operations on the stiffness matrix can only proceed if the matrix is split into smaller objects, thereby permitting different parts of the stiffness matrix to migrate to various processors in the network. The engineer is still faced with maintaining data coherence in the program because a given block may be updated by two or more processors. Maintaining data coherence in a distributed memory machine is much more difficult to accomplish than in a shared memory machine because the address space is fragmented. This implies that a given processor must unequivocally establish that no other processor is using the object that is about to be modified. This complicates programming the application considerably.

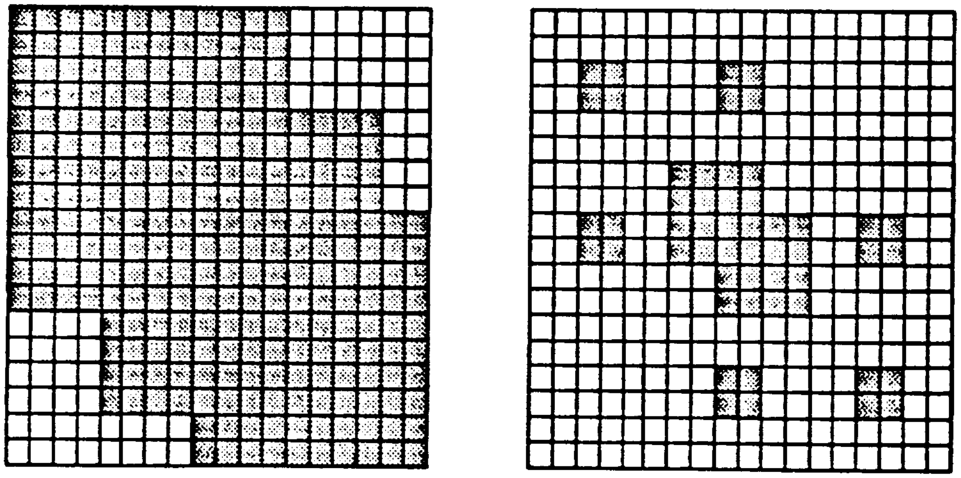
If a data object is subdivided into blocks, how large should the blocks be? The disadvantage of using large blocks instead of smaller blocks is the greater potential for a conflict to occur. The disadvantage of using smaller blocks instead of larger blocks is the increased overhead of managing more data objects. Figure 2.4-2 illustrates the differences between using small and large blocks for the sample problem described in figure 2.4-1. The figure shows the effect of using two different block sizes. When the two by two blocking factor is used, fewer conflicts occur than in the four by four scheme.

Hierarchical data structures that can support multiple levels of data granularity are ideally suited for parallel computation because the concurrency can be exploited at many levels.

2.5 DATA AND MEMORY MANAGEMENT

Forming and managing the complex data structures that result when a programmer segments large data objects is called data base management. It consists of two parts; managing the logical data structures, and managing the memory that holds the data objects.

Data and memory management are problems encountered with serial as well as parallel systems. Most engineering programmers have difficulty with problems requiring large data spaces. The data spaces envisioned for the future are two orders of magnitude larger



Stiffness Matrix Conflicts
4x4 blocking factor

Stiffness Matrix Conflicts
2x2 blocking factor

Figure 2.4-2 Data Conflicts for the sample problem of figure 2.4-1

than for today's problems; the fact that programmers may need to handle a disparity of memory types complicates the problem further.

It is important to recognize that engineering data base management is not the same as classical data base management as described in the computer science text books. Engineers must work with complex array structures - not simple relations and tuples. Engineers typically create a large data space in a short period of time; rapid access and high throughput are very important. The data space often disappears immediately after problem solution. Classical DBMS deals with large data spaces built over time and maintained forever. Security and longevity are very important.

The mappings inherent in classical DBMS are needed to provide flexibility; they are too cumbersome and inefficient for the kinds of engineering problems envisioned herein. Objects must be passed directly to the application. This results in some loss in flexibility and security; these disadvantages are offset by corresponding gains in throughput.

2.6 OTHER FACTORS INFLUENCING PERFORMANCE

There are several performance factors that must be considered when developing parallel programs [77]. These factors are unique to the machine. They are:

- *computational bandwidth*: how fast data can be processed by the cpu,
- *memory bandwidth*: how fast data can be processed by the memory,
- *communication bandwidth*: how fast data can be processed by the interconnection network,
- cache memory or vector registers: high speed buffer areas,
- and the amount of primary and secondary storage.

When developing a parallel program, the algorithm can be "fine tuned" to the machine architecture in order to exploit the machine to the fullest potential. References to data objects are programmed in a manner that reflect the memory bandwidth and the organization of the available cache, primary and secondary storage. For distributed memory machines, the communication bandwidth is a very important factor. If is is slow relative to the processor bandwidth, the program must be designed with a minimal amount of inter-processor communication (message passing).

Since no two machines are likely to have identical specifications, it is reasonable to expect that the performance of a given application on two machines will be different. It is important nonetheless that a given application should be somewhat comparable and easy to port, especially when considering the spiraling costs of software development.

In the future, it will become necessary to develop software that will span different types of machine architectures. Algorithms and data structures must be expressed independently of the architecture.

CHAPTER 3

MIMD ENVIRONMENTAL ISSUES AND NEEDS

This chapter develops the concept of a software environment for MIMD computing. The benefits of using various types of tools that can exist within the environment are analyzed. The pros and cons of the various alternatives are outlined. The language layer approach is selected.

3.1 PROGRAMMING ENVIRONMENTS

3.1.1 Introduction

The previous chapter described several factors that need to be considered when developing finite element (engineering in general) software on MIMD computers. The software engineer is responsible for correctly integrating all of these concepts into his system. The magnitude of the problem is expressed visually in figure 3.1.1-1. Engineers can express the application software in terms of mathematical algorithms; they probably can think of it in terms of high level tasks that can be done concurrently, and in terms of the associated data objects; they have some knowledge of the MIMD related problems that were discussed in chapter two. The problem is to synthesize all of that information, and to convert it to a working engineering system on specific MIMD hardware. The mapping from software concept to hardware is accomplished via the programming environment, which acts as the programmers software window to the hardware. Typically, the programming environment delivered with the hardware consists of an operating system, compilers, and various tools to achieve concurrency and parallelism in programs.

On serial computers the "standard environments" are often augmented with broader based environments that are more in tune to specific applications. This chapter will make the case for extending such environments to MIMD systems too.

3.1.2 Why We Need a Broader Based Environment

The examples in the previous chapter illustrated that there is a significant amount of large grained parallelism that exists within the finite element method. It is difficult to exploit

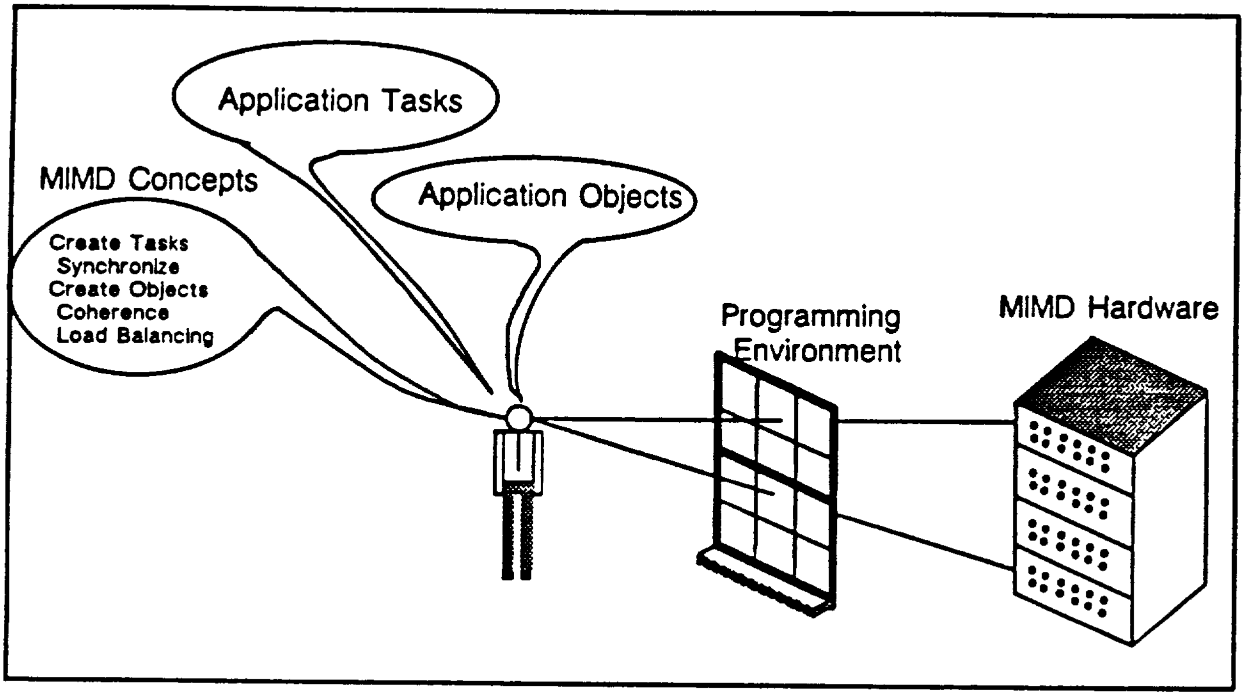


Figure 3.1.1-1 Function of the Programming Environment

this type of parallelism without detailed knowledge of the problem. That depth of knowledge is generally not available to compilers -- even those capable of performing an extensive data dependency analysis. Consequently, exploitation of this concurrency is generally left to the software engineer. He must simultaneously contend with the engineering problem, all aspects of task control, and all aspects of data and memory management.

The task scheduling mechanisms for loosely-coupled machines are significantly different from those on tightly coupled machines. There is even a disparity within the group of tightly-coupled machines. The data dependency-synchronization approach cannot be incorporated on loosely-coupled systems because of the segmented address space combined with the hardware and communications overhead. Hence, developing portable application codes that exploit this concurrency is a very difficult task. What works on one machine will often not work on another.

In order to overcome these difficulties, an environment that will facilitate developing machine and architecture independent application code is needed. The engineer should be able to express the problem, generic information about concurrency, and which data objects are to be used, in a procedural language. The environment should provide the engineer with a uniform approach to programming the application on all MIMD computers. The system that supports the language must be responsible for mapping the application to the hardware architecture of the target machine. It should handle the scheduling and synchronization of tasks in addition to resolving data coherence conflicts and managing memory for a variety of hardware architectures. In effect, the programmers view of the system as seen through the window of figure 3.1.1-1 should be invariant and opaque to the hardware behind the window.

Hence, the object of this research is to develop a new programming approach by prototyping a software environment that does all of the above on a distributed memory MIMD system. Unlike most other work in this area that primarily emphasizes task control, data and memory management are treated as crucial aspects of the problem in this concept.

3.2 TYPES OF TOOLS IN THE ENVIRONMENT

There are several alternatives for identifying and expressing the concurrency in an existing application developed in a serial programming language. They include:

- modifying / extending the programming language to incorporate the requisite features to express concurrency (e.g. FORTRAN and FORCE[44]);
- developing compilers that are capable of identifying and optimizing the concurrency in the program automatically (e.g. Parafrase [17]);
- re-programming the application using new languages that already have the constructs needed (e.g. ADA) or developing a new programming language.

Various solutions to the first method are presented in [31], [44]. They include the use of a language layer, new library routines, and new constructs and features required to identify concurrency and address the related aspects. McGraw [60] identifies some of the advantages and disadvantages of such techniques; minimal changes are required in the application.

The use of special purpose routines leaves the original language and its compiler untouched. However, the compilers lack of knowledge about how parallelism is being used in the program results in some difficulties. These difficulties include the analysis of how data is being shared between tasks and, the possibility of incorrect optimization. Furthermore, incorrect code can easily be written because the programmer is only weakly restricted in the ways synchronization constructs and shared data can be used.

The FORCE [44] is an example of a special purpose software tool that is used to exploit concurrency within an application. The FORCE insulates the programmer from process management, however the programmer is responsible for the data coherence issues of parallel programming. The FORCE consists of a set of FORTRAN 77 callable macros that support the parallel execution of DO loops using pre-scheduling and self-scheduling schemes. The FORCE includes constructs to allow for mutual exclusion, synchronization, and/or sequential execution when necessary, and constructs for data based control of execution. In this approach, the programmer is largely responsible for identifying the concurrency within the application.

Another alternative that can be used to identify and express the concurrency within an application program is to extend the programming language by incorporating parallel constructs. Hardware manufacturers provide unique language extensions for the software they support. For example, the language extensions for FORTRAN on the Cray XMP and the ALLIANT are not the same. Hence, language extensions are not standard; neither are they machine or architecture-independent.

Significant efforts have been directed towards developing special compilers that automatically identify concurrency in existing programs [17], [60]. This approach requires the use of an extremely sophisticated compiler. Most of the emphasis of this technique is focused on the analysis of do-loops. The question of how much concurrency the compiler can detect still needs to be answered. This depends on both compiler technology, and the nature of the program being analyzed. The performance of a program depends on detecting data dependencies so that the instructions can be scheduled and synchronized in order to produce correct results while still maximizing the amount of concurrency and vectorization within the program.

A data dependency is an implicit sequential ordering of the access to a data element. The order of access must be enforced to ensure that correct results are obtained. Hence, dependencies can limit the amount of parallelism that exists in a program. Data dependencies are very difficult to detect when multiple levels of subroutine calls are present within the program. This is exactly the problem when identifying coarse-grained concurrency in a program. A well written program will have many levels of subprograms: the potential for concurrency is not obvious to a compiler. In such cases, the programmer has to use compiler directives. The directives assist the compiler in generating the scheduling and synchronization instructions for concurrent operations. Compiler directives, like language extensions, are very machine dependent.

Yet another alternative is to develop the program in a new language e.g. ADA. The use of ADA for scientific applications has also been considered as a viable alternative programming language. Its success within the scientific community depends on whether mathematical and scientific libraries can be developed, and on its future popularity among programmers [73], [84].

Finally, developing a new procedural language can be a formidable task, depending on how sophisticated the language constructs need to be. Such an option is very manpower-intensive and hence extremely expensive. Given the tremendous investment in existing application codes, this option must be viewed either as a last resort, or as a means for developing ideas on how existing systems would gradually "evolve" into structures more amenable for parallel processing. The advantage of developing a new language is that it can be tuned to enhance possibilities for expressing algorithms that contain concurrency. More detailed discussions related to developing a new language can be found in [60]. Since hardware architectures may vary dramatically, it may be very difficult to develop a compiler that will optimize the new language for high performance on each target machine.

None of the approaches described above addresses the issues related to management of the large complex data spaces associated with engineering systems.

A slight variation on the new language theme is to use a language layer approach. The application is developed using two languages. The first language, a High Level Language (HLL), is both architecture and machine independent. It is used to specify the overall algorithm in terms of tasks, objects used by the tasks, and the potential for coarse grained concurrency of tasks. The HLL program also acts as a driver for routines written in a lower level procedural language.

The "guts" of the program are implemented in a procedural language like FORTRAN. These routines become the workers for the tasks defined in the HLL. They receive data objects as arguments and perform the needed operations. Consistency of data objects and proper task synchronization is guaranteed by the HLL program that invokes the procedural program.

The procedural subprograms can be "fine tuned" to a given machine by taking advantage of the capabilities of the procedural language compiler on the target machine. This can be done by either using compiler directives for the target machine (machine dependent) or by introducing machine independent compiler directives that can be filtered out as the application is moved to a new target computer.

This two level approach provides a number of advantages over the approaches described earlier. Application programs written in the HLL program are machine and architecture independent. They expresses a solution to the application in a manner that would work on any MIMD configuration. Thus it eliminates one of the major objections to using other "more standard" techniques. In addition, since the HLL is specific to the MIMD problem, it can be designed with features that relieve the programmer of the need to consider the details of how tasks are assigned to processors, and how to express the details of the synchronization necessary to make the program work properly. These concepts can be implicitly imbedded in the language constructs, hence the programmer need only concern himself with the application problem. Such language constructs already exist in application languages like SISAL, FP and KRC [60].

The language layer approach is very flexible, permitting one to easily experiment and implement new concepts in the language and the underlying support system. It facilitates the development of prototype systems for MIMD computers. Hence it was the alternative selected for the environment developed herein. The next section describes how the language layer concept has evolved over the years.

3.3 EVOLUTION OF LANGUAGE LAYER ENVIRONMENTS

The purpose of the software environment is to provide the programmer with a consistent view of the hardware regardless of what it really looks like. This view is referred to as the virtual machine because such a hardware configuration generally does not exist. The programmer develops the application program for the virtual machine. The environment is responsible for mapping the program written for the virtual machine onto the actual hardware provided.

In order to give the reader a better appreciation of what is involved in supporting the programmers view of the system, a brief history of the changes that have taken place puts things in perspective. Figure 3.3-1 illustrates how the programmers view of the system has evolved over the years. It also shows how the language layer concept has evolved based on the engineers need to solve large problems.

In the 1960's, the view of the hardware consisted of a single processor that operated on data. The processor executed the program and referenced data that resided within a data space. This is shown as the figure at the top of figure 3.3-1. Engineers traditionally programmed their applications in FORTRAN. The engineer's expectation of the computer's capability grew with time; he attacked larger and more complex problems. The programs that had to be developed were more complex and sophisticated. The application problems required the manipulation of large volumes of data -- large enough to exceed the storage capacity of main memory by several orders of magnitude.

In order to solve these problems, more complex data structuring techniques were developed and a High Level Language Layer (HLL) was used to access and manipulate data. The data structures were not available in earlier procedural languages. The increased complexity and details of managing data were hidden from the engineer by the environment. The HLL served as an interface between the procedural subprograms and the new sophisticated data structures. Conceptually, operations in the HLL were mapped to procedural subprograms as shown in the simple example illustrated in figure 3.3-2.

The figure shows how a high level operation called *ADD_VEC* will add *N* elements of two vectors *B* and *C*. The result of the addition is stored in vector *A*. The actual vector addition operation is performed in the FORTRAN subroutine called *addvec*. The data objects referenced in the HLL operation (*A*, *B*, *C* and *N*) are automatically made available to the FORTRAN subroutine by the programming environment. In this concept, both the data definition and the actual data are totally separate from the procedures that operates on them.

In the Language Layer approach, data objects were naturally segmented into blocks of data in order to facilitate manipulating large data objects. For example, a logical data object

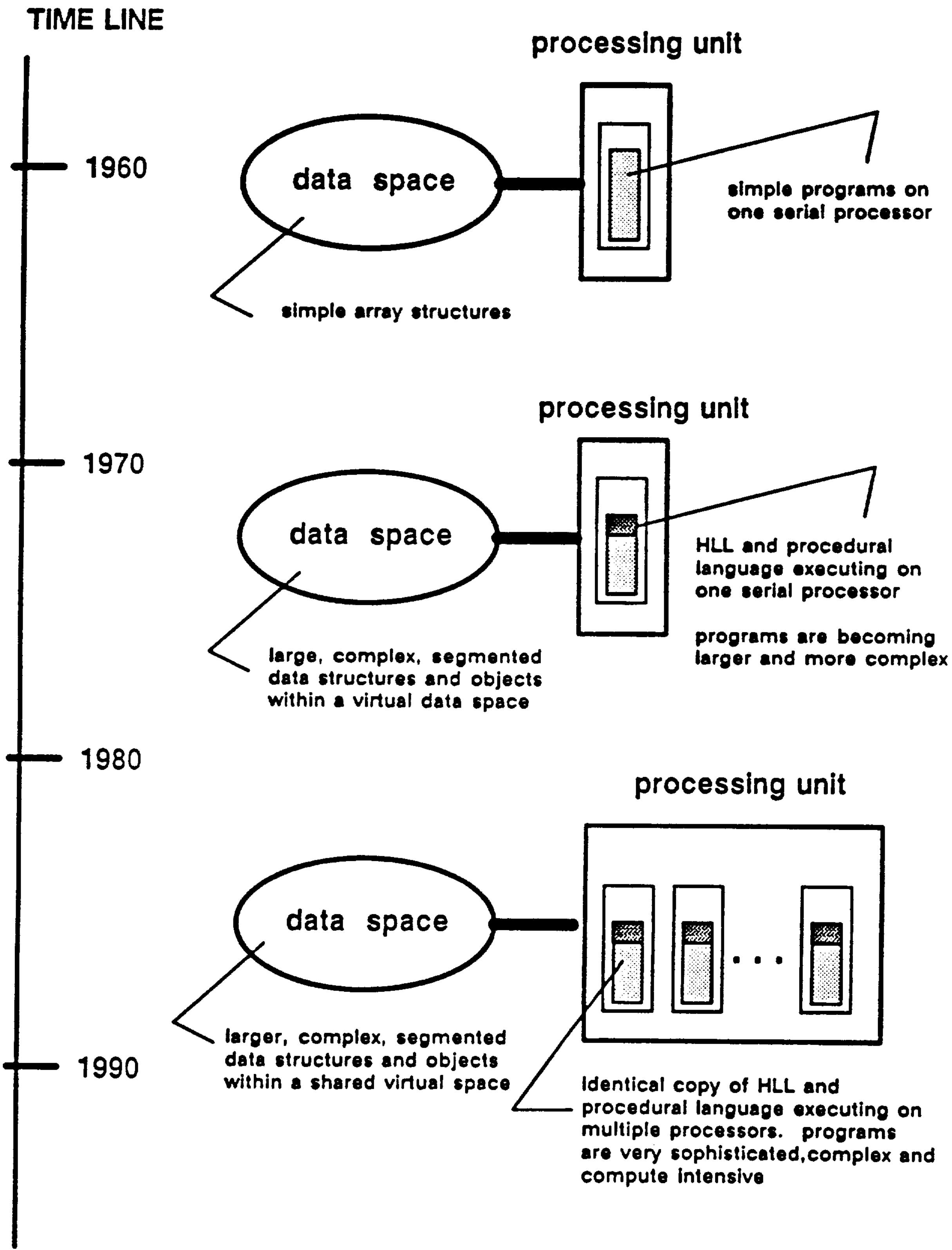

TIME LINE
1960
1970
1980
1990
processing unit
data space
simple programs on one serial processor
simple array structures
processing unit
data space
HLL and procedural language executing on one serial processor
programs are becoming larger and more complex
large, complex, segmented data structures and objects within a virtual data space
processing unit
data space
. . .
larger, complex, segmented data structures and objects within a shared virtual space
identical copy of HLL and procedural language executing on multiple processors. programs are very sophisticated,complex and compute intensive

Figure 3.3-1 Evolution of Programmer View of the System

HLL Code

```
...
...
ADD_VEC(A, B, C, N)
...
```

Data Definition

```
INTEGER N
A VECTOR REAL N
B VECTOR REAL N
C VECTOR REAL N
```

FORTRAN subroutine

```
      subroutine addvec(a,b,c,n)

      real a(n), b(n), c(n)

      do 100 i = 1, n
              a(i) = b(i) + c(i)
100   continue

      return
      end
```

Figure 3.3-2 HLL instruction and corresponding FORTRAN subprogram

like the stiffness matrix of a structure was subdivided into multiple physical objects (called hypermatrices) so that only those parts of the object that were being operated on had to be made available in memory at any point in time. Thus, the environment helped circumvent the limitations on memory space in that computing era.

In recent years, advances in hardware technology have lead to the development of processors with large amounts of main memory employing virtual memory techniques embedded in hardware. The advances in software technology have lead to new languages that are portable and can be used to develop sophisticated data structures. Why haven't engineers simply re-developed their systems in these new languages? How expensive is it to switch and what are the benefits of doing so? It appears that the re-development costs are very high and the potential benefits in terms of improved performance do not warrant the changes. The advantages of the improved hardware technology can be exploited simply by fine tuning the environments to the hardware. Some new languages like "C" have incorporated more sophisticated data structures that are mapped to the virtual memory of the hardware. Unfortunately the hardware accessible data space is still inadequate for the problems envisioned herein. That, combined with the fact that "C" is still a relatively low level procedural language, has not eliminated the need for these software environments.

The types of engineering problems encountered today are much larger and much more complex that at any point in history. The computational requirements of these problems have exceeded the capacity provided by most serial processor systems, and engineers must resort to the use of parallel computers to achieve their objectives. Unfortunately, multiprocessor computers are much more difficult to program than serial computers. Developing architecture independent software is even more difficult because of the variety of hardware configurations; hence the current generation of applications software for parallel computers tends to be very machine dependent and expensive to develop. There is a need for more suitable software environments on MIMD systems.

The environment proposed herein is the third stage shown in figure 3.3-1. The programmers view of the processing unit consists of multiple computing elements. In the new configuration, each processor can be executing a different part of the same program. The organization of the data space remains the same, however the underlying support systems is much more sophisticated in order to support the more complex and widely varying MIMD hardware configurations.

The changes in the hardware effect three distinct areas of the environment; they give rise to the following concerns:

- How does one express concurrency in the high level language -- that is, how does the engineer identify the concurrency within the application program? New control constructs are required for this purpose; what should these constructs look like?
- How should data objects be defined? What is appropriate for the new hardware?
- How will the new the new language constructs and data objects be supported, and what are the implications of the changes with regard to the underlying MIMD system? This is the most difficult aspect of the software environment.

Each of these issues are discussed further in the following sections. The issues related to what the programmer "sees" are discussed in section 3.4. The issues related to what goes on behind the scenes — i.e. what the environment should provide — are discussed in section 3.5.

3.4 PROGRAMMING ISSUES

This section describes the issues that directly effect the programmer. These issues are primarily related to how the engineer can program the application. Hence, they are concerned with the High Level Language — how is it used, and what features are required in the HLL? To better understand the issues, it is necessary to define the purpose of the HLL and analyze each objective that must be accomplished through it. The purpose of HLL will be to provide programmers with a mechanism to:

- define data structures and identify data objects,
- define procedures that reference those data objects,
- identify the potential concurrency within the application,
- pass control to the application subprograms written in a procedural language.

3.4.1 Data Definition Issues

Ideally a programmer can define data objects in a manner that is natural to the physical problem. The definition should be independent of the hardware and details of the algorithm. Unfortunately, that is not always possible or desirable. The mapping of the data objects like the stiffness matrix K, to a system's memory/ disk system has profound effects on performance.

Consider, for example, a system that has only multi element objects of a class termed "matrix." Then the K matrix will need to be defined a single huge matrix. Three problems arise. First, where can we put it? -- most hardware system cannot accommodate such a large data object. Second, most of it is empty -- K is sparse. Third, if K is treated as a single object, in order to maintain data consistency, only one processor can modify the object at any given time -- therefore the potential for concurrency is restricted.

Environments of the 1970's handled the first two problems (there was no concurrency in these systems) by introducing additional classes of objects such as pointers. With such a vehicle, objects such as K could be defined as a tree data structure containing only non-zero submatrices (nodal objects) and pointer objects that tie the submatrices together to form the unit called K. This type of data structure is referred to a "Hypermatrix." It solved two of the three problems mentioned above. One can access a small part of K and, only the non empty parts need to exist.

The hypermatrix concept also has direct benefits related to concurrency. If the nodal submatrices are directly addressable objects in the data structure called K, different parts of K can be used simultaneously by different processors in an MIMD system. Thus any new environment that treats large objects as sets of discrete objects will benefit significantly. Hierarchical data structures seem to be well suited to parallel processing applications. The objects at lower levels in the hierarchy imply a greater potential for concurrent operations because they can be referenced simultaneously. (Note that the root node can still be a point of conflict.)

Therefore, it can be concluded that the high level Data Definition Language (DDL) in the environment must be flexible so that the programmer can create a variety of data structures. Among the features required is the ability to develop multi-dimensional objects, arrays, pointers, and matrices; it must also provide support for hierarchical and relational data structures.

3.4.2 HLL Control Structure

In order to operate on the data objects, a host language is needed. The control structures in the HLL must incorporate the basic IF-THEN-ELSE, DO WHILE, DO FOR, CASE and CALL constructs in order to develop general purpose programs. Ideally, it would also support object oriented programming constructs. However, the latter can be considered an even higher level language that could be based on the HLL. These features (excluding the OOP constructs) have already been incorporated in existing software environments. In order to support parallel computing new environments must also provide the programmer with machine independent constructs to express the concurrency in the algorithm. These constructs must be simple and easy to use.

A program (or process) can be decomposed into a series of tasks, some of which can be executed concurrently. A task is simply one or more instructions, hence the size of a task can vary considerably. Since the potential for concurrency exists in do-loops, and in portions of code that can be divided into mutually exclusive tasks — a natural approach is to simply extend the syntax of existing do-loop structures and to provide the programmer with the ability to define tasks explicitly. The synchronization of tasks requires the use of barriers [31], or semaphores [22] and monitors [56]. The details of synchronization and scheduling can be incorporated into the control structures that identify concurrency so that they are hidden from the programmer.

Figures 3.4.2-1 and 3.4.2-2 illustrate two simple methods of expressing concurrent operations independently of the architecture. In figure 3.4.2-1, two mutually exclusive tasks are explicitly defined within the DO IN PARALLEL and ENDPAR statements. The TASKBEG and TASKEND statements are used to mark the beginning and end of the tasks. These tasks are embedded within the "DO IN PARALLEL-ENDPAR" statements implying that the tasks can be executed independently of each other

```
DO IN PARALLEL

    TASKBEG    GRANULARITY 5
           statement 1.1
           . . .
           statement 1.n
    TASKEND

    TASKBEG    GRANULARITY 5
           statement 2.1
           . . .
           statement 2.n
    TASKEND

ENDPAR
```

Figure 3.4.2-1 Mutually Exclusive Tasks to be Executed Concurrently.

```
DO FOR I = M, N, K IN PARALLEL GRANULARITY 10

        statement 1
        . . .
        statement n

ENDDO
```

Figure 3.4.2-2 Parallel Do-Loop

Figure 3.4.2-2 shows the syntax for specifying a parallel do-loop. M, N and K are the start, end, and incremental-step values of the loop index I, respectively. The group of statements enclosed between the DO FOR and ENDDO constitute a task. Tasks can contain subtasks and therefore, multiple levels of tasks can be defined.

To facilitate programming, the programmer should not have to specify the assignment of tasks to processors. The "run time system" that executes the instructions should manage this responsibility. The implications of this feature are more profound for distributed memory machines than for shared memory machines. Generally, in distributed systems, the programmer must program how to get tasks and the corresponding data objects to and from a given processor. In tightly coupled systems, the compiler generates the scheduling and synchronization primitives automatically. This disparity between the two systems makes portability very difficult. *In essence, the HLL environment makes programming a distributed memory system similar to programming a shared memory computer — and in doing so, facilitates portability.*

In order for the system to schedule concurrent tasks efficiently and perform load balancing, it is necessary to know the size of tasks. Since the computation of a given HLL operation can vary considerably, it is necessary for the programmer to provide the system with some information regarding the task sizes.

3.4.3 Task Granularity

The task scheduling mechanism within the run time system will assign tasks to processors based on the task granularity specified by the programmer as illustrated in figures

3.4.2-1 and 3.4.2-2. The GRANULARITY parameter is used by the programmer to indicate his best guess of the relative sizes of tasks within the program. It is called relative granularity herein and, by itself, is not adequate information to schedule tasks within a real system. It is a machine independent form of expressing the size of the tasks. Processors also have a granularity: some processors are suited to performing large tasks; others are suited for small tasks. Processor granularity is not part of the HLL; it is hardware dependent, and hence must be made transparent to the programmer. Interpreting the processor granularity and relative granularity must be the responsibility of the task scheduling algorithm. For example, the granularity values are used to determine the appropriate number of iterations of a parallel do-loop that should be assigned to a processor as a reasonable "unit of work."

3.4.4 References to Data Objects in the HLL - Data Access Issues

Ideally, a programmer only needs to reference a data object in the HLL in order to gain access to it. However, the object space is dynamic: objects need to be created; objects must move from processor to processor etc. The process of obtaining an object is not just a simple address lookup. It involves three steps, they include: data base management, memory management, and maintenance of data coherence. These functions should be performed automatically by the system. Then, referencing data objects in the dynamic object space is no more difficult than in any procedural language. The execution of the functions to support this is both complex and time consuming, and is crucial to the overall effectiveness of the system. However, like task scheduling, these operations are really transparent to the programmer; hence, further discussion is left to the section on run time support later in this chapter.

The primary purpose of the HLL is to gain access to objects so that they can be used by subprograms written in a procedural language. If the HLL does not indicate how the object will be used (e.g. if the object will be modified), that is, if the HLL only specifies where the data object is used; concurrent operations on the object cannot be performed because the system would have to assume that the data object will be modified in the HLL operation that referenced the object. This assumption inhibits concurrency because data coherence must be maintained, that is, only one processor can modify an object at any time. In order to improve the performance of the data and memory management systems, the programmer must specify in the HLL how the data objects will be used. Three data attributes are provided in the HLL for this purpose; they are:

- Access — indicates the type of access required: read or write,
- Locality — specifies the location of the data object: local or global memory,
- Release — specifies when the data object should be released: immediately after the HLL instruction is executed or after the duration of the task.

References to data objects in the HLL are "tagged" with the appropriate attributes. Figure 3.4.4-1 illustrates how data attributes are used in the HLL. The figure shows two HLL statements. The attributes that are identifiable are: r -- read access; g -- place object in global memory; i -- release the object immediately after the HLL instruction is executed.

In the first statement (s1), A, B, and C are vectors. The operation ADD_VEC will add vectors B and C and store the result in A. Therefore, A is obtained for write access while B and C are obtained for read access. In the second statement (s2), K_GLOBAL is the global stiffness matrix, K_ELEM is the element stiffness matrix, INCID contains the nodal incidences and ELNUM is the element number. In the operation ADD_STIFF, the contribution of the element stiffness matrix is added to the global stiffness matrix. In a hybrid memory system, K_GLOBAL and INCID can reside in global memory, hence they are tagged with the global memory attribute. Since K_GLOBAL will be modified, it is obtained for write access. Furthermore, K_GLOBAL is tagged with the immediate release (*i*) so that It can be made available to any other processor immediately after the ADD_STIFF operation is performed. The element stiffness matrix K_ELEM is located in local memory and accessed for "read". ELNUM is simply a data element that may be part of common and is not tagged with any data attributes.

```
s1:     ADD_VEC( A:w , B:r ,  C:r )

s2:     ADD_STIFF(  K_GLOBAL:gwi,  K_ELEM:rl, INCID:rg,  ELNUM)
```

Figure 3.4.4-1 Examples of using Data Attributes

3.4.5 Task Common Areas

In procedural languages there is always a mechanism to specify static global data objects that are accessible to more than one subprogram. In FORTRAN it is COMMON. In the MIMD environment, the concept of COMMON must be extended slightly. First, as with its counterpart on serial systems, it is global to all subprograms on a processor. However, it is considered to be private data relative to the same subprograms executing on other processors in the MIMD system. Thus, COMMON must really be interpreted as a TASK COMMON.

When a task is initiated on a processor, the status of the COMMON area must be passed to the processor as part of the initial task environment. In effect, it is a copy of the COMMON area from the processor that generated the new task. However, there are some important differences. Suppose the HLL requires that all variables that control parallel do loops (M, N, and K in Figure 3.4.2-2) should appear in the COMMON area of the HLL program. Then, at run time, when a processor is assigned a portion of a parallel do-loop, the values of M and N in COMMON would reflect only the part to be done by that processor. Thus one processor might receive a TASK COMMON with M=1; N=3, while another processor participating in the execution of the parallel do-loop would receive M=4; N=8 etc.

Similarly, if the above processors encounter concurrency within their assigned tasks, the COMMON area on each of them becomes the initial environment for the new tasks on other processors. The concept of how this works must be known to the programmer. However, like many of the other things in the HLL, the associated run time operations can be made transparent by the run time environment.

3.4.6 Low Level Fortran Procedures

The high level language is used to identify coarse grained concurrency within the application. The FORTRAN subprograms that eventually do the work, are compiled on the target machine, and can be fine tuned in order to derive maximum performance by exploiting any fine grained concurrency recognizable by the compiler. The data dependencies that occur within the FORTRAN programs should be detected and optimized by the compilers provided by the target machine.

3.5 RUN TIME SUPPORT ISSUES

This section describes the issues and needs for the underlying run-time support system that is responsible for mapping the programmers view of the system (described earlier), to the hardware that is made available to the user. The discussion begins with the issues related to scheduling and synchronization, and is followed by the issues related to data management.

3.5.1 Synchronization and Scheduling

The types of communication and synchronization operations that are used on loosely coupled systems differ significantly from the corresponding operations on tightly coupled systems. In order for the environment to provide the programmer with transparency, it is necessary to develop a uniform approach to managing the scheduling and synchronization of tasks that will span all types of hardware over a period of many years.

The DDL and HLL described earlier provide the needed transparency at the source program level. The corresponding HLL and DDL compilers could generate machine specific object code based on these statements. However, that would necessitate different compilers for each machine. An alternative is to compile the HLL application program into an intermediate level (IL) code that is machine independent. The advantage of this approach is that the same compilers and IL code can be used by all types of MIMD computers. The scheduling and synchronization instructions for concurrent task execution can be specified in an architecture independent manner at this level too. The disadvantage of this approach is that the IL code needs to be interpreted at run time. This is always slower that direct execution of machine code, and it may make it impossible to achieve any speedup for fine granularity problems on some types of hardware. Nevertheless, it was selected as the method of choice, and is especially appropriate when implementing a prototype system as complex as the one envisioned herein.

At compilation, when parallel constructs are encountered in the HLL, the instructions that define the tasks must be encapsulated by instructions for the "concurrent virtual machine." These instructions identify the task control variables, the granularity of the task, and the location of the task within the total program. In essence they specify where the process can be "forked" into a concurrent task execution, and where the concurrent operation must synchronize — the point where they must "join"; i.e., where serial process resumes again.

The number of and types of processors used to execute a given application program is not specified in the HLL and may vary from job to job. As a result the run time environment must contain a dynamic task scheduling algorithm that operates from the virtual machine instructions in the HLL, the actual number of processors being used during this job, and information about the granularity of the processors. This approach is general, and can be used effectively for scheduling tasks on both homogeneous and non-homogeneous processing systems.

In summary, scheduling and synchronization of tasks can be very difficult for the programmer to manage by himself. While shared memory computers provide the appropriate synchronization mechanisms automatically with the support of a data dependency analysis and compiler directives, such facilities are not generally available on distributed memory machines. Furthermore, scheduling algorithms are generally based on the assumption that all the processors have the same performance capability. It is conceivable that networks of nonhomogeneous processors can be formed — what should be done in such cases? If the programmer is responsible for the scheduling of tasks on a nonhomogeneous distributed memory machine — he will very likely lose sight of the application itself. Hence it is necessary to provide some mechanism of automated task scheduling that can be applied to a variety of hardware configurations.

3.5.2 Data and Memory Management

This research work is aimed at problems that may require data spaces between 40 and 100 GigaBytes(GB). Experience with modern computer systems has shown that the hardware/software virtual memory component facilitates the development of application programs considerably. The environments of the 1960's-1970's demonstrated that the combination of virtual memory, virtual data bases, and a comprehensive data structuring facility made possible the development of some of the most comprehensive and most used engineering systems in the world. In these environments the responsibility for data and memory management is isolated from the programmer; they generally provide reasonable overall levels of performance [23]. How does one map those concepts to MIMD systems?

Figure 3.4.4-1 demonstrates how objects are referenced symbolically in the HLL. Objects that are leaves in tree data structures are referenced via additional FORTRAN-like subscripts. An example might be **A(myproblem, node_id, 1, 1).** This implies that at run time the object will appear in the memory of the processor that makes the request. Objects, and any necessary intermediate level pointer objects, are created automatically and dynamically at run time, and are assigned to the logical data space. That space is then mapped on demand to the memory of the processors by the memory management system. The DBMS is responsible for a) tracing out the tree structure to find the object, b) creating both intermediate pointer objects and the final data object, and c) assigning appropriate data attributes to maximize the possibility for concurrency. The user specifies the attributes for only the actual data object. All other intermediate objects are handled automatically by the DBMS.

We have very little knowledge and experience about the behavior of combined MIMD/ DBMS systems. Consequently, there is no basis for changing the functional design of this part of the environment from the one used in an earlier environment. As a result we opted for utilizing the existing array structured DBMS from the POLO system [53].

The memory management system is responsible for mapping the virtual object space to a specific processor's address space for the DBMS. The memory management system makes it appear to the DBMS that any object (whether it is in the processor's memory, another processor's memory, or on a disk somewhere) is in the current processor's memory. Thus, it must deal with the various memory topologies of modern MIMD hardware. The important issues related to this aspect of the problem include; memory hierarchies, data coherence, virtual memory, and parallel i/o, and are discussed in the next sub-sections.

The data/memory management system on a parallel system for engineering problems,must deal with large and complex data spaces that are mapped onto many memories and which require extremely high throughput. This problem is unmanageable for all but a very few engineering programmers; it is therefore desirable for this process to be totally automated from the application programmer's point of view.

3.5.3 Memory Configurations

Examples of various memory configurations were shown in figure 1.2-1. Shared memory and distributed memory systems might be considered "classical" systems. They represent the two extremes relative to types of data access. In shared memory systems the software on any processor can access the entire virtual memory space. On distributed memory machines the processors can only directly access their own memory. All other transactions are done by message passing.

The hardware architecture of hybrid memory computers is the most complex. Examples of such configurations include the ETA-10 [27] and Cedar [12] supercomputers. In addition to having a global shared memory, the ETA-10 has local memory for each processor. The Cedar processors have access to both a shared cluster memory as well as a shared global memory (see figure 1.2-1). This property gives them the appearance of being a message-passing or loosely-coupled system. The Cedar machine can be programmed as a tightly-coupled machine, and in certain circumstances the cluster memory may act as a cache for the global memory. A similar environment can be created on the ETA. Although the ETA no longer exists, its configuration is still valid and should be considered. Still other configurations may be proposed in the future.

Regardless of the memory architecture of an MIMD system, the software environment's memory management system on each processor must make the total memory of the system appear as one contiguous virtual address space to the DBMS on the same processor. The DBMS must manage the memory on the processor so that there will always be room to satisfy requests for objects; it must obtain objects from other processor memories and from the system disk environment. The disk system is necessary because the assumed aggregate virtual space of 100 GB will always be larger than the available primary memory.

In moving from single processor environments to multiprocessor environments, mapping the object space onto the real memory becomes an order of magnitude more complex. In some cases, multiple copies of a data object can exist in the system. The difficulty with this is insuring that all the copies that are generated in different processor memories are identical. It becomes a serious problem when multiple copies of a given object exist and one processor has to modify the object. This is known as the data coherence problem.

3.5.4 Data Coherence

Data coherence implies that a data object is both valid and consistent. This means that on distributed memory systems all copies must be kept consistent. Maintaining data coherence is also a major problem on shared memory systems. Even though only one copy of an object exists in memory, many processors may be referencing the object simultaneously. One must insure that the object is not modified by one processor while another processor is still using the object. In shared memory systems, all processors can have immediate access to the information describing the state of a given object. Access to a data object can be monitored and coherence can be enforced relatively easily.

On distributed memory machines messages are passed among the processors to control access to objects. The latency in the message passing network implies that the exact status of the object on a given processor is not known precisely. For example, by the time the message is received at a processor requesting access to a given object, that processor might not have the object any more.

On machines with multiple memories, maintaining coherence requires that extra copies of a data object must be invalidated or updated whenever a copy of the object is modified. This implies that other processors that are using the object must give up their copies of the object or invalidate them. The environment must contain mechanisms for keeping track of those processors that have a valid copy. Other considerations include: how to get an object?, where to get it from?, and when to get it?

In order to achieve the appearance of a shared virtual memory for data objects, it is necessary to have a software based local and global memory manager for each type of memory in the system. Each processor must be capable of executing the functions of the memory managers for those types of memories to which it has access. Objects may move between physical memories. Consequently, each memory manager must not only keep track of the data objects currently being stored in its memory, but must also be responsible for managing the global data space by maintaining parts of tables that control access and ownership of data objects in other memories. Through internal message passing (not necessarily hardware) these managers must keep each other informed of the location and status of data objects, and must insure that data which is needed by the DBMS on a particular processor is made available automatically to the requesting processor -- when the access rights are proper.

3.5.5 Parallel I/O

The tools to manipulate and maintain the data space both during a job and, after a job has been completed, must also be addressed in light of the rapidly changing disk technology. It is very important that the software environment be able to adapt to new disk management features as they become available.

An effective virtual memory system requires high speed I/O to support it. Some modern MIMD architectures have very sophisticated I/O subsystems that automatically convert program I/O requests to multiple I/O requests on different disks i.e. striping. The architecture of some MIMD systems appear to have considered the implementation of I/O as an afterthought; i.e., nothing beyond what might appear on a serial computer. For example, ETA has a very fast pipe that leads to an entire disk system while on the FLEX , access to disk is restricted to certain processors. The only way to manage this kind of uncertainty is to provide for it in the MIMD environment.

A single software disk manager in the environment could be used to communicate between the memory managers and the real disk system. However, that would be a severe bottleneck. Consequently, the memory managers on each processor must contain a subset of a disk management system that decides how to map objects to the real system disks. Each real disk must be associated a software component called a "disk manager" that controls access to that disk and its corresponding page allocation tables. Configuration tables can be used to map this aspect of the environment onto a specific architecture. Thus, if the machine has a sophisticated disk system, the configuration tables would reflect it, and hence the environment would utilize what is provided by the manufacturer. If the machine does not have a sophisticated disk system, the environment may be able to make up for it.

For example, on systems without sophisticated disk systems one can break large data objects up into manageable blocks called pages. Pages can be distributed over many physical disks. Messages can be sent "simultaneously" to retrieve pages from disks in order to re-form a large object. If the message passing network is very fast compared to the speed of the disks, one can achieve significant time savings via this parallel I/O concept.

3.6 SUMMARY OF NEEDS

The following is a summary of the components that must be incorporated within the programming environment:

- A Data Definition Language that permits the programmer to develop hierarchical data structures of various granularities.
- A High Level Programming Language (procedural - functional type) that supports at least two types of parallel constructs as shown described in section 3.4.2. It must be machine independent and flexible. It should allow the programmer to specify tasks independently of the hardware provided. The programmer must be able to quantify the relative size of the task to improve the run time scheduling and synchronization of tasks.
- Compilers to generate IL code. Two levels of IL code are needed. One level describes the problem, the other to describe the data structures. They need to generate instructions that permit dynamic run time scheduling and synchronization of tasks.
- An IL code interpreter / loader -- to load the appropriate segment of IL code and interpret the IL code instructions generated by the HLL compiler
- A Control Program Module that will coordinate the synchronization and scheduling of tasks dynamically. It should be flexible enough to adopt several scheduling strategies.
- An Interface Component between User and the System that is responsible for processing user commands and initializing the system.
- A Data Management Module that will be able to resolve what data objects the memory manager should get.
- A Memory Management Module that can map a virtual address space to the various memory configurations. It should provide dynamic memory management and data coherence at the object level.
- A Disk Management Module that will provide an interface to secondary storage and support parallel I/O and disk striping.

CHAPTER 4

OVERVIEW OF THE CONCEPTUAL DESIGN

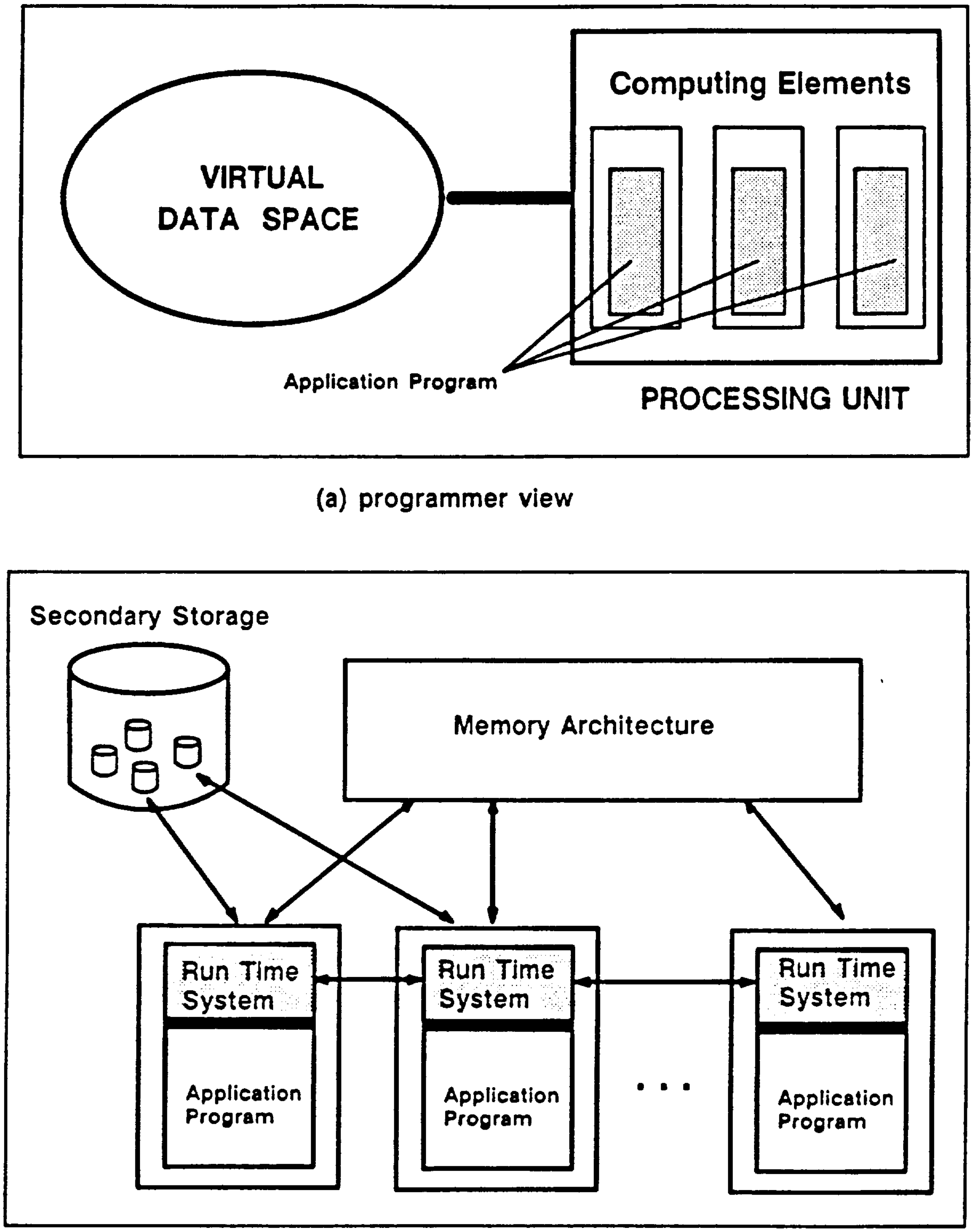
The previous chapter identified several features that are needed in the programming environment. This chapter illustrates the conceptual design of an environment that satisfies the needs outlined earlier. Only a brief overview of the functional components and their interrelationships is given here. Detailed discussions of the individual components are then given in subsequent chapters.

4.1 SYSTEM CONCEPT

The view of todays MIMD virtual machine as seen by the programmer is illustrated in figure 4.1-1a. The virtual machine consists of two parts; a processing unit and a virtual data space. The processing unit consists of several computing elements that execute various parts of an identical copy of the application program. The virtual data space contains data objects that are:

- *persistent* - objects referenced in the application will exist after the application is terminated unless they are explicitly deleted by the programmer,
- *distributed* - multiple copies of the object may be distributed within the virtual data space,
- *coherent* - objects referenced by multiple processors will be consistent, and only one processor at a time is allowed to modify a given data object,
- and *virtual* - the objects are virtual in the sense that they will appear on demand in the memory of the processor that needs the object. The objects do not always have to reside in primary memory.

A virtual machine is just that - virtual. Figure 4.1-1b illustrates the view of the run time system in reality. Each processor in the processing unit executes a program consisting of the run time environment combined with the application program. The heavy line between



(a) programmer view

(b) reality

Figure 4.1-1 Virtual Machine Concept

them represents the interface between the HLL's view of the world as shown in figure 4.1-1a and the real world that has distinct memory architectures, interconnection networks, and disk systems. What is the run time system?

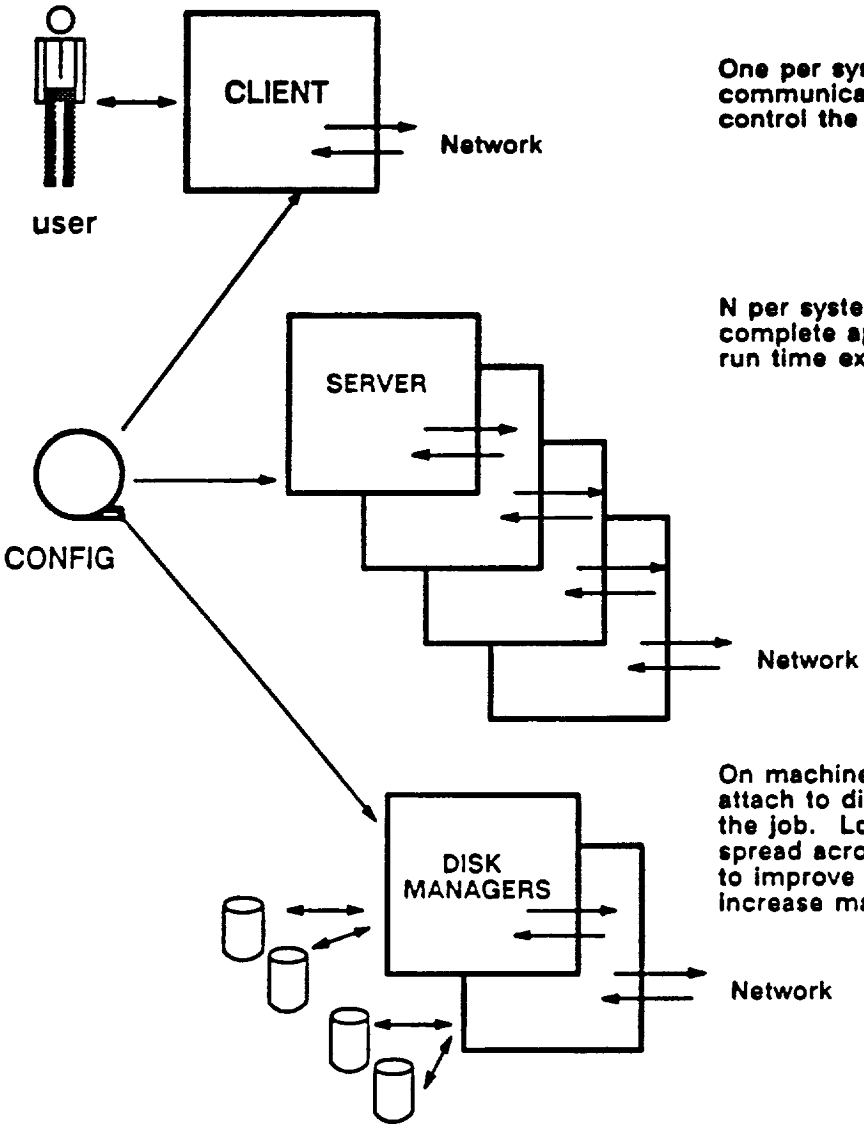
4.2 SYSTEM PROGRAMS AND OPERATION

There are three distinct types of program that are used to support the view provided by the programming environment. They are called the Client, Server and Disk Manager programs herein. The organization of these programs is shown in figure 4.2-1. Each program runs as one process on a host processor. In a typical working environment, there is only one copy of the Client program, there can be multiple copies the Server, and multiple copies of the Disk Manager programs. A configuration file is used to specify how each type of program is mapped to the various processors in an MIMD configuration. Typically, the Client can run on the same processor as a Server or Disk Manager. There is very little to be gained from mapping more than one Server to the same processor since that is the program that contains the application program; they would compete heavily for the same processing resource.

The Client program controls the operation of the system. It is responsible for initializing the system, and for scheduling and synchronizing the execution of tasks when the application program is running. The Client is also an interface between the *user* and the other components in the run time system.

The Server program contains most of the run time environment plus all of the application program. It will execute tasks assigned to it by the Client. All of the Server programs are identical. However, the processors that execute the Server programs may differ in performance. Hence, in non-homogeneous computer systems, the Servers are considered to have different processing capabilities. This is an important factor that must be considered in the task scheduling process.

The Disk Managers are programs that interface between the Servers and the disk hardware. The Disk Managers will service requests from the Servers to open, close, delete and copy data bases. The Disk Managers will also write data objects to disk files and read data objects from the files.



One per system to communicate with user and control the operation of the system

N per system. Each has complete application and run time executive.

On machines that can actually attach to disk files used in the job. Logical data files are spread across multiple disks to improve performance and increase maximum logical file size.

Figure 4.2-1 Functional Components Of The Environment

A logical data base may be subdivided into many physical sub-files that are distributed among the disks on a system. In such cases, one or more Disk Managers are then required.

All the programs communications with each other via a **reliable** communication network.

Figure 4.2-2 schematically illustrates the interaction between the three types of programs during the execution of an application. In the following text, the notes in parentheses refer to the location of the annotated text in the figure.

When a Server is idle it reports in to the Client (middle left). Initially all Servers are idle.The figure shows that the Client is responsible for assigning units of work to the Server programs(top right). Units of work consist of one or more tasks; the size of the unit of work is determined by the Client when it examines the relative granularity of the task and of the processor to which it is being assigned. The first unit of work is to tell one Server to go into executive mode and process instructions from the user(upper left). The user responds with

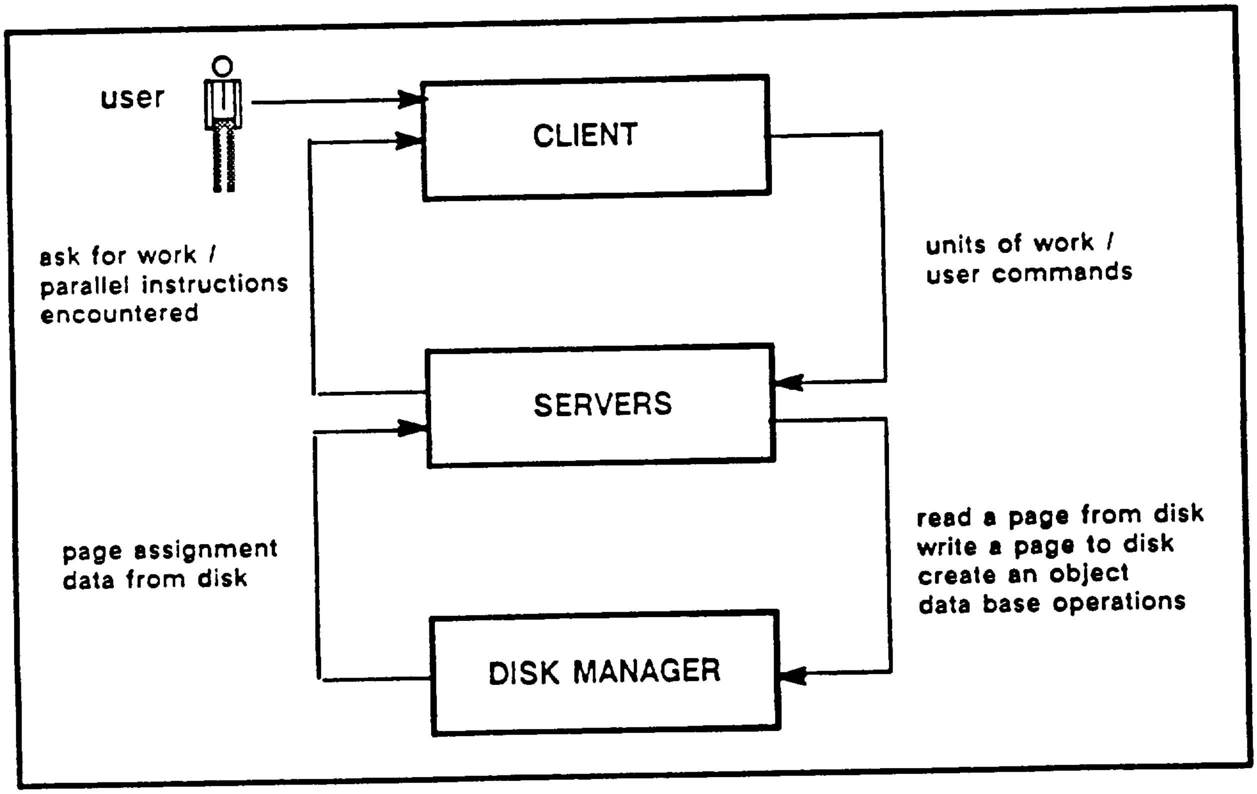


Figure 4.2-2 Client - Server - Disk Manager Interaction

a command to execute an application program. The Client will forward the commands and data from the user to the Server. Now one Server is executing the application.

When parallel constructs are encountered within the application program, the Server will send the pertinent information to the Client(middle left) and wait for the Client to assign it some work. After assigning a unit of work to the Server, the Client will check if more work is available, if so it will hold it until it can locate an idle Server. Conversely, when a Server finishes executing some work, it will ask the Client for more work(middle left). Thus, the Client keeps marrying idle Servers to tasks until the job is completed.

The Server will communicate with the Disk Manager when data should be written to disk or read from disk(lower right). User commands to manipulate data bases (open, close, copy...) are sent to the Disk Manager manager via the Server. System type operations that manipulate data bases originate from the Server and are send to the Disk Manager. When data objects are created or deleted, the Disk Manager is responsible for allocating and de-allocating the records in the physical disk file(lower left).

The details of the Server, Disk Manager and Client programs are described in chapters five, six and seven, respectively.

CHAPTER 5

FUNCTIONAL DESIGN OF THE SERVER PROGRAM

This chapter describes the functional design of the Server program. The Server is the heart of the environment and consists of two parts — the run-time executive modules and the application subprograms. Together, these two components are referred to as the run-time system. The following sections describe how the run-time system is generated, its major components and, the interaction of the components.

5.1 GENERATING THE RUN-TIME SYSTEM

The run time system contains the application; thus it must be bound with the latter each time a new application is created, or each time a change is made. The procedure to generate the run time system is illustrated in figure 5.1-1. The application programmer supplies the Data Definition, the High Level Language program, and the procedural subprograms for the application. These three components are shown in the shaded box at the top left hand side of the figure. The resulting run time system is shown in the shaded box on the right side of the figure. It is produced by combining the IL code segments generated from the DDL and HLL compilers, and binding the application object code with the run time environment's object code. The following subsections describe the components shown in figure 5.1-1 in more detail.

5.1.1 The Data Definition Compiler

The Data Definition Compiler generates an internal form of the data objects defined using the data definition language DDL. The internal form contains the information necessary for the HLL compiler to generate the instructions that access data objects during execution. The reason for keeping the DDL separate from the HLL is that a number of HLL programs can be compiled assuming the same data space defined in a common DDL. Thus it provides a mechanism to integrate large systems from a series of smaller HLL programs; each program can both access and utilize the data space created by another program that employs the same DDL -- in this respect the data space takes on the persistence attribute of a data base.

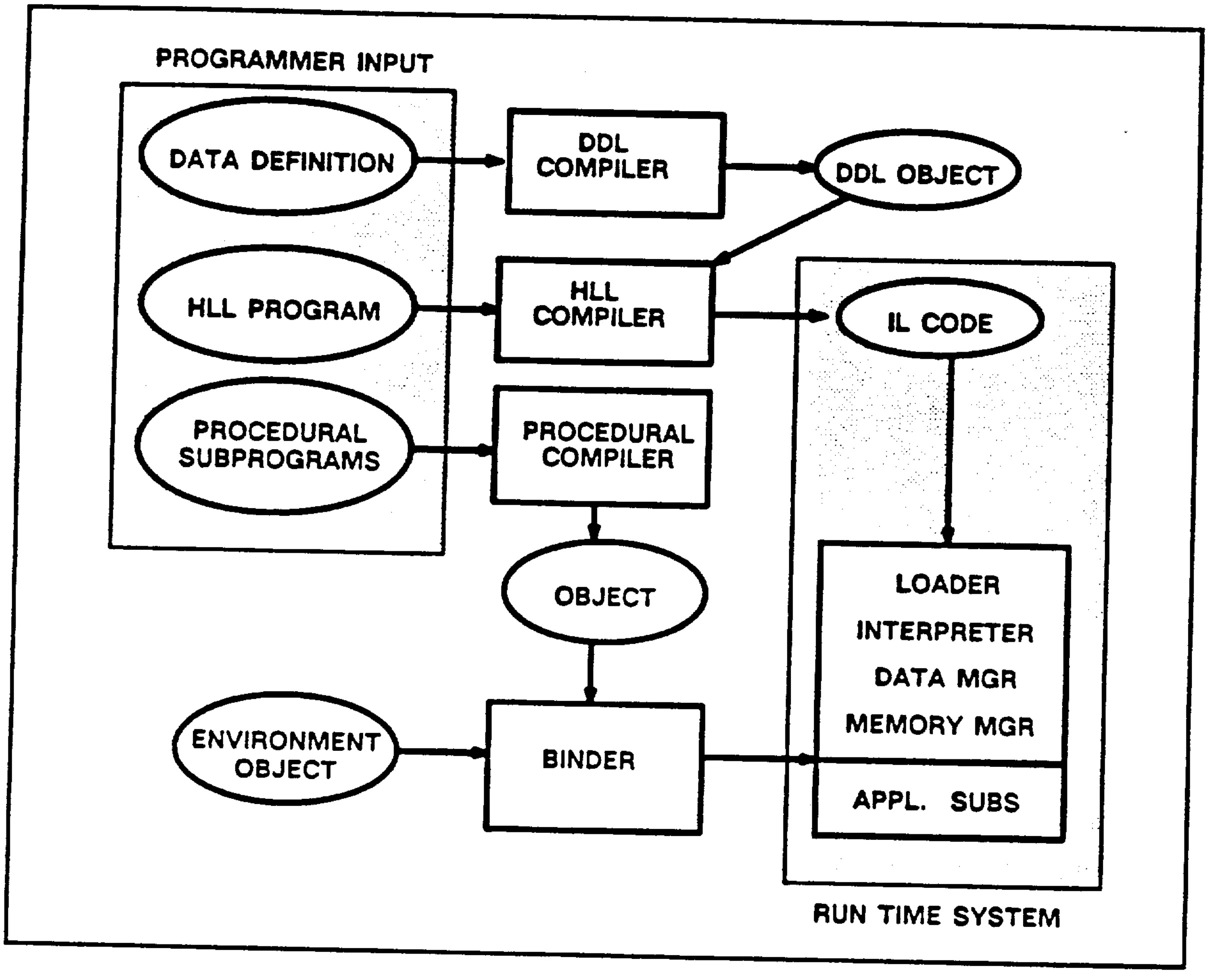


Figure 5.1-1 Developing the Run-Time Executive

5.1.2 The High Level Language Compiler

The HLL compiler generates Intermediate Level (IL) code. The IL code consists of an array of instructions that are interpreted by the IL code interpreter in the run-time system. The instructions are designed to get data objects and to call the appropriate FORTRAN subroutine to perform the action corresponding to the operation specified in the HLL program. References to data objects are resolved into a low level, address free form, for interpretation by the data manager during execution. In DBMS terminology this is called early binding, and results in a system of higher performance than the more general, and more flexible, late binding systems.

Before going on it might be useful to understand the general format of the IL code for a given set of HLL instructions. This is shown in figure 5.1.2-1, which illustrates the concept of how a nested parallel do loop is transformed into IL code. The example describes how the elements of two matrices are added together concurrently. Each element within the matrices A, B and C is a 50 by 50 sub-matrix, and is defined as an individually addressable object. The figure does not show the imtermediate levels of pointers maintained by the data base management system. The outer parallel do-loop sets the variable i, while the inner parallel do-loop sets the variable j. The relative granularity of the outer loop is ten times larger than each iteration of the inner loop. The high level operation 'ADD_MAT' will add the elements of matrices A and B, and store the result in C. The actual calculations are performed in a FORTRAN subroutine that is not shown in the figure.

The form of the IL code instructions generated by the HLL compiler are shown in diagram below the HLL code. The IL code contains information about the concurrent tasks, task granularity, and task synchronizations points. It also contains instructions that are used to increment the do-loop counters and fetch the data objects.

5.1.3 The Procedural Compiler

The application subroutines that support operations used in the HLL code are compiled by the procedural compiler of the target machine. The application can be fine tuned to the hardware by invoking the appropriate optimizations provided by the compiler. The object code for the application is bound together with the object code that supports the system modules.

5.2 RUN TIME SYSTEM COMPONENTS

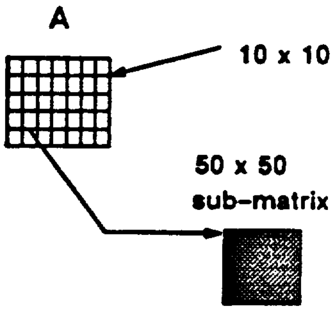
The run time system was derived iteratively. One way of justifying the concept is to show how it works via a "run through" of various situations that it might be expected expected to handle.

The major components of the run time system includes: a Data Pool; an IL Code Loader; an IL Code Interpreter; a Data Manager; a Memory Manager; and, the Application Subroutines. The following sections describe the functions of each component. The discussion is based on figure 5.2.1-1. It is an overview of how the components interact with each other, and how they interact with the Client and Disk Manager.

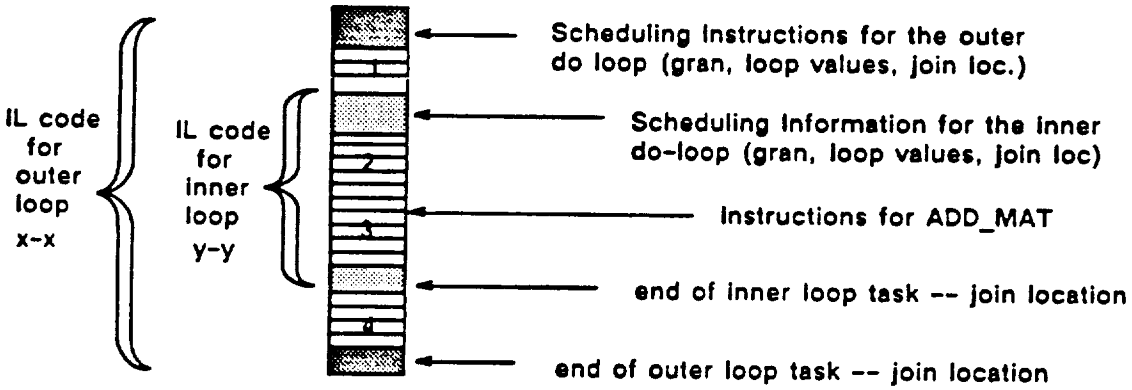
```
    DATA DEFINITION:    A GROUP 10 10 50 50
                        B GROUP 10 10 50 50
                        C GROUP 10 10 50 50

    TASK COMMON I, J

x:  DO FOR I = 1, 10 IN PARALLEL GRAN 10
y:      DO FOR J = 1, 10 IN PARALLEL GRAN 1
            ADD_MAT(A(I,J,1,1), B(I,J,1,1), C(I,J,1,1))
y:      ENDDO
x:  ENDDO
```



(a) High Level Code to Perform Matrix Addition and Sample Data Structure


IL code
for
outer
loop
x-x
IL code
for
inner
loop
y-y
1
2
3
4
Scheduling instructions for the outer do loop (gran, loop values, join loc.)
Scheduling information for the inner do-loop (gran, loop values, join loc)
Instructions for ADD_MAT
end of inner loop task -- join location
end of outer loop task -- join location

1. control logic for loop iteration check for loop index I

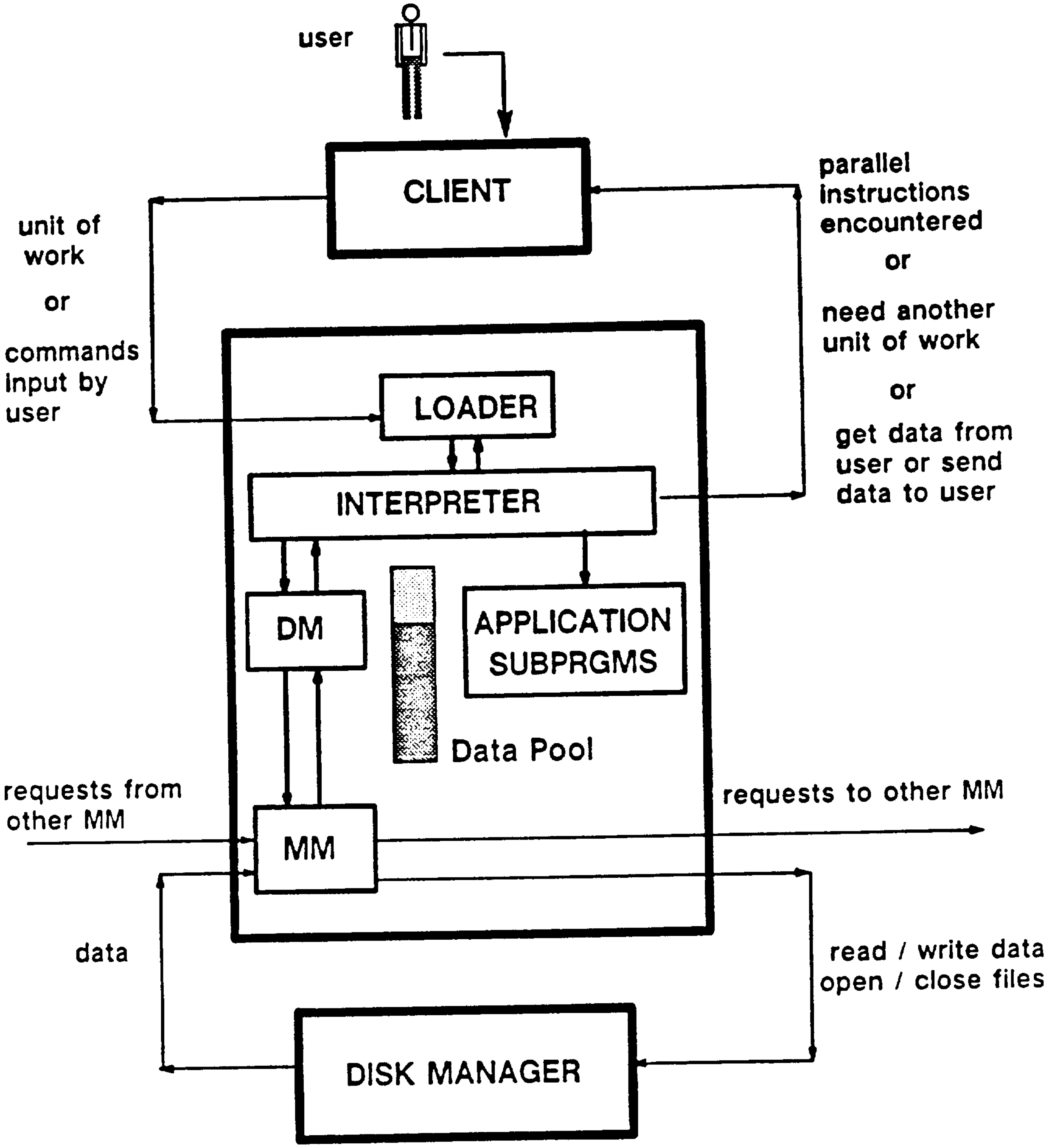
2. control logic for loop iteration J, instructions to get data objects and call the routine to perform matrix addition

3. instructions to increment loop variable J

4. instructions to increment loop variable I

(b) IL Code format corresponding to HLL code above

Figure 5.1.2-1 HLL - IL Code Transformation



Key: MM - memory manager
DM - data manager

Figure 5.2.1-1 Interaction Of Server With Other Components

The **Data Pool** is a buffer area in which any object referenced by a given module is temporarily stored. It is divided into two areas: one area is used solely by the system, the other is used for the application. The system portion of pool contains the IL code instructions and TASK COMMON areas. The size of the system area depends on the size of the common area and the size of the IL code segment. When parallel tasks are encountered, the system area is saved as the initial environment object for all tasks that will be descendents of the current task. This object is initialized in the data pool on each processor before task execution begins.

The IL code is in essence a series of loosely coupled programs. Each program is segmented into subprograms. Since the total application may be quite large, it is necessary to have a procedure that can fetch segments of IL code into the processor's memory (data pool) on an as-needed basis. When the Server receives a unit of work to execute, the **Loader** will simply load the environment object that contains the appropriate segment of IL code. Since the IL code segment is stored as a data object, it is easily obtained from the virtual space just like any other data object.

The **Interpreter** translates the IL code that was generated by the HLL compiler. There are several types of instructions. An example of the IL code that is generated from some HLL code was described earlier and shown in figure 5.1.2-1. When the interpreter encounters the synchronization and scheduling instructions in the IL code, the Server will send the task information and the environment object (task common) to the Client, and wait for a response in the form of a unit of work.

Prior to invoking a subprogram that corresponds to an HLL operation, the interpreter will translate the DBMS instructions that were generated to "fetch" the operands (objects) of that HLL operation. The **Data Manager (DM)** is responsible for executing the DBMS instructions. The data management system operates at a logical table level. It knows about how objects are defined and how they can be identified. The DM is responsible for associating an HLL reference to an object such as A(i,j,1,1) with an object identifier. An object identifier, henceforth *pointer*, is unique for every object within the data space. The pointer specifies the physical location of the object within the data base. Details of how the pointer is made up can be found in section 8.3.2.

When the DM has identified the pointer that corresponds to an object that must be brought into the Data Pool, it will invoke the **Memory Manager (MM)**. The MM will get the object into the Data Pool and return the address to the DM.

For hierarchic data structures, the DM will automatically traverse a tree to get to the required object. Each time the DM has to get an intermediate level object that contains the pointer to the next object, the MM is invoked to get the data object into the data pool.

The functionality of the DM and MM modules is very important. By defining them as separate modules, one can replace the MM module without modifying the DM module. The interface between the DM and MM is not complicated. New MM modules that support differing MIMD memory configurations can easily be inserted. All of this activity is transparent to the application program. Hence, they can be ported to MIMD machines with differing memory architectures quite easily.

The Memory Manager is responsible for managing the pool area. The basic functions of a memory manager are to get and/or create objects in the data pool when they are requested by the Data Manager and, to remove and/or delete objects from the data pool when they are no longer needed by the application subprogram. In order to accomplish this, a virtual memory approach is used. The memory manager gets its instructions from the Data Manager. In order to perform its operations, the Memory Manager may communicate with the disk system or other Memory Managers.

When an object must be created, the MM requests the Disk Manager (DM) to allocate physical records from the data base. Using the returned information from the DM, the MM will build in the data pool both an encapsulated data object, and a unique address free pointer to the object in the data base. It then returns both the pointer and the location of where the new object exists within pool to the Data Manager.

When the DM requests the MM to delete an object, the physical records that were used by the object are "returned" to the file (data base) and the space occupied by the object in the pool is "de-allocated."

When an object is requested by the DM, the MM will check if the object is already in the pool area. If the object is in the Data Pool, the MM will check if the access requested -- *read* or *write* — is consistent with the current state of the object. If the access check is consistent with the current use, the MM will return the location of object within pool to the DM.

If the object is not in pool or, if the access requested is in conflict with it's current state, the MM will get the object into pool with the appropriate access. The memory management system is responsible for insuring that the object requested by the DM is coherent.

This is a key feature — it simplifies the programming process considerably. The programmer simply references the object in the HLL operation and it is made available to the FORTRAN subprogram. The programmer does not have to develop any code to determine where the object is, or how to get it.

5.3 DETAILED DESIGN OF THE SERVER

This section provides a detailed discussion of the operations within the Server program. The discussion is based on figure 5.3-1. This figure illustrates how the components interact with each other in more detail than the illustration of figure 5.2.1-1. It shows some auxiliary components that are required to support the interaction between the major run-time components, and the interaction between the Server's components and the Client, the Disk Managers, and other Server programs.

The auxiliary components are: the Send Task; the Receiver Task; the Token Scanner; and, the queues and buffer areas. The Send Task, shown at the bottom of the figure, is responsible for sending messages from the components in the Server program to any other process on the network. The Receiver Task, shown at the bottom of the figure (identified as RECV'R TASK), is responsible for receiving messages from other processes and forwarding them to the appropriate component within the Server. The Token Scanner tokenizes commands and data entered by the user. The language processing part of the interpreter parses them based on HLL descriptions (not described herein.) The queues are buffer areas that provide a temporary location to store messages.

The data flow lines are tagged with character identifiers. The control/data flow lines are tagged with numerical identifiers. These characters and numbers will be used in the discussion below; a description of an action that corresponds to a change in control or a message transmission in the figure will be accompanied by the tag in parenthesis.

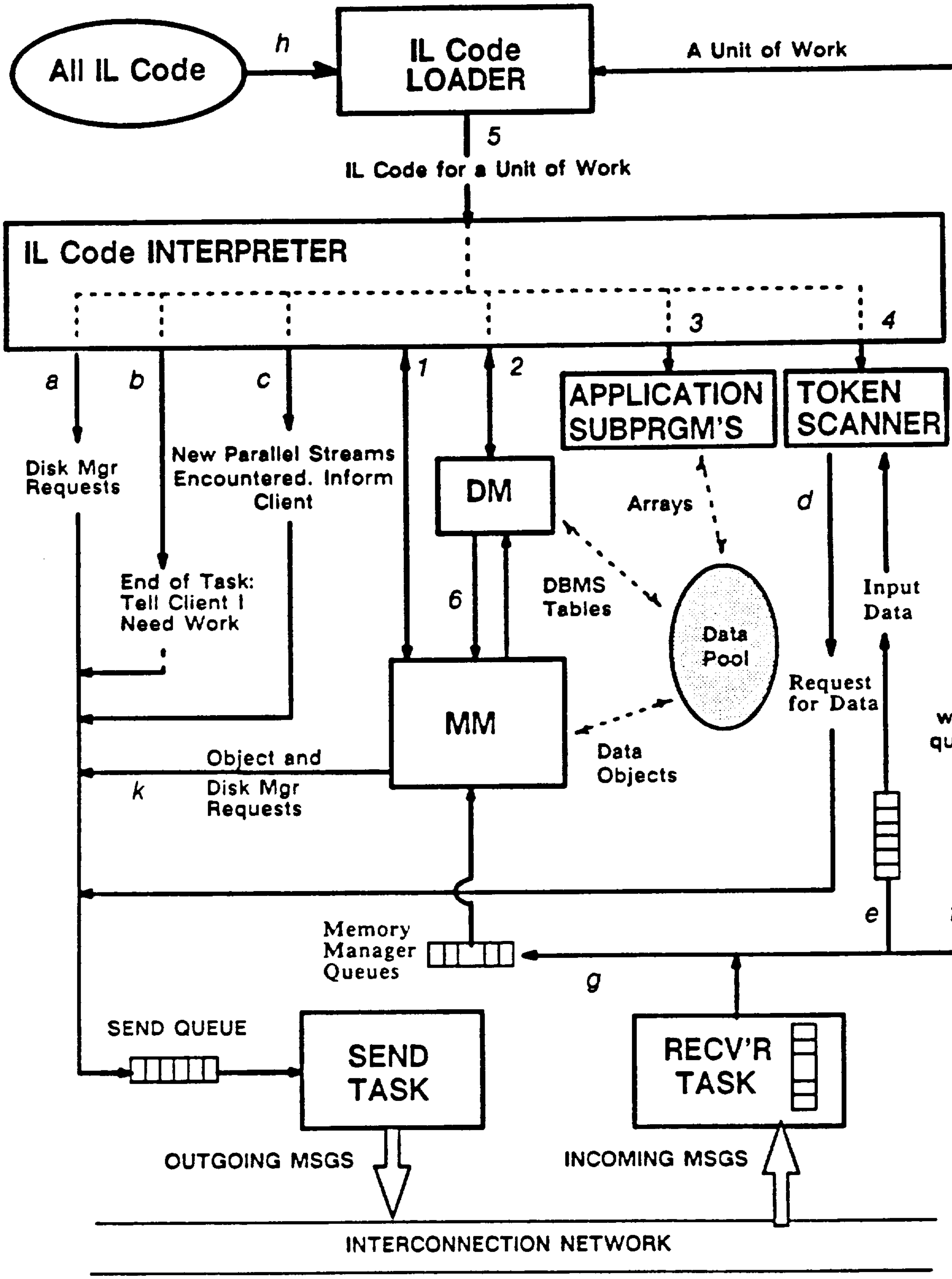


Figure 5.3-1 Functional Details of the Server Program

All outgoing messages (a,b,c,d,k) are placed in the Send Queue. The Send Task will process the messages in the order that they were put into the Queue. The Interpreter, Token Scanner, and the Memory Manager (MM), are the components that share the Send Queue.

All incoming messages are placed in the buffer area of the Receiver Task. The Receiver task will forward the messages to the Memory Manager, the Loader, or the Token Scanner. Messages (g) sent to the Memory Manager are put in the Memory Manager Queue. Messages (f) to the Loader are put in the Work Queue. Messages (e) sent to the Token Scanner are put in the Input Data Queue.

Reference to the matrix addition example illustrated in figure 5.1.2-1 will be made in order to explain how the Server program operates and interacts with the Client and Disk Manager. The remainder of this section describes how the Server executes a program like that shown in figure 5.1.2-1. The discussion is based on the configuration in figure 5.3-1.

Whenever the user invokes the system, the Client will initialize the Server process. The first task that the Server must execute is to initialize the major system components. When this is done, the Server will ask the Client for some work. The Server will wait for the Client to respond with something to do. The first Server to check in with the Client will be told by the Client to process commands from the user. This Server will then invoke the token scanner to obtain data. The Scanner requests data from the Client who then prompts the user. Eventually, commands and data entered by the user make their way to the Token Scanner which processes the information (e). The "run" command is a unique operation in that the Scanner will pass control over to the Loader (not shown). The Loader will get the appropriate segment of IL code (h) into the Data Pool and pass control (5) to the Interpreter.

During the execution of an application program, the Interpreter may encounter instructions that identify concurrency within the application. Referring to the sample code described in figure 5.1.2-1, the Interpreter will encounter the first set of synchronization and scheduling instructions in the IL code for the outer loop. When these instructions are encountered, the interpreter will send the task common and pertinent information (c) to the Client. The Server will wait for the Client to respond with some work to do. The information about the unit of work is sent to the work queue (f), at which point control reverts to the Loader. The loader will make sure that the correct segment of IL code is made available to the Interpreter.

Control (5) will be passed back to Interpreter again and it continues translating the instructions.

In the matrix addition example, the unit of work will consist of one or more iterations of the do-loop. The number of iterations that make up a unit of work depends on the granularity of the task and the granularity of the processor. Further details on task scheduling are described in section 8.4.4.

The operations described above are repeated when the second set of scheduling and synchronizations instructions are encountered for the inner parallel do-loop. When the Interpreter resumes again it will encounter the instructions that were generated to get the data objects for the High Level operation "ADD_MAT." During execution of this inner loop, the operations on the outer loop are suspended. When all inner loop calculations are completed, the Client notifies the original Server that it may proceed with the next operation in the outer loop.

The objects A(i,j,1,1), B(i,j,1,1), and C(i,j,1,1) referenced in the HLL operation "ADD_MAT" must be brought into the data pool before the FORTRAN subroutine that corresponds to "ADD_MAT" can be invoked. The address of the operands are the locations of the where the objects are within the data pool. The "fetch" operation is essentially a call to the data management system. It is responsible for returning the address of the object to IL code interpreter.

When the interpreter encounters data base management instructions, control (2) is passed to Data Manager (DM) module. The DM operates on Data Base Management System (DBMS) tables that are located in the Data Pool. The DM will pass control (6) to the Memory Manager (MM) when an object is needed. When the data object requested by the DM is not in the data pool or a change of access is necessary, the MM will a send message (k) to other MM's or Disk Managers to get the object. The MM will wait for a response (g) before passing control (6) back to the DM.

Requests (g) from other MM's for data objects used by this Server are processed by the MM component. Similarly, requests for a given object used by other Servers are eventually sent to the MM component of the other Server. The MM's are responsible for maintaining coherence of the data objects.

Data coherence implies that no more than one MM can obtain write-access to a given object at any time. Furthermore, only one MM is allowed to create a specific object. Multiple copies of an object can exist in a state of read-access only. When a MM requires an object for write-access, all other MM's must relinquish their use of the object first. Data coherence is maintained by appropriate communications with other MM's. In distributed memory machines, this is accomplished by message passing. In shared memory machines communication may be (is) accomplished by using shared variables.

After all of the data objects for a given HLL operation are accessible in the data pool, the interpreter will pass the addresses of the objects to the appropriate FORTRAN subroutine in the application. This occurs when the interpreter encounters an instruction to invoke an application subprogram. Control (3) is passed to the module that executes the application program.

After the Interpreter has finished translating all the instructions for the unit of work assigned to it by the Client, it will send a message (b) to the Client, asking for more work. A response (f) is expected in the work queue. The message sent back to the Server could be a unit of work consisting of some more iterations of the inner parallel do loop. If all the iterations of the inner loop have been processed, and the Server suspended a task at the outer loop level, the Client will respond with a message to 'join' or resume operations on the outer loop. In this way, the Client controls the synchronization of parallel tasks among the Servers.

If the Client doesn't have additional work for the Servers, the Servers will wait until something becomes available, i.e. another parallel stream is encountered in the application program.

As the Interpreter translates the IL code for various parts of the application program, it will encounter different types of instructions. For example, an instruction to get data from the user means that control (4) is passed to the Token Scanner who will issue a "request for data" message (d). The Token Scanner will wait for the user to respond with some data (e).

When the interpreter encounters system instructions to get data objects, the interpreter will pass control (1) to the MM. The MM will send a message (k) to get the data object. The requests may be sent to other MM's or the Disk Manager.

System Instructions that require the manipulation of the data bases are transformed into requests (a) that are sent directly to the disk manager.

5.4 Considerations for Implementation

There are two major concerns that must be addressed when implementing the design of the Server program. The first concern is related to the control aspect of the modules. The second concern is related to the use of the queues.

The Server was described as if the components were operating serially. That is not the case. It is necessary for some of the components to operate concurrently and independently of each other. These implications are not apparent from just looking at figure 5.3-1. For instance, while the Server is waiting for the Client to respond with a unit of work, it may be necessary for the Memory Manager to respond to messages from other Servers. This implies that the Send and Receiver Tasks must also be operational while the Server is waiting for work.

The group of modules that consists of the IL Code Loader, Interpreter, Application Subprograms, Token Scanner, and the Data Manager, operate in a serial mechanism. For instance, the Interpreter cannot translate instructions until the Loader makes them available. Since the DM and Application Subroutines are invoked by the Interpreter they too become part of the group. Hence, this group must be able to work independently of the MM, Send Task, and Receiver Task. At the same time it would be ludicrously slow for the DM to pass messages to its own memory manger. Therefore the MM module is split into three components internally. One component is part of the task containing the DM. The others are independent and described later. **For implementation purposes, the Server process must be multitasked.** Further details are given in chapter eight.

The second design concern is related to the use of queues. Some of the queues shown in figure 5.3-1 are shared (used) by two ore more modules. Therefore, operations on these queues must be synchronized to avoid deadlocks and data coherence problems. These are described in chapter eight.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 6

CONCEPTUAL DESIGN OF THE DISK MANAGER PROGRAM

6.1 INTRODUCTION

This chapter describes how the functions of the Disk Manager lead to the conceptual design of the program. The disk management system is responsible for handling requests from the memory managers of the SERVER programs. It isolates the SERVER program from the intricacies and deficiencies of the manufacturer supplied disk system.

The functions of the Disk Manager include opening and closing the physical disk files, writing data to disk files and reading data from disk files, and managing the use of records for all disk files assigned to it.

6.2 DISK MANAGER CONFIGURATION

The Disk Manager (DM) configuration is illustrated in figure 6.2-1. It is decomposed into four modules. The Send and Receiver tasks are functionally identical to the components of the same name in the Server program (described in the previous chapter). The Disk I/O Interface task is the main task of the Disk Manager. The Page Allocation task is a small task used to improve performance. It will be discussed first.

The purpose of the Page Allocation module is to rapidly service requests for the allocation of records in a particular disk file. When the Memory Manager creates or deletes objects, messages will be sent to the Disk Manager. The Receiver Task will forward the message into the Page Allocation Queue. The information in the message includes the type of operation (create or delete), and the file number.

When an object is created, the Disk Manager will respond by allocating some records from the appropriate disk file for the object. The record numbers are sent back to the Memory Manager who will encapsulate the information in the documentation of the object. When an object is deleted, the Memory Manger will send back the records numbers and their corresponding file number. The Disk Manager will update the File Table by updating the Free List of Pages.

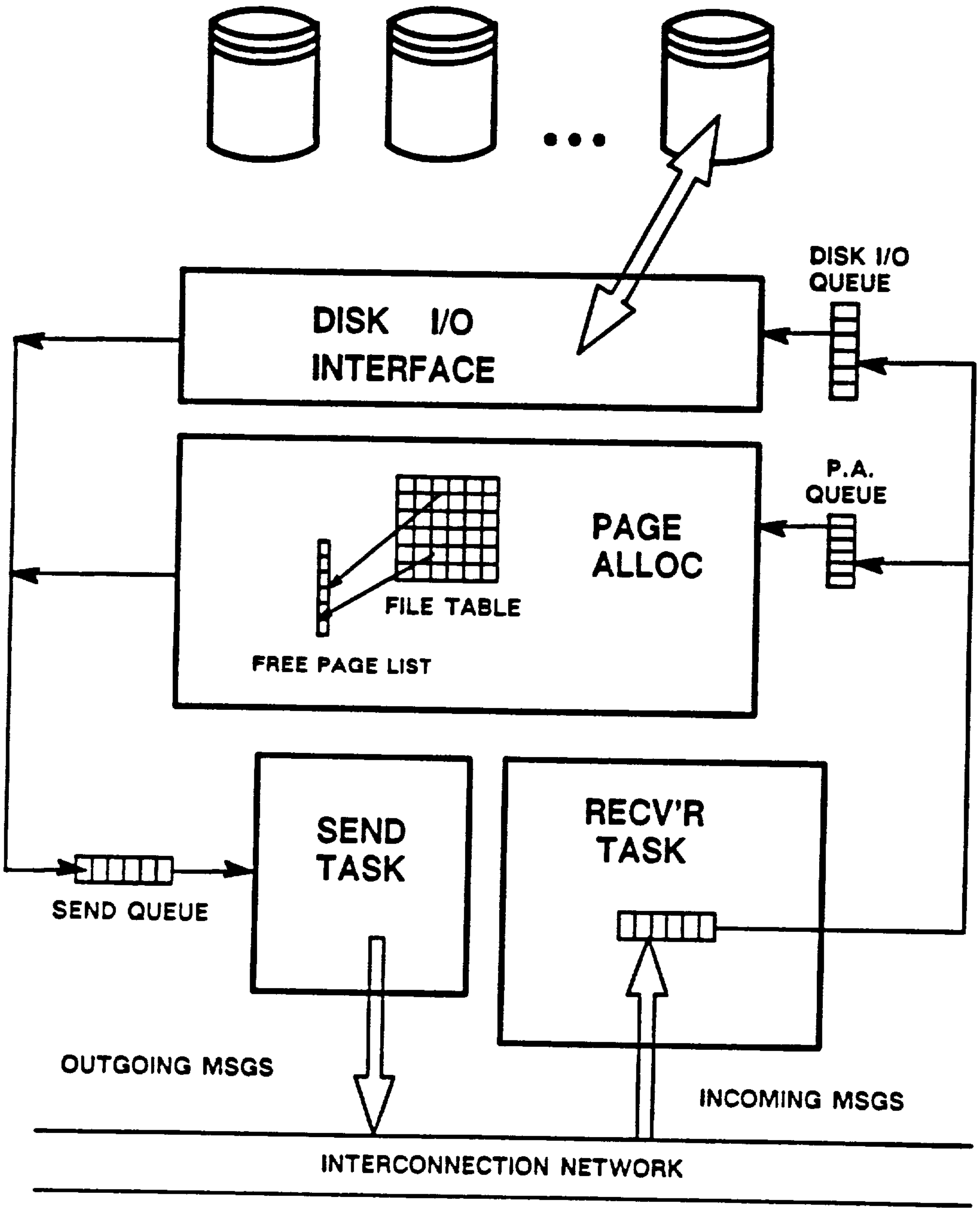


Figure 6.2-1 Disk Manager Configuration

The Disk I/O Interface module receives messages from Servers to write and read objects to and from the disk. This is the only place in the environment that the files are operated on. Thus there is a high degree of isolation from the hardware disk environment.

When a data object is written to a disk file, the Memory Manager will send the file number, record numbers and data corresponding to the object. This information is put into the Disk I/O Queue. The Disk I/O module will write the records to the disk file. When an object is to be "read" from a data base, the Memory Manager will supply the records and the disk file number corresponding to the data object. The Disk I/O task will read the records from the appropriate disk file and send the data back to the Memory Manager.

6.3 CONSIDERATIONS FOR IMPLEMENTATION

Since the Disk Manager must be able to process both page allocation and disk i/o type requests, and since there is a large disparity in the amount of the time spent processing these types of requests, they have to operate independently of each other. Thus, when the Disk I/O Interface is waiting for disks to operate, the Page Allocation task can continue to service requests from Memory Managers. As discussed earlier, the Send and Receive task must also execute independently.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 7

CONCEPTUAL DESIGN OF THE CLIENT PROGRAM

7.1 INTRODUCTION

This chapter describes the functions and components of the Client program. The Client program is used to control the operation of the system. It is responsible for 'booting' the system and for managing the scheduling and synchronization of tasks within the application program. The Client program is also an interface mechanism between the user and the system.

7.2 ORGANIZATION OF THE CLIENT PROGRAM

Figure 7.2-1 illustrates the Client program from the viewpoint of the functional components and internal communications. The initialization component that starts up all other programs, is not shown. The user is shown at the upper right side of the figure. The main components of the client include:

- the *receiver task* - takes messages received from the network and places them in appropriate task queue,
- the *send* task - removes messages from the send queue and transmits them to the receiver task on the destination node,
- the *task manager* task - manages the execution of tasks defined in the application program,
- the *stdin* (*st*andar*d in*put) task - puts data and commands entered by the user into the stdin queue. It permits the user to type-ahead.
- the *send data* task - transfers data from the stdin queue to the send queue,
- the *stdout* (*st*andar*d out*put) task - takes data from the stdout queue and directs it to the user.

PRECEDING PAGE BLANK NOT FILMED

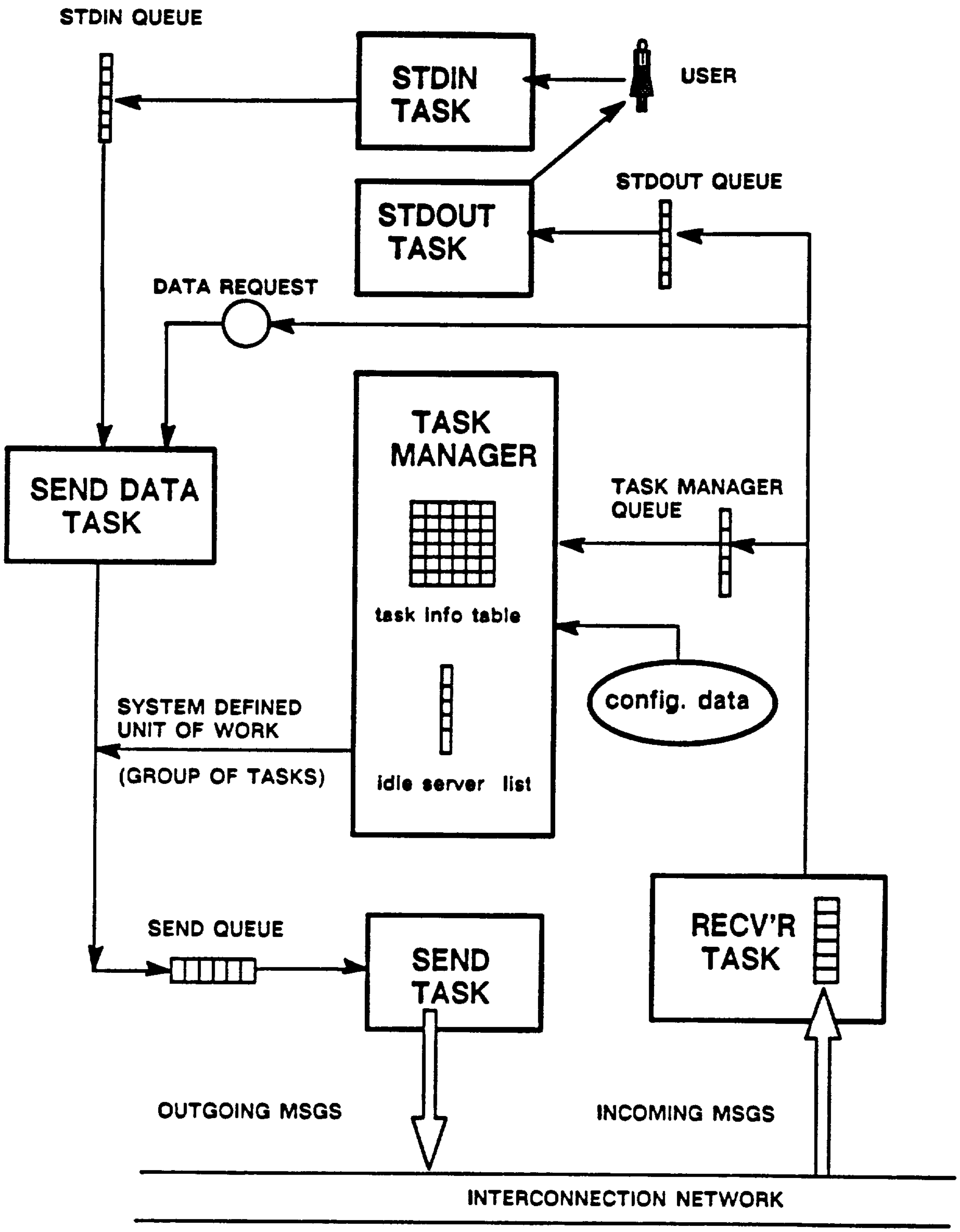


Figure 7.2-1 Conceptual Design of the CLIENT Program

- the *config* file – specifies the configuration of the network as defined by the user. It contains a list of processors that will execute the server, disk manager and client programs. The processor granularity of each server is also identified.

The following sections describe the operation of the Client. The discussion is based on the figure 7.2–1.

7.3 SYSTEM INITIALIZATION / TERMINATION

A configuration file is used in the startup procedure. The file contains information that specifies the number of Servers and Disk Managers and the processors that will execute those programs. The Client will invoke the Server and Disk Manager programs on the processors as specified in the configuration file. The file also contains information about the processor granularity. The latter is used by the Client's scheduler when allocating tasks to Servers (see scheduling and synchronization).

During system initialization, the Servers will "check-in" asking the Client for something to do. The messages from the Servers are put into the Task Manager Queue (shown on the right side of the figure) by the Receiver Task. The Task Manager will update the Idle Server list as the Servers check-in. When all the Servers have checked in, the Client will assign the *monitor* task to one Server.

The Server that executes the *monitor* task is responsible for processing the commands from the user that are sent via the Client. The Server that gets the monitor task executes the serial operations in the program and is responsible for processing user commands. It is the only program that uses the Token Scanner (see figure 5.3–1).

When the user wants to shut down the system, the Client will broadcast a "secure objects" message to all the Servers. The Servers will secure the data objects in primary memory by forwarding them to the Disk Managers. Only those objects that are both in memory and have been modified are written back. After this is done, the Client will broadcast a "shut-down" message to all the Servers and Disk Managers. When this is done, the Client program will stop.

7.4 USER INTERFACE

An interface between the user and the Server programs is required because the Server programs may be executed on remote processors. Any input or user–command can be sent to the appropriate Server program by the Client. Output for the user can be sent from any Server program to the Client who will then forward the information to the user. Examples of the input a user might enter include:

- compile or execute a HLL or DLL application program;
- open, close or format a data base;
- data requested interactively from the application program.

The Stdin Task is responsible for sending the commands and data entered by the user to the appropriate Server. Data and commands are entered into the Stdin queue first. They reside in the queue until one of the Server programs requests data. When a request for data is received, the Send DataTask will respond by sending the next message in the Stdin queue to the Server.

The output that is sent to the user generally consists of a request for data or the result of an operation. The user interacts with the Client program only. Messages sent to the user from a Server are directed to the Client's Stdout queue. The Stdout task will process the message and deliver the information to the user.

7.5 SCHEDULING AND SYNCHRONIZATION MECHANISM

During the execution of an application program, the Client will schedule and synchronize the execution of tasks among the Servers. For this purpose, the Client maintains a *task information table* and an *idle server list*. The task information table shown in the figure, is actually a data structure that monitors those tasks that need to be executed and those that are currently assigned to Servers. The idle server queue consists of Servers that want some work.

The Client assigns *units of work* to the Servers. A unit of work consists of one or more tasks. Tasks are combined to form units of work that are appropriate in size for the processor. In this way the scheduling and communications overhead is amortized over the cycles of the task instructions. Determining the appropriate unit of work for a Server is a function of the relative task granularity, and the granularity of the processor associated with the Server. Further details of the scheduling mechanism can be found in section 8.4.4.

The scheduling algorithm can be modified easily. The current algorithm tries to match the granularity of the *tasks* to the processor granularity. Alternative algorithms include the Guided Self Scheduling (GSS) approach [68] or a combined GSS and matching granularity approach. Further details can be found in the next chapter.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 8

IMPLEMENTATION

This chapter contains a discussion of the details of the important algorithms needed to implement the environment on a real MIMD system. The prototype of the software environment was implemented on a network of Apollo workstations.

The chapter has five parts. The first part describes the model for the low level functional tools that are needed to implement the environment on an MIMD system. Typically, all machine dependencies will appear within this model. The second part of the chapter describes the implementation of the virtual memory system, and shows how virtual memory is incorporated into Server program. The third part describes the implementation of the Disk Manager program and the organization of data bases in the system. In the fourth part, the implementation of the Client program is described. The discussion is focused on how tasks defined in the application program are scheduled and synchronized. Finally, there is a discussion of some of the issues unique to the implementation of software environments on different classes of MIMD computers.

8.1 FUNCTIONAL TOOLS FOR SUPPORTING THE SYSTEM PROGRAMS

The model of the software environment that has been developed during this program of research is conceptually architecture independent. Therefore, it is imperative that the underlying tools that are used to support the operations of the environment also be architecture independent.

This section describes the functional tools that support the operations in and between the Server, Client, and Disk Manager programs. These tools are conceptually and functionally architecture independent; they are generally available on all MIMD machines. However the actual implementation of the tools may vary slightly from one machine to the other because of the differences in their operating systems. The functional components that are required include: messages for communication, multitasking support, events, locks, and buffer areas (queues). Each component is described briefly in the following subsections.

8.1.1 Multitasking

The model for the functional design of the Server, Client and Disk Manager programs in chapters five, six and seven identified the need for the modules within these programs to operate independently of each other. The ability to decompose a program into separate modules that operate concurrently is known as multitasking. In the multitasking environment, multiple threads of execution (i.e. tasks) are able to run within a single address space. The operating system controls the execution of the tasks. Tasks can run concurrently because of a scheduling scheme based on time slicing, priority levels, and events. Procedures called by tasks have to be re-entrant; that is, the procedures must be designed so that multiple tasks can execute the procedures concurrently.

Since all tasks share the data objects of the parent process, events and locks are used to manipulate data objects that are referenced by two or more tasks. Tasks communicate with each other by using messages and shared variables.

The model for the environment implies that the OS scheduling algorithm is invoked whenever an event like "message received" occurs. The OS must then examine task priorities and enter the active task with the highest priority. Most other scheduling models such as time-slicing will work, but they will cause unnecessary delays in getting objects to waiting processors.

8.1.2 Messages

Messages are the most common method of inter-processor communication for distributed memory systems. They are optional for shared memory systems. Messages require two levels of handling; first, there must be routines to build them easily, and secondly, there must be a way of passing them to the various processors in the configuration.

A message consists of two types of information. The first type is the encapsulation data that describes the contents of the message. The second type is the data content itself. The encapsulating header contains the length of the message, the destination and the source.

The communications routines in the SEND and RECEIVE tasks on each processor are responsible for getting messages to their destinations. The details of how to do this are machine dependent. The prototype implementation is described in a later section of this chapter. Since it is machine dependent it is very carefully isolated within the environment so that it can be easily changed.

The environment assumes absolute reliability; all messages must get to their destination.

8.1.3 Queues

While in transit, messages are temporarily stored in circular buffer areas called Queues. Messages are processed on a first-in-first-out policy. The size of a queue is determined by the expected size of incoming messages. Since queues are shared between two or more tasks, a lock mechanism is required to ensure mutual exclusion.

If a queue cannot accommodate the next incoming message, the task that wants to add the message must wait until enough messages have been removed from the queue. If a task is expecting a message and finds the queue empty, it will wait until a message is received. Therefore each queue has two event associated with it: (1) waiting for something to be put into the queue and (2), waiting for something to be taken out of the queue.

8.1.4 Events

Two operations, namely *wait(x)* and *signal(x)* are used by tasks to communicate that a given event has occurred. The first operation causes the task to wait for event "x" to occur. The second operation signals that event "x" has occurred. The most common use of events is to notify a task that data has just been read from or written into a shared buffer area. When a task waits for an event to occur, it does not consume any cpu resource.

8.1.5 Locks

Locks are hardware implemented atomic actions that are used to provide exclusive use of a shared object. High level functions are needed to *initialize*, *lock* and *unlock* a variable that guards a shared object. For this purpose, three functions map to the lock functions used on the given hardware. This is done to facilitate programming and isolate the machine dependent procedures to the lowest levels of operation, thereby facilitating portability.

In the environment, *Locks* are primarily used to provide a task with exclusive use of a queue. They are required for all queues and procedures that can be accessed by two or more tasks.

If two or more tasks attempt to lock the same object, the operating system will only allow one task to go ahead. the other task will automatically wait until the object is unlocked. The event functions are used to signal when queues become available for use.

8.2 IMPLEMENTATION OF THE SERVER PROGRAM

This section describes the implementation of the Server program using the functional components described in the previous section. The implementation of the virtual memory system has a profound effect on the design of the Server program as is described in this section.

8.2.1 Virtual Memory

Chapters five and six described the conceptual design of the Server and Disk Manager programs respectively. This material pointed out the need to implement a mechanism for the Memory Manager and Disk Manager to manipulate data objects within the virtual space of the environment. The concept of a virtual memory system was described in chapter 3. The following sub-sections describe why the virtual memory mechanism was selected and how it was implemented.

The concept of virtual memory is not new; it goes back to its first use on the Atlas computer in the early 1960's. Since then, various forms of virtual memory have been developed, they include paging, segmentation, and paged segmentation systems. Virtual memory techniques first emerged in software and are now incorporated as part of the hardware of many modern machines.

Using the virtual memory system of the target machine was considered as an alternative for managing the data objects in this environment; however there were three fundamental reasons for not using it, they included: (1) virtual memory is not available on many of the distributed memory MIMD computers; (2) those machines that do have virtual memory are unable to support all of the needs of the environment. Typically, the data space is not large enough, and they do not maintain data coherence at the object level; (3) the model described in previous chapters calls for a virtual address space that includes all of the memory and disk space available on the target machine. Consequently, a software based virtual memory system was implemented.

Creating a shared virtual data space among a system of distributed memory modules was an important step in the development of the software environment. Three types of virtual memory systems namely *paging*, *segmentation* and *paged-segmentation* were considered. The primary concern was to maintain the coherence of data objects used by the processors and to perform dynamic memory allocation on a relatively small area of primary memory identified earlier as the data pool. The paged-segmentation approach was selected for implementation purposes. The following subsection describes the advantages and disadvantages of the various virtual memory schemes. It is intended to provided some insight to the issues related to virtual memory for those not familiar with the subject.

8.2.2 Software Virtual Memory

The concept of virtual memory is well known and understood. Deitel [19] and Denning [20] provide a good discussion. Basically, virtual memory divides real memory into regular size blocks called pages or variable size blocks called segments. In some systems, a segment can consist of one or more pages. Every data object is mapped to a page, or to a set of pages, or to a segment. When a data object is referenced in the program, the hardware and the operating system cooperate to insure that the correct page or segment is made available in real memory.

When a pure software virtual memory system is used, it is necessary to introduce the following constraint: The data objects that are brought into memory must be contiguous in the real memory space. This is done to insure that the application subprogram that needs access to the object actually does have access to the entire object. In a hardware based virtual memory scheme, the object does not have to be contiguous in memory because the hardware can dynamically translate virtual addresses into non-contiguous real addresses. In a software based system, once the control of the system is passed to the application program, the memory manager cannot be invoked again until the application subprogram has been executed and control is returned to the Memory Manager. This is not a major issue in algorithms where large objects are represented as hierarchies of small objects.

In paging systems, all pages are the same size. In order to insure that the data object is contiguous in memory, the size of the page must be large enough to accommodate the largest object that can be encountered in an application program. The approach is illustrated in figure 8.2.2-1. This method is complex and creates problems when very large objects are used. The presence of large objects translates into very large pages; it tends to degrade performance for three reasons.

The first reason is that Input and Output (I/O) on large pages tends to cause operating system degradation because performing I/O is very time consuming. The rate of I/O is very slow compared to rate at which the CPU can perform calculations. Hence the cpu must wait for a significant amount of time before it can do "real" work.

The second is that, when large pages are used, one is forced to place many small objects on some pages. If a particular processor needs only one small object, the whole page must be transferred to that processor. This can result in significant overhead. Furthermore, it is possible to encounter a situation in which each of the many objects on a page are

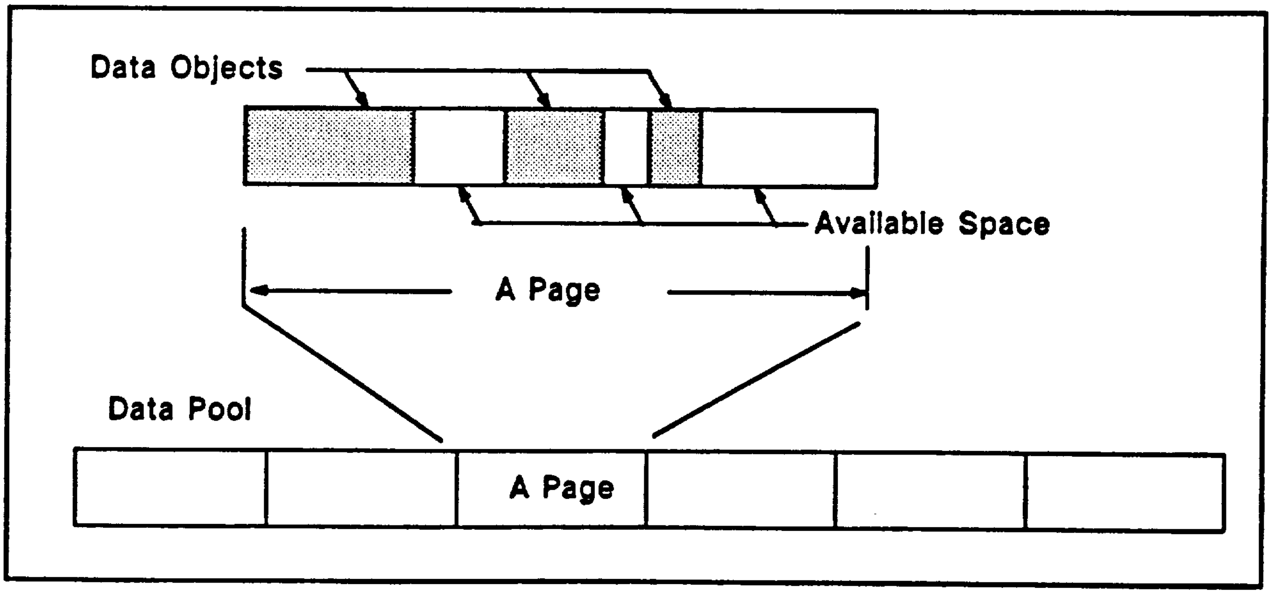


Figure 8.2.2-1 Mapping Objects in a Paging System

required by different processors - thus preventing concurrent operation if some objects are required for read and others for write-access. Such a situation makes maintenance of data coherence almost impossible, and could result in severe degradation of overall performance.

Third. When data objects are deleted from the virtual space, there is *internal* fragmentation on the pages — holes between the objects within the page — that must be handled properly. At the same time there is no *external* fragmentation of the data pool space, because each page exactly fits into a slot of identical size in the memory independently of the holes on each page.

A pure software segmentation scheme overcomes some of the above problems. Each object becomes an individually addressable segment. Inter-memory movements are based on segments, not pages. Therefore, only those objects/segments that are needed by a processor are sent to that processor. Each segment can have its own "access rights", and therefore, the transfer of a segment can be made without having any of the inconsistencies that might be encountered in a pure paging system.

In a segmentation system is no *internal* fragmentation within a segment - it represents a single object. The problem with segmentation is *external* fragmentation: the data pool has holes, as shown in figure 8.2.2-2. In such a scheme it is often necessary to move objects in the data pool and to coalesce holes in order to fit new objects (segments) into the Data Pool. As segments are written back to disk, or are no longer required, "holes" of various sizes are

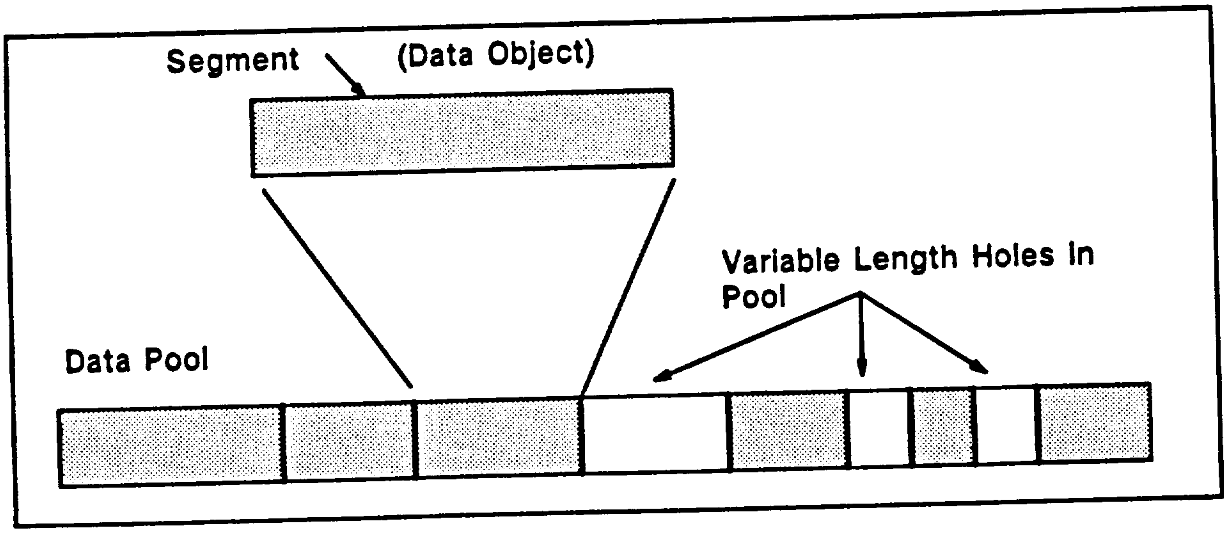


Figure 8.2.2-2 Mapping Objects in a Segmentation System

formed in the real memory space. Objects brought into memory may not fit into any of the available holes, even though the aggregate "hole space" is large enough. Various schemes are available to take care of the fragmentation [47].

A combined segmentation-paging system can also be used. It mitigates some of the problems associated with fragmentation in a pure segmentation system at the cost of some wasted space in both memory and disk. Both are considered to be relatively cheap today. In this scheme memory management is performed on a page size granularity basis. Segments begin on a page boundary and occupy a number of whole pages as shown in figure 8.2.2-3. Generally there is some loss at the bottom of the last page within an object. The advantage of using such a scheme is that the page size can be chosen so that its size is convenient for communication purposes [51]. Small objects will fit on a single page almost anywhere in memory. For very large data objects, the space required in memory can be translated into a number of contiguous pages. Alignment of the multi-page segments in memory does present a problem similar to that described earlier.

In extreme cases, the multi-page segmentation scheme combined with the aforementioned constraint on contiguous blocks may force *garbage collection* or data *compaction* within the real memory space. This is the only serious drawback to this scheme because

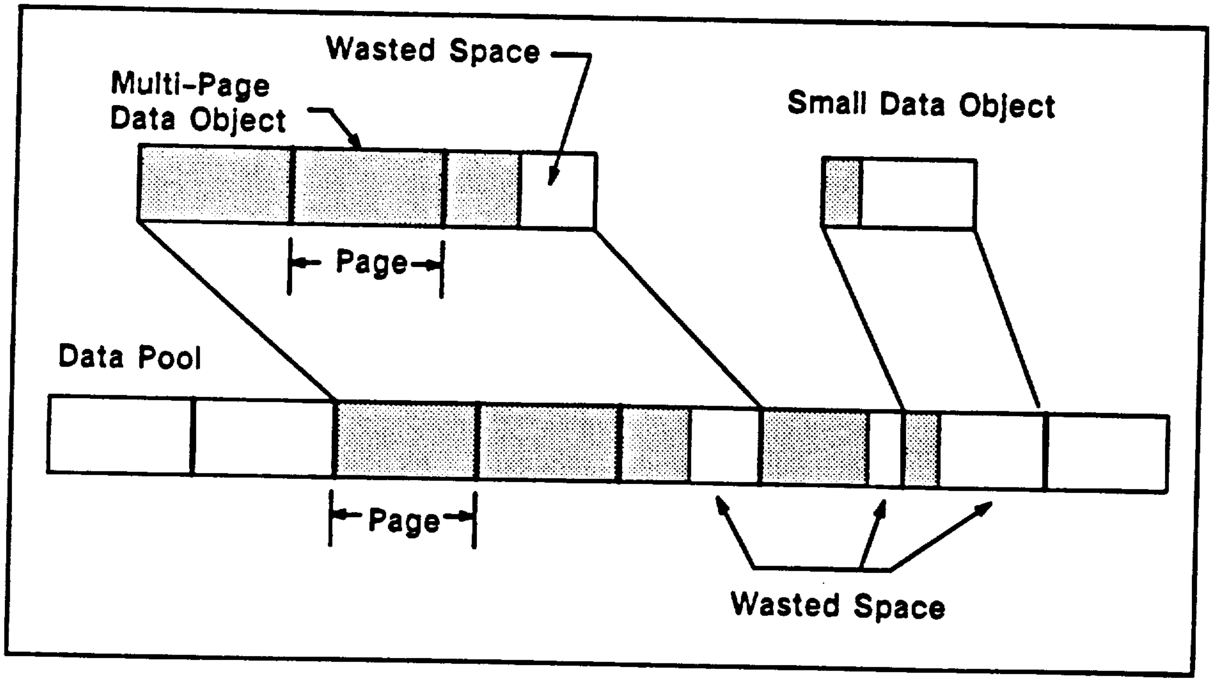


Figure 8.2.2-3 Mapping Objects in a Paged-Segmentation System

system resources are consumed while data is being rearranged. This could be a severe problem in global memory systems because all the processes must be stopped while the compaction is being performed.

8.2.3 Maintaining Data Coherence

The strategy that was incorporated to maintain data coherence in the virtual memory system is similar to the techniques used to maintain coherence of data blocks within a multi-cache system [65], [75], [51]. The implementation of the Memory Manager system is based on some earlier work by Li and Hudak [51]. They examined the problem of data coherence in a shared virtual memory for a network of Apollo machines. They modified the operating system. This approach differs from theirs in that this concept is developed in software, and that this system also performs dynamic memory management. In the environment developed herein, ownership and data coherence is is based on data objects, not pages.

There may be millions of data objects within the data base. It is essential that a Memory Manager to be able to locate an object quickly and to gain access to the information about its use (or state of the object). This concern lead to the concept of a three actor system consisting of the *owners*, *users* and *requesters* of objects. The term *user* refers to the actor component of the memory manager, it should not be confused with an earlier application of the same term to refer to the person that uses the system.

A memory manager that has a copy of an object is considered to be a *user* of that object. A memory manager that wants access to an object is considered to be a *requester* of that object. A memory manager that maintains information about the status of the object is known as the *owner* of that object. Ownership of objects is shared among all Servers in the system.The status of an object includes the list of *users*, *requesters*, and current access rights to the object. The owner of an object can be determined by examining the object identifier (pointer). Distributing the ownership of objects among the MM's alleviates potential bottlenecks. This is especially important when there are several million data objects within the virtual data space.

Any Memory Manager (MM) can own, use, or request objects. Thus each MM must be prepared to assume the role(s) of: *user*; *owner*; and, *requester* when procuring a given object. The interaction between these roles when an object is procured is illustrated in figure

8.2.3-1. Access to an object implies that the MM has the object within the data pool and the object is in the correct state - *read* or *write* mode. Therefore a MM can be a *user* of an object, but may not have the correct access. This situation occurs when the MM has the object for *read* access and wants to get the object for write-access. If the object is already in memory (Data Pool) with the correct access, the Memory Manager will simply return the location of the object to the Data Manger.

When a MM requires access to a data object, that MM assumes the role of the *requester*. The *requester* determines the *owner* by using a simple algorithm that is a function

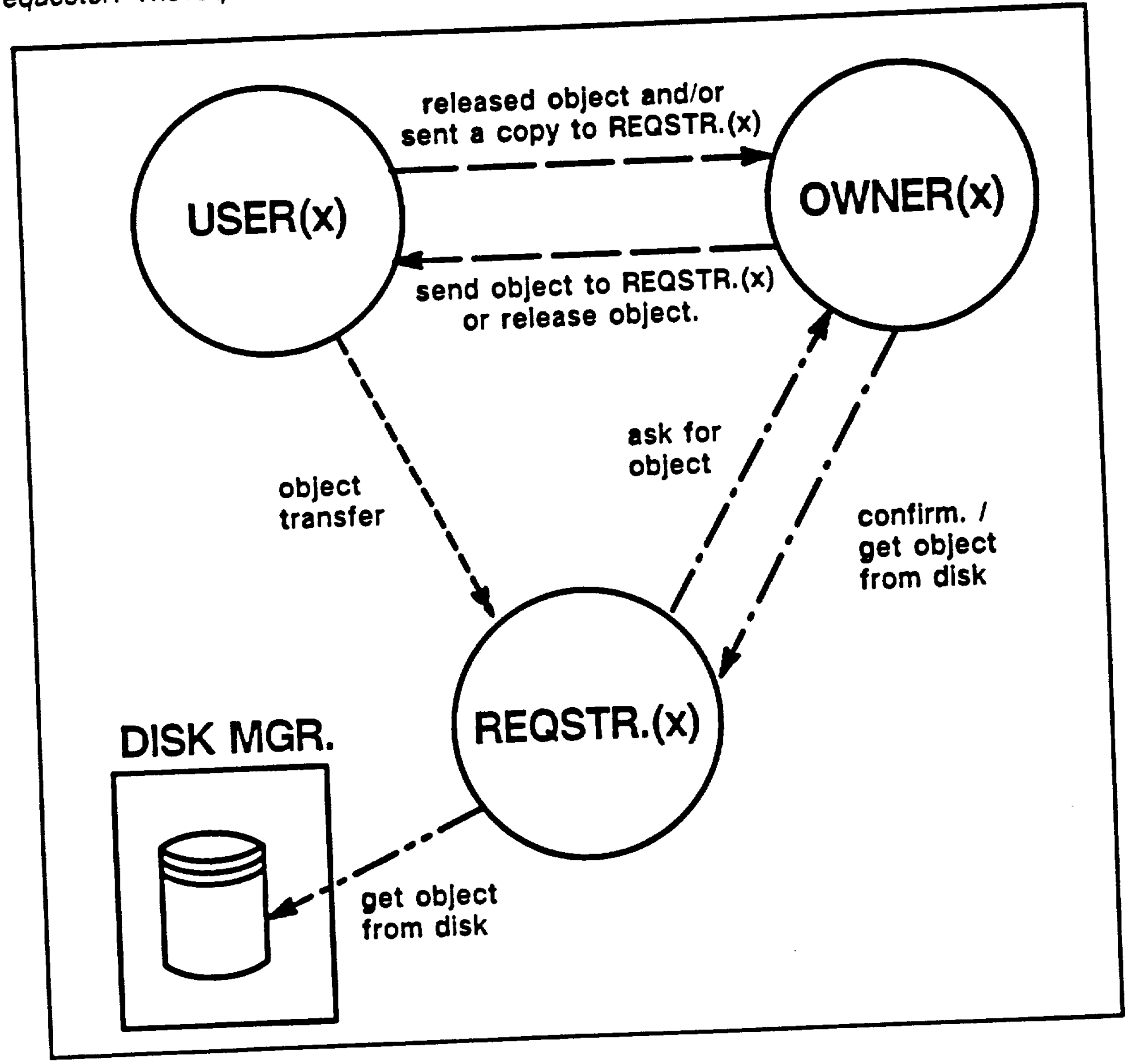


Figure 8.2.3-1: Owner / User / Requester / Disk Mgr. Interaction

of the number of memory managers and the object pointer. The *requester* sends a message to *owner* asking it for permission to use the object.

The message sent to an *owner* includes information about the access required and if a copy of the object is needed. The *owner* examines the current status of the object and makes a decision with respect to data coherence. There are several scenarios that can manifest themselves when the message is received by the *owner*. The following list describes the four most basic cases that can occur and the corresponding action that is taken:

1. There are other requests pending for the requested object; current request is put into a queue.

2. The requested object is not in use; *requester* is told to get the object from disk.

3. The requested object is in use, and there is no access conflict; the *owner* will ask the a *user* to send a copy of the object to the *requester*. After the *user* has sent a copy of the object to the *requester*, the *user* will inform the *owner* that a copy was sent to the *requester*.

4. The requested object is in use, and there is an access conflict; current *users* are told to relinquish their copy of the object as soon as possible. After the *users* have released the object and informed the *owner*, the *requester* is permitted access. If the *requester* needs a copy of the object, one of the *users* is told to send a copy before releasing the object. If at the time of the request, the *requester* is also a *user*, it will wait until all the other *users* have given up the object. The *owner* will inform the *requester* when to proceed.

Figure 8.2.3-1 is a simplified overview that illustrates the mechanism required to get an object. In a message passing system, the latency (delays) in sending information across the network can be significant. By the time a message reaches its destination, the status of the objects within the system can change. Furthermore, messages between the *user*, *requester* and the *owner* may "cross" each other, and this may lead to messages that are processed in an order that is different from that which was originally intended.

In order to avoid the problems related to latency and asynchronization, the system can be programmed so that confirmation messages are sent back with every request, but this approach forces serialization and degrades performance considerably. Alternatively, if all the different types of problems can be identified, and if there are not too many of them, contingencies for each type of problem can be incorporated in the design of the memory manager. This approach may lead to much better overall performance at the cost of some extra software. It was adopted herein. Further information on this subject is discussed in the next chapter, entitled "application problems."

8.2.4 Organization of the Server Program

The organization of the memory manager in the server program is illustrated in figure 8.2.4-1. The shaded portion of this figure is actually an exploded view of what really happens in the MM module of figure 5.3-1. This tripartite approach to handling the coherence problem naturally maps to the three Memory Manager Tasks. The following subsections describe the function of the OWNER, USER and Local Memory Manager (LMM) tasks. The SEND and RECEIVE tasks operate as described earlier in chapter five.

The OWNER task is responsible for knowing the state of every object within it's domain. The domain is defined by using the "mod" function of the object pointer and the number of Servers. The *Owner Table* is used to identify the access of the object (read or write), and a list of all the *users* and *requesters*. The owner table is used exclusively by the owner task. The owner queue is a shared between the *owner* and *requester* task, hence, locks and events are employed to insure the queue is accessed properly.

The USER task is responsible for processing the messages sent to the Server from the *owner*. There are two types of messages processed by the USER: "send a copy of the object" ; and, "release the object." The *Object Table* identifies the location of the objects that are currently in the Data Pool. It provides information on how the Data Pool is being managed. The LMM and the USER tasks of a Memory Manager share the object table and the Data Pool. Their interactions are kept consistent via a set of local shared variables and mutual exclusion locks. The user queue is shared between the *user* and *requester* task. Hence, the same considerations apply to the access of it as do the owner queue.

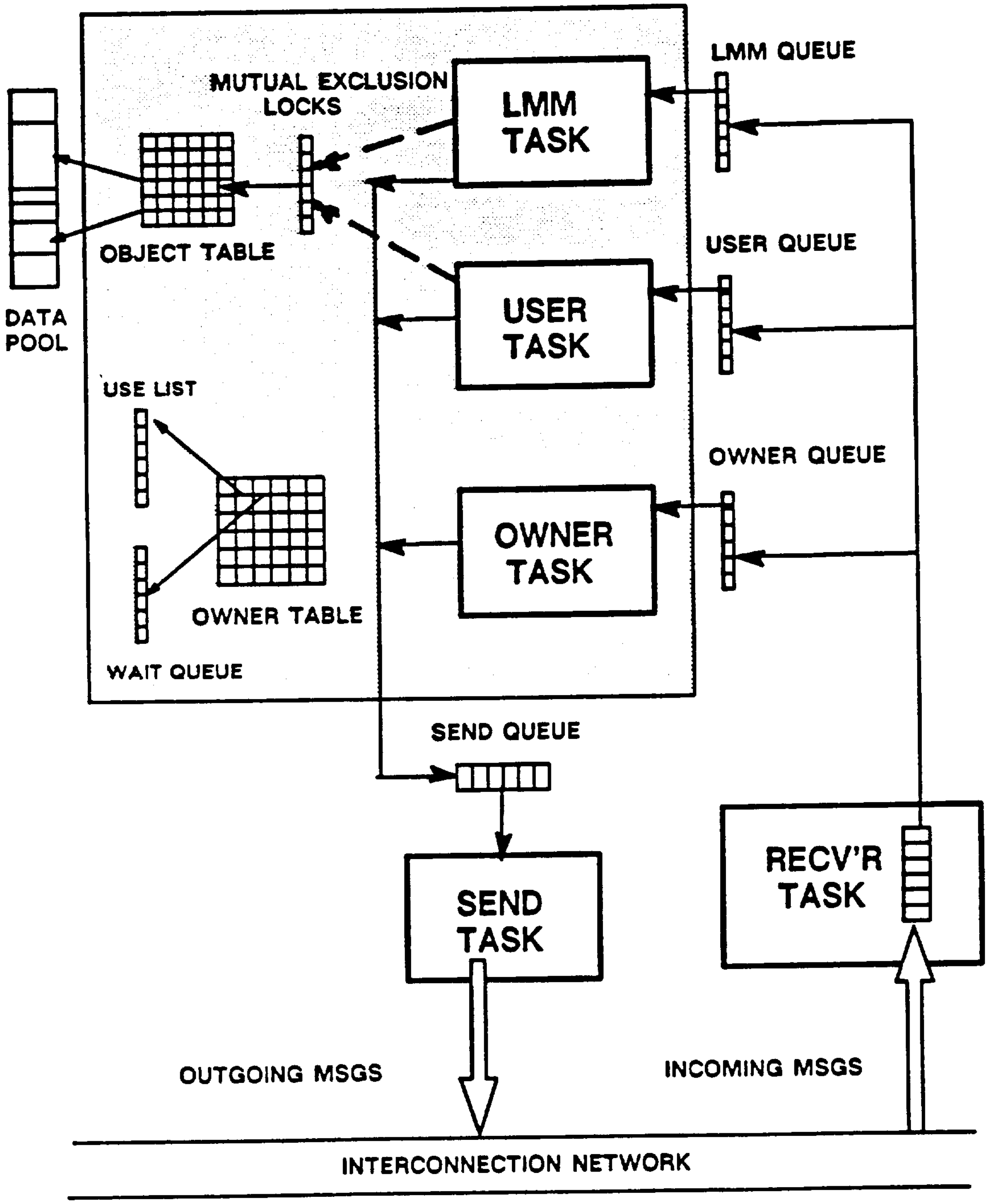


Figure 8.2.4-1. Implementation of the Server Program

The LMM is the main task that drives the server program. It contains IL Code Loader and Interpreter, Token Scanner, Data Manager Routines, and Application Subprograms. It is the task that issues the request for an object. Hence, it is the the *requester* component of figure 8.2.3-1. This task also contains the DDL and HLL compilers that were described in section 5.1. The LMM queue is shared between the *LMM* and *requester* task. Hence, the same considerations apply to the access of it as to owner queue.

8.3 IMPLEMENTATION OF THE DISK MANAGER PROGRAM

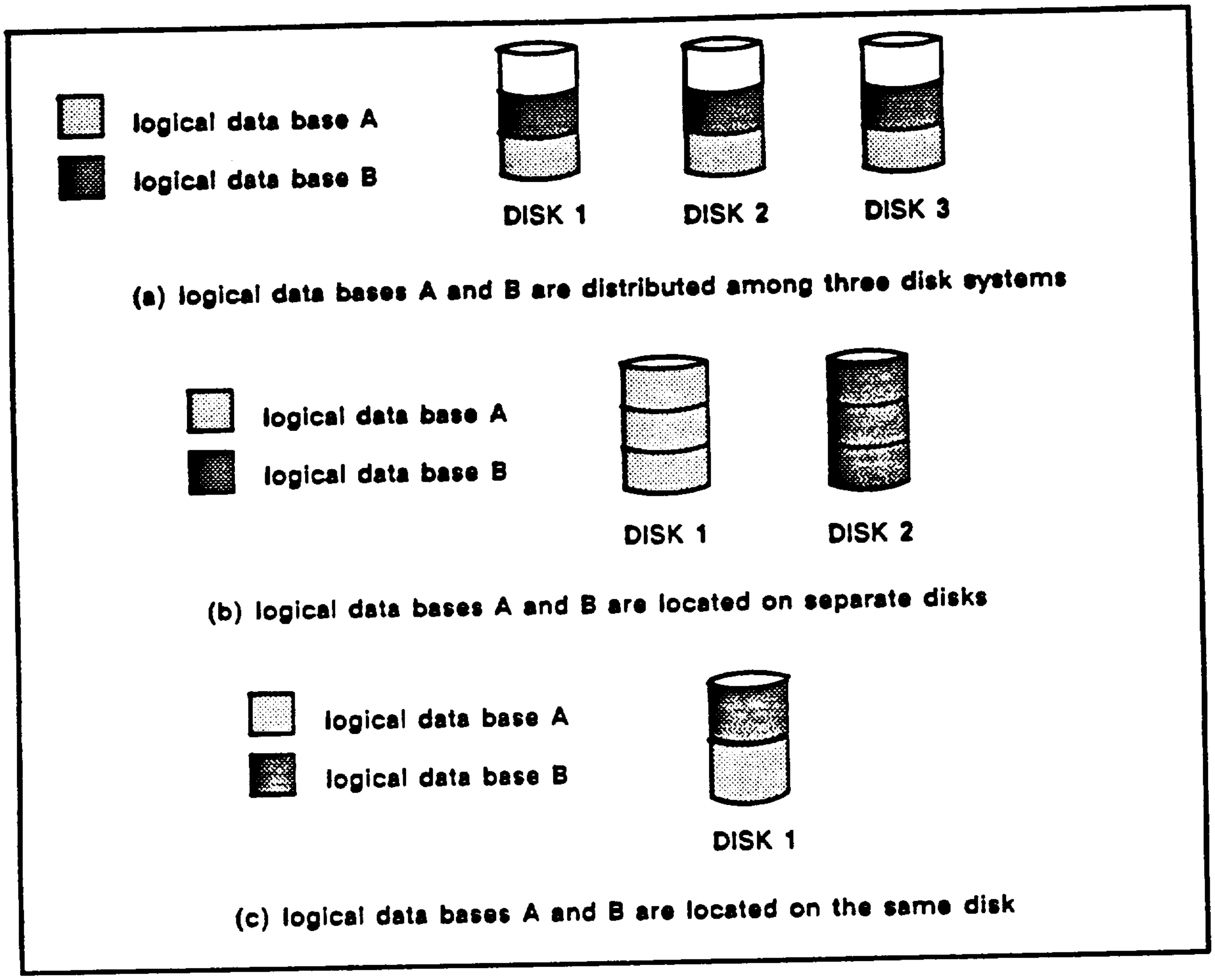
In this section, implementation of the disk manager program is described. The organization of tasks within the program is identical to what is described in chapter six. The program is subdivided into four tasks as shown in figure 6.2-1. They are the Disk I/O, Page Allocation, Sender and Receiver tasks. The functions they perform were described in chapter six, hence, they will not be repeated here. The following sub-sections describe how the data bases are organized.

8.3.1 Mapping the Data Objects to Disk Files

The totality of all the data objects is called the data space. A programmer may choose to break it down into logical units called data bases. Large data bases can be subdivided into sets of random access files. Each file can be mapped to a single disk on the I/O system. Hence a logical file can be mapped to several physical files.

Figure 8.3.1-1 illustrates the flexibility provided by the disk management system. In figure 8.3.1-1a, the logical data bases A and B are distributed among 3 disks, hence there are six distinct physical files, two for each disk. In figure 8.3.1-1b, logical data base A resides entirely on disk 1 while logical space B resides entirely on disk 2, hence there are two distinct physical files, one on each disk. In figure 8.3.1-1c, the logical data bases A and B are mapped to individual physical files on the same disk. The user can choose any combination of patterns for organizing a logical data base.

The records of each file are identical in length and, when mapped to the data pool of a Server, are customarily called pages. Each object requires one or more pages in a single data base. If a data base is distributed over a set of disk files, then multi-page objects are



(a) logical data bases A and B are distributed among three disk systems

(b) logical data bases A and B are located on separate disks

(c) logical data bases A and B are located on the same disk

Figure 8.3.1-1 Mapping Logical Data Spaces onto Disks

distributed too. The latter has important ramifications on the performance of obtaining large objects for secondary storage.

When the Memory Manager gets an object from disk, it has to specify the data records that make up the object. The information about a record is packed into a pointer. The pointer identifies the disk file and record number. The pointer is described next.

8.3.2 Object Pointers and Data Base Size

The pointer or "token" that identifies an object contains enough information to retrieve the first page of the object. Objects always begin at the top of a page. If it is a multi-page object, the other pages are identified in the encapsulation data on the first page. This design has important implications regarding the time required to retrieve objects from disk.

A multi-page object may be retrieved from secondary storage much faster if it is distributed among several disk systems instead of being located on a single disk file because multiple disk managers can simultaneously perform i/o operations on various parts of the object. This occurs after the first page of the object has been retrieved by the memory manager, and the tokens for the rest of the object have been identified and sent to the appropriate disk managers.

The token that represents an object consists of a) the physical file number, b) the length of the object (number of pages), and c) the record number in the physical file. The number of pages is included in the token so that a processor requesting an object can allocate the space its data pool prior to requesting the object. The current make-up of each 31 bit token is:

- 3 bits to specify the number of physical files over which an object can be distributed,
- 8 bits to specify the size of the object in terms of the number of pages,
- 20 bits to specify the record (page) number in the physical file.

The 32nd bit is always zero to prevent problems with FORTRAN's interpretation of integer data. Thus, the maximum size of a data base is limited to 8 distributed files, each containing approximately 10^6 pages. If a 1000 word page is used, each data base can reach about 32 Gigabytes in size. Since, the system can handle more than one data base at a time, it is likely that *users* will run out of hardware before reaching the limits on data base size.

System programmers can easily change the allocation of bits to suit their needs. For example, the maximum size of an object is limited by the maximum number of pages an object can occupy and the page size (which does not appear in the token). With the current token that is 256 (8 bits) 1000 word pages, or about 1 megabyte per object. One could easily use 10 bits for maximum object size and 18 bits for the maximum number of pages. That would result in a system with a maximum of 256,000 pages containing objects up to 4 MB each. Alternatively, one could use larger pages to increase the maximum object size. However, since objects always start at the top of a page boundary on disk and in memory, small objects result in significant losses in disk space and real memory when large pages are used.

8.4 IMPLEMENTATION OF THE CLIENT PROGRAM

The organization of the Client program was implemented exactly as described in chapter seven. Hence, this section describes only the details of the strategy employed for scheduling and synchronizing the tasks encountered in the application program.

For prototypying purposes, a centralized scheduling system was implemented in the Client program. This system controlls the execution of the application program. In a centralized system, the "task manager" knows the status of every Server in the system. Therefore, allocating tasks to Servers is easily accomplished. In a distributed scheduling system, the status of the entire system is much more difficult to manage because the status of the Servers is modified by several sources.

The algorithm used for scheduling the tasks among the Servers was based on matching the granularity of the "units of work" to the processor granularity of the Server. A "unit of work" consists of one or more tasks. The following sub-sections describe how the scheduling process works.

8.4.1 Relative Granularity

The relative granularity of a task is the programmer's interpretation of the relative size of the tasks encountered in the application program. It is important because it can provide the system with information on how to balance the load for a given type and set of processors. Selection of relative granularities must be determined by considering the combined effects of the algorithm and the size of the problem.

If the granularity of a task can be defined relative to a standard task, the load balancing procedure is simplified considerably because the scheduler can easily determine how much work to assign to a given processor. The expression of standard task granularities is called relative granularity herein. The concept is new, and is a key component of the proposed solution.

Several alternatives exist for expressing relative granularity. They include: a linear scale; a logarithmic scale; or simple parameters defined by the variables *coarse, medium* and *fine*. The relative granularity of a task is specified in the application program. Examples were shown in figures 3.4.2-1 and 3.4.2-2. The prototype was developed using a linear scale.

8.4.2 Processor Granularity

The processor granularity is specified in the configuration file. It represents the ideal size of a unit of work that should be assigned to a Server program that is running on a given processor. The processor granularity must be specified in the same context of the values specified in the application program.

The processor granularity can be determined by executing sample portions of the application code on a given processor. For example, iterations of a do-loop can be executed on a processor. Then, using the information about how much time was taken and the relative task granularity, the programmer can establish the processor granularity. Alternatively, if the programmer knows the amount of calculation involved in a given task, then the relative granularity of that task can be used to set the processor granularity.

When an application is ported from one machine to another, the absolute granularity of the new system can be determined by modifying the original processor granularity values by considering the change in the processor, memory, and communications bandwidth. For example, if the application is ported from a system in which the processor are twice as fast as the current system, then the processor granularities are divided by two in the "config" file (figure 4.2-1).

8.4.3 Mapping Relative Granularity to Processor Granularity

If the tasks in the application program are too fine grained, it may be very inefficient to exploit the concurrency at that level for a hardware configuration that has a few very fast processors. Similarly, it may be very inefficient to schedule a few coarse grained tasks among a system consisting of many relatively slow processors because not all the processors get to participate (poor load balancing).

Hence, the scheduler in the Client will try to assign the appropriate number of tasks to a Server. This is accomplished by combining or "chunking" tasks defined in the application program. For example, if a parallel do loop is encountered with a granularity of 10 (linear scale), the scheduler will allocate two iterations to a Server with a processor granularity of 20, or ten iterations to a Server that has a processor granularity of 100.

The *user* can set the absolute granularity so that the scheduler is forced to exploit the concurrency at a given level. For example, consider an application that consists of three nested parallel do–loops, in which the relative granularities are 100, 10 and 1. If all the processor granularities are set to 100, only the highest level of parallellism will be exploited. The lower level do–loops will be executed serially.

The examples above indicate that there need not be a direct relationship between programmed parallelism and run time tasks because of the disparity in performance between various systems.

A much more sophisticated system could be developed so that the processor granularity of a given Server is updated dynamically during the execution of an application. This feature can be utilized in a multiprogramming environment; i.e., one in which several *users* are running multiple computer jobs on the same system.

8.4.4 Automated Synchronization and Scheduling

The Client is responsible for the scheduling tasks among the Servers. The information regarding parallel streams of task execution are located in the instructions that are interpreted by the Server program. When the Server encounters this information during the execution of the application program, the Server will send the information to the Client which will place the information in a work queue. The Server will then ask for some work. The information sent to the Client generally consists of:

- type of task – iterations of a parallel do loop, or *user* defined tasks,
- loop indices or number of tasks and corresponding granularity,
- location of tasks within the program; where the tasks fork (begin) and join (end).
- the environment object – this data object contains the common variables at the time the parallel stream was encountered.

After updating the "work queue" with the parallel task information, the Client will assign the first unit of work from this stream to the Server that encountered the stream. If there is more work to be spawned from this stream, the Client will assign work to other idle Servers as they become available.

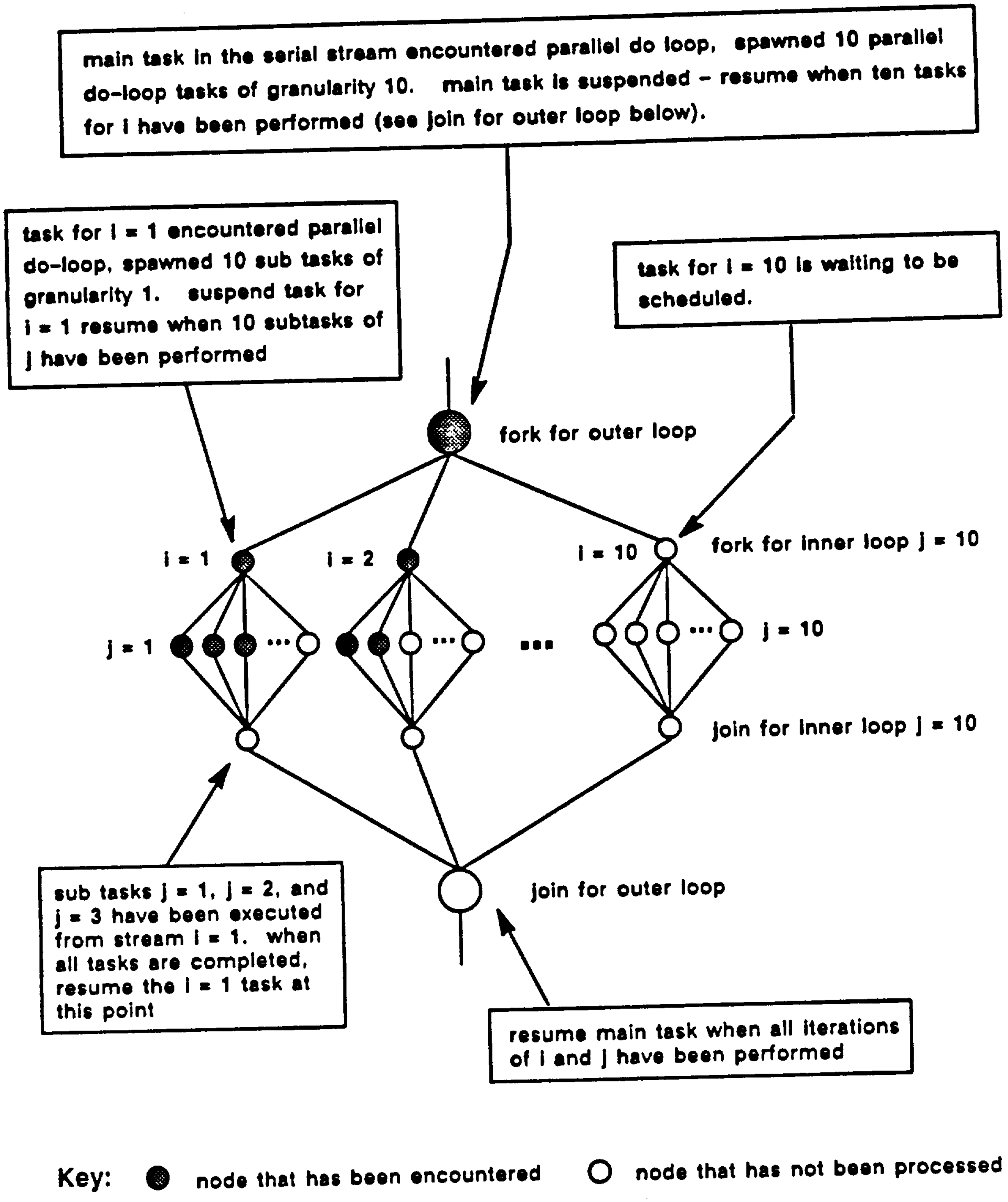
The "work queue" is actually a hierarchical data structure that is used to keep track of: active tasks – and those Servers that are executing them; suspended tasks – and those Servers that were suspended; tasks that are waiting to be executed in a given stream; the tasks that spawned parallel streams; the task granularity information regarding task; and, the environment object associated with streams. This information is used by the task manager to correctly assign the tasks to the Servers. When a Server finish its unit of work, it will check back with the Client asking for more work. The Client will update the "work queue" and then try to assign the Server another unit of work.

Figure 8.4.4-1 illustrates the scheduling procedure in more detail. It shows the task execution graph for the matrix addition problem described earlier in section 5.1.2. The node at the top of the graph corresponds to the point in the program when the first parallel do loop is encountered. It is assumed that this point is encountered in the "main" serial task (although it does not have to be).

The parallel task information is sent to the Client – who finds out that there are ten tasks, each one of granularity of ten. The main task is suspended and tasks consisting of an iteration at the outer loop are assigned to Servers.

As the Servers execute the outer loop tasks, they will encounter a second level of concurrency. This information is sent back to the Client. The figure shows that iterations i =1 and i=2 were assigned to Servers when the second level of concurrency was encountered. The Client will suspend those tasks and schedule the iterations of the inner loop. When all the inner loop tasks have been performed, the Server that was assigned the outer loop is informed to resume the execution of the outer loop.

When all of the outer loop iterations have been performed, the Client will resume the execution of the main task.



main task in the serial stream encountered parallel do loop, spawned 10 parallel do-loop tasks of granularity 10. main task is suspended - resume when ten tasks for i have been performed (see join for outer loop below).

task for i = 1 encountered parallel do-loop, spawned 10 sub tasks of granularity 1. suspend task for i = 1 resume when 10 subtasks of j have been performed

task for i = 10 is waiting to be scheduled.

sub tasks j = 1, j = 2, and j = 3 have been executed from stream i = 1. when all tasks are completed, resume the i = 1 task at this point

resume main task when all iterations of i and j have been performed

fork for outer loop
i = 1
i = 2
i = 10
fork for inner loop j = 10
j = 1
j = 10
join for inner loop j = 10
join for outer loop
Key:
● node that has been encountered
○ node that has not been processed

Figure 8.4.4-1 Task Scheduling for Matrix Addition Example of Figure 5.1.2-1

The Client's scheduling policy will try to enforce data locality by assigning units of work from a given stream to a particular Server. For example, if the Server granularity is 1, then the Client will try to assign all the "j" iterations for a given value of "i" to a particular Server.

If the granularity of a given loop is too small, the Client will "chunk" iterations of that loop. For instance, if the processor granularity of a Server is 5 units, and the inner loop relative granularity is 1, the Client will schedule 5 iterations of the inner loop to the Server. If the granularity of the Server is 20, the Client will try to schedule two iterations of the outer loop (gran 10) to the Server. Hence, that Server will execute the inner loops of both outer loops serially.

Since the scheduler is an isolated component within the system, it can be easily modified to incorporate different scheduling strategies such as Guided Self-Scheduling [68] and Pre-scheduling of do-loops .

In the former case, the iterations of the do-loop are assigned to the processor based on how many processors are being used and how iterations of the do-loop have to be done at the time of scheduling process. For example, if there are 100 iterations of a do loop and five processors, the first processor gets (100/5) 20 iterations, the second processor gets (80/5) 16 iterations, the third processor gets (74/5) 14 iterations .... The principle behind this scheme is to try to get all the processors to finish at the same time. The delay in getting processors started on their tasks is offset by assigning fewer tasks (iterations).

In the pre-scheduling scheme, a given processor is restricted to executing a subset of loop iterations. For example, if there are four processors, and the loop values begin at one and end at 20, processor 1 will be assigned iterations 1, 5, 9, ... whereas processor 4 will be assigned iterations 4, 8, 12, ... The advantage of this scheme is that data locality is usually enforced.

8.5 SPECIAL CONSIDERATION FOR PARALLEL PROCESSING

This section focuses on special aspects of implementing an environment on MIMD computers. Since the model of the environment is architecture independent, it can be implemented on a variety of machines. Based on the resources that were available, it was decided to initially implement the environment on a distributed memory system - specifically, a network of Apollo workstations. *The following section is an example of how the environment is mapped onto the hardware of Apollo system.*

8.5.1 Mapping the Environment on Apollo Workstations

Apollo's Concurrent Programming Support (CPS) was used to implement the functional design of the programs. CPS consists of a set of routines that create and manage a multitasking environment within a single process. Hence, the Server, Disk Manager and Client programs were decomposed into the tasks as described earlier using CPS. Further information on the CPS system can be found in [6].

The communication system in the model is based on a mechanism that will guarantee that a message will be received at the destination – only one message has to be sent from the source. This type of messaging system is often referred to as the "reliable datagram protocol." It is very well suited to support asynchronous message passing between tasks. However, the protocol was not available on the Apollo system at the time the prototype was developed. Instead, fully connected UNIX sockets were used to provide the programming model for the prototype. The prototype employs an ARPANET addressing scheme and a User Datagram Protocol (UDP) to interconnect the modules. This scheme demands more memory overhead and restricts the number of processors, but is adequate for the prototype. It also appears to be very slow. It is a very isolated aspect of the system. Future versions will probably utilize very specialized high speed networks and reliable datagrams.

In the CPS system, the Aegis operating system will "swap" tasks in and out of operation based on a time slicing policy combined with the information about the priority of tasks. This is analogous to "time sharing" at the task level. This concept does not map exactly to the model developed herein, and consequently, the operation of the Server program and the virtual memory system need some special consideration for implementation. These concerns were primarily related to processing messages sent between tasks.

8.5.2 Observations On Message Passing Systems

In the prototype system, messages are sent asynchronously, and in most cases (90%) there is no confirmation message sent back. This policy was adopted so that the amount of message passing is minimized, which has important consequences on the performance of the application problem. Using this approach there is no guarantee that a given message will be processed in the order that it was sent or received by the receiver task at the node. Furthermore, the latency of the communications system will result in messages that cross each other. The alternative is to send back a confirmation message for any data request. Lastly, the use of shared variables combined with messages as a form of communication requires special considerations.

8.5.3 Shared Memory Considerations

The total environment must be capable of operating on a variety of MIMD architectures. Some of them were illustrated in figure 1.2-1.

However, shared memory computers are the only other viable and commercially available taxonomy as of this writing. Therefore the discussion in this section is primarily focused on how the prototype system maps to a shared memory architecture. There are two questions that arise; "how does the three program (client-server-disk manager) concept map?," and "how does each of the individual programs map?."

The answer to the first question is that one could merge the Disk Manager, Server, and Client programs in a shared memory system. A partial prototype was developed during the early phases of the investigation. In essence, the object and the owner tables were consolidated, and all communication was accomplished by using shared variables. A monitor approach was used to ensure mutual exclusion.

However, it is not necessary to merge the Client, Server and Disk Manager. There may be significant advantages from both the flexibility and the maintenance points of view, to leaving them as three programs. For example, in the three program approach, one could place a more sophisticated Client on a workstation with the Servers executing on the special purpose hardware of the shared memory system. The user, who interfaces with the Client, would derive all of the benefits of the workstation. Graphics, multiple windows, mass storage ... all become possibilities. In this scheme the only connection to the multiprocessor is the network connection to the Server, which is internal to the environment. If the environment is reduced to one program, the Client would run on the MIMD system. The latter would need to do all graphics etc. too.

The Disk Manager is maintained as a separate program because it provides the only way to achieve parallel I/O on several different hardware configurations.

If the three-program concept remains in place, the shared memory can be treated as "n" separate memories as illustrated in figure 8.5.3-1. Each section of the memory is used exclusively by one of the Server programs, the Client, and/or Disk Managers. In such a scheme the SEND and RECEIVE tasks of each program simply move data from one processor's queues to another processor's queues. This is a very simple change, but the environment carries along a significant overhead from the distributed memory system concept. For example, there are multiple copies of the objects in the total memory that are "read-only." Ideally, if memory were treated as a single entity, there would be only one copy of such objects. In addition, there is unnecessary movement of objects between queues . One move is too many!

There is one significant advantage to this approach. There is a high likelihood that the memory used by a particular processor is in a hardware box that is not accessed by other processors. Contention for memory is minimized. In shared systems this contention has been a major problem - one that limits the number of useful processors that can be placed in the system. Thus, this scheme provides a tradeoff between carrying along software overhead and solving the hardware contention problem.

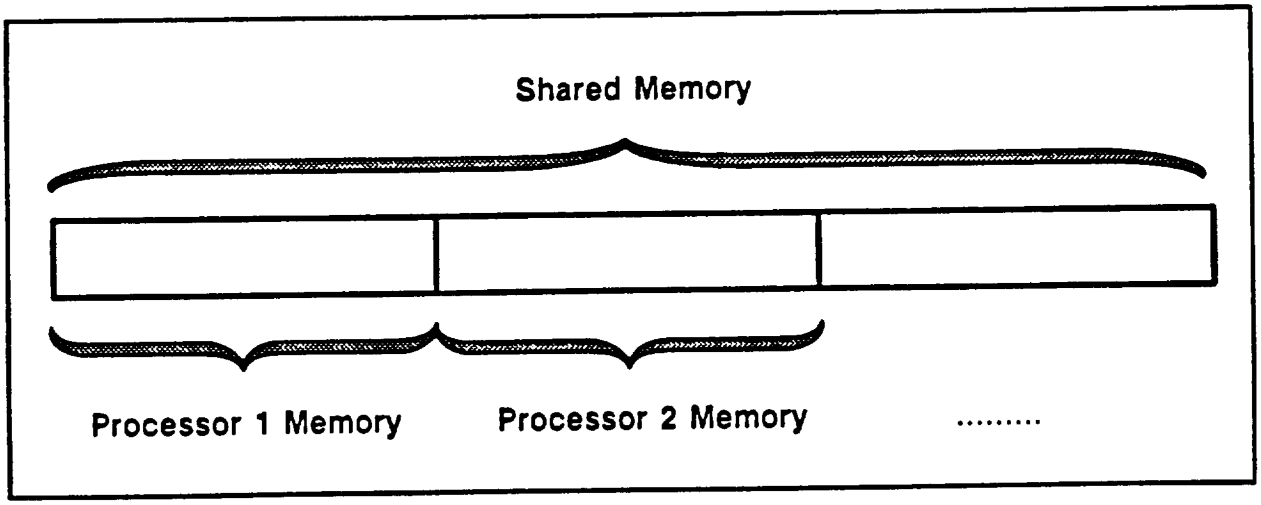


Figure 8.5.3-1 Segmented Shared Memory

The distributed environment can be modified to a "true" shared memory environment by keeping the Client-Server-Disk Manager concept, but making changes in the memory management module of the Server. Then, Send and Receive tasks would employ shared variables and locks for communications purposes.

In this scheme the memory manager in each Server operates as a monitor. Each processor has a Server that can assume the responsibilities of the system wide memory manager via a series of mutual exclusion locks. When one processor is the memory manager, other processors can operate on data that they already have, but cannot access the information in the memory management tables for those entries that are locked.

The user and owner tasks are eliminated in the shared memory module because there is only one global memory. The object and owner tables (figure 8.2.4-1) are consolidated into a single table. This table is used to maintain information about the location of objects in pool, and the list of users and *requesters* for every object in pool. In the monitor approach, all processors can become the owner of an object on an as-needed basis.

In this system, there is a potential conflict each time a Server begins executing a new DBMS sequence via the HLL; i.e., each time a subprogram is about to be called. Prior to calling a subprogram the DBMS must find the needed objects. It may have to compete with other servers for every object required by DBMS and the application itself. This implies acquiring the locks for entries in the object table and acquiring access to the data pool.

Management of the global memory by the Memory Manager is much more complex than in the distributed memory case because now there are *n* processors that share the data objects in the pool. The management of the memory pool takes on an added dimension. When memory reorganization is necessary, as when large multi-page objects are created or brought in to memory, all processors must be stopped at the end of their currently executing HLL instruction. They cannot continue to work on their current data objects because the latter may be moved in this process. After reorganization, all processors can resume operation.

Thus, it is not clear that there are significant advantages to the truly shared memory system. Any single advantage seems to be offset by a significant, and potentially more serious disadvantage!

CHAPTER 9

APPLICATION PROBLEMS

The development and application program for solving two demonstration problems is described in this chapter. The problems are, a hyper-matrix multiplication, and block Cholesky decomposition. For each problem, there are three parts: description, programming notes, and results. These problems were selected to illustrate the features of the software environment and demonstrate the performance of the system. The discussion of the application problems is preceded by a description the hardware that was used.

9.1 HARDWARE USED FOR DEMONSTRATION PROBLEMS

The environment was prototyped on an Apollo network. The network consisted of 60 nodes. They nodes were DN3000's, DN3500's, and DN4000's. The application programs were restricted to use of the DN3000's for experimental purposes. They have a Motorola 68020 processor with a 68881 floating point chip. Each processor has 4 MB of RAM. The virtual memory capacity is 65 MB. The working set of the operating system is approximately 2.5 MB of the 4 MB of RAM. Some of the processors have disks, others operate diskless -- it is transparent to the environment developed herein. In the experiments, Disk Managers were always executed on processors with disks. Servers and the Client programs were executed on either type (disk / diskless) of computer.

9.2 APPLICATION 1: MATRIX MULTIPLICATION

9.2.1 Description of the Problem

This test problem computes the product C = A*B. Where A, B , and C are defined as hypermatrices. A hypermatrix is a hierarchical data structure in which each element of a hypermatrix is a submatrix. Figure 9.2.1-1 shows the hypermatrices used in the application problem.

In order for the problem to be realistic, the size of the submatrices were chosen to be similar to those used in existing general purpose structural systems; both FINITE and ASKA

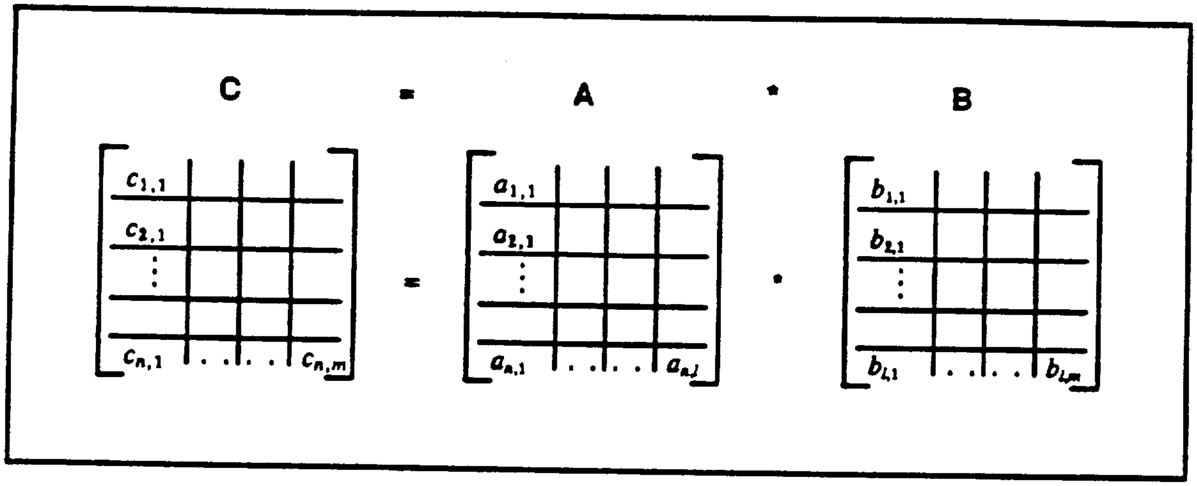


Figure 9.2.1-1 Hypermatrix Multiplication Problem

typically operate on submatrices of from 40x40 to 60x60 double precision words in size. Therefore, a submatrix size of 50x50 double precision was selected. The matrices A,B, and C are dimensioned to 400x400 double precision words. Larger sizes would have been desirable, but the execution time on a single processor, which is required to calculate speedup factors, becomes exorbitant.

The objective of this test was to examine the performance of the memory management system. The size of the problem was created so that load balancing was easy to maintain, however the problem was large enough so that only part of the data could fit in the memory (Data Pool) of a Server. Although the problem is relatively small, the volume of data used is more than twice the size that can fit into the data pool of a single processor.

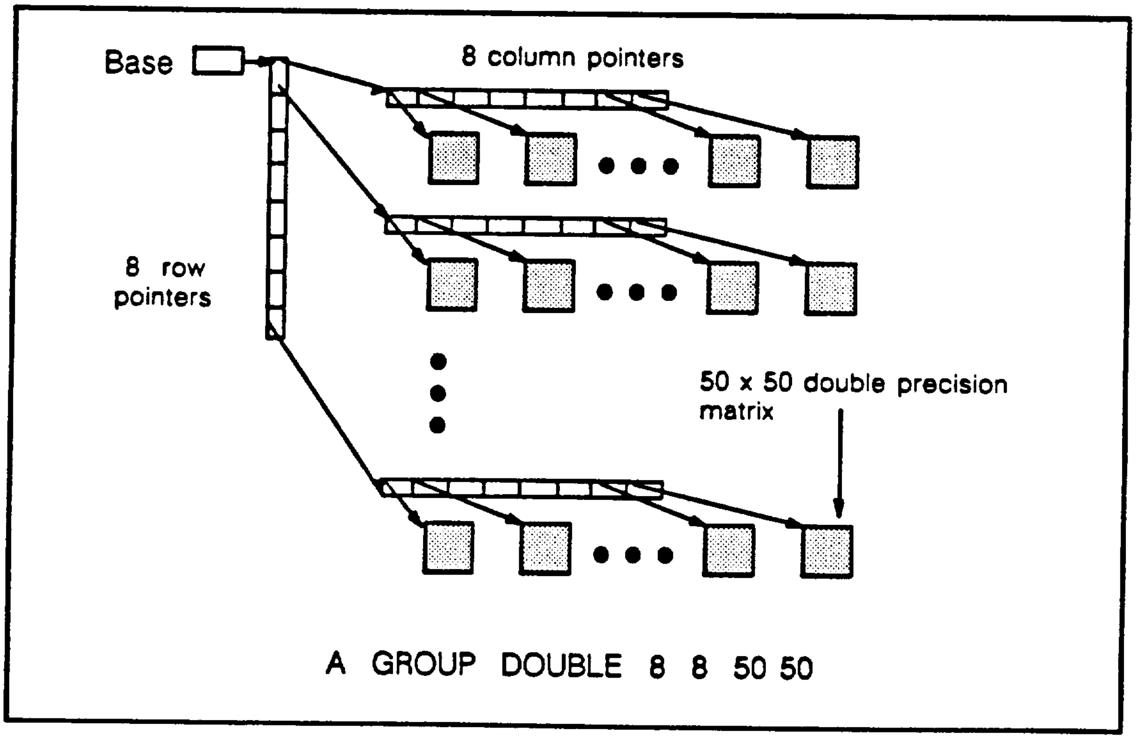
This hypermatrix problem is naturally amenable to concurrent operations. Vectorization was not considered because it is not available on the hardware we used. However, On computers with vector processors, vectorization would be most effective in the FORTRAN routine that is mapped to the High Level Language Operation.

The equation for calculating each term of a hypermatrix is identical to the term by term calculation; i.e., $c_{i,j} = \sum_{k=1}^{l} a_{i,k} * b_{k,j}$, where the multiply implies a matrix product. The C matrix was initialized prior to the execution of the triple matrix product. Although the time required to create the c's dynamically occurs in some problems, it was not considered here.

9.2.2 Programming notes

The data structure of a hypermatrix is shown in figure 9.2.2-1. The statement at the bottom of the figure is the Data Definition command that is used to generate the data structure. The hypermatrices A, B and C are hierarchical data structures. They are defined as 64 elements organized into eight hyper-rows and eight hyper-columns. Each element consists of a 50 by 50 double precision submatrix. In this problem, a fixed size data structure was used. For general applications, the data structure can contain variables in place of the numbers used. The data management system can dynamically update the size of the structure during the application problem.

The figure below shows how the data structure is organized. The base pointer points to an array of row pointers. Each entry in the row pointer, points to an array of column pointers. Each entry in the array of column pointers points to the 50x50 double precision submatrices.



```
A GROUP DOUBLE 8 8 50 50
```

Figure 9.2.2-1 Data Definition of Hypermatrices A, B and C

Hence, a reference such as "A(i,j,1,1)" will point to the first row and column entry of the submatrix element of the (i,j) block. This particular mechanism is biased towards row access of data however it does not effect the performance significantly. An alternative hypermatrix structure would be to use a one level, two dimensional array of pointers to the submatrices.

Subdividing the matrix into 64 submatrices introduces the possibility of concurrent execution in solution procedure because individual submatrices can be used simultaneously on independent processing modules. However, the one or two level pointer scheme makes a big difference in performance. The one level scheme unnecessarily inhibits concurrent operations because only one processor at a time can get at the submatrix pointers. This can result in conflicts, and hence delays when accessing an object — this is crucial if objects are created dynamically during the execution of the application problem; it is not significant if the matrices are created ahead of time because, in these circumstances, all processors could access the pointer array for "read-only."

The HLL code that performs the triple matrix product is shown in figure 9.2.2-2. Loop indices **I**, **j** and **k** are task common variables. The variables *nrow* and *ncol* were initialized to the values of eight (not shown in example). The **use definition** instruction identifies the data base definition that is used for the application. The data definition consisted of three statements defining the data structures for **A**, **B** and **C**. The example of the data statement was shown in figure 9.2.2-1. **Matmul** is an HLL operation that is mapped to a FORTRAN subroutine -- it has three parameters. The routine will add the matrix product of the first and second parameters to the third parameter. Although not shown in the HLL, the sizes of the individual matrices being passed to the Matmul routine are made available automatically by the DBMS. In the call the **c** submatrix is tagged with the write (*w*) and immediate release (*i*) data attributes. Since the **a** and **b** submatrices are not modified, they are tagged for read only (*r*). Therefore, multiple copies of the **a** and **b** submatrices may exist throughout the network.

There are three programmer defined levels of concurrency in this application. Each iteration of the the triple nested do-loop can be performed independently of any other iteration. Hence the text highlighted in bold is used to express the concurrency within the application. The granularity values are the relative values of the amount of computation

```
common i, j, k, nrow, ncol

use definition = matrix_definition

do for i = 1, nrow in parallel granularity 64
   ...
    do for j = 1, ncol in parallel granularity 8
   ...       do for k = 1, nrow in parallel granularity 1
                   matmul(a(i,k,1,1):r, b(k,j,1,1):r,  c(i,j,1,1):wi  )
             enddo
 ...    enddo
enddo
```

Figure 9.2.2-2 HLL Code for the Hypermatrix Problem

involved in performing one iteration of the parallel do-loop. The outermost loop is eight times larger than middle loop which in turn is eight time greater than the inner-most loop. The granularity of the matmul operation has the value of one unit.

The processor granularity influences the scheduling of loop iterations. If the processor granularity is set at one unit, the Client will schedule tasks at the inner-most loop level, there are 512 tasks of submatrix multiplications. If the processor granularity is set to 8 units, the concurrency is exploited at the middle loop. The client will schedule 64 tasks consisting of computing one element in the "C" matrix. If the processor granularity is set at 64 units, the concurrency is exploited at the outer loop level, the client will assign the computations needed for computing one row of the C matrix to a Server . When two or more Servers were used, the granularity was set at 8.

The High Level code shown in figure 9.2.2-2 is architecture independent. This application program and the corresponding FORTRAN subprogram for the matmul operation will execute on a distributed memory and a shared memory computer without any modification. One can easily fine tune the application to a given processor system by modifying the size of the data objects and incorporating special hardware dependent features like vector operations in the FORTRAN subprogram.

9.2.3 Results and Observations

The performance of the matrix multiplication is illustrated in figure 9.2.3-1. The data obtained from the tests are presented in table 9.2.3-1. The speedup is measured relative to the time required to solve the problem on one server. The time required to solve the problem using one Server is approximately ten percent more than the time required to solve the same problem on a single processor using a very "tight" and optimized FORTRAN program. Thus, the overhead of the run time support system is reasonable considering the ease of programming. The time shown is based on the "wall clock" — it is not the same as cpu time.

Table 9.2.2-1 Performance of the Hypermatrix Multiplication

number of processors	time (minutes)	speedup
1	60.0	1
2	32.5	1.8
4	17.9	3.4
8	10.0	6.0

Figure 9.2.3-1 shows two plots. The plot of the speedup vs. the number of processors used is shown at the top of figure and the plot of time required to solve the problem vs. the number of processors used is shown at the bottom of figure. The figure indicates that the utilization of the processors was approximately 75%. The speedup increased proportionally as more processors were added.

Generally, the factors that contribute to the disparity between the ideal speedup and the actual speedup obtained include: overhead for task scheduling and synchronization; the overhead in data communications; the capability to load balance; and, the amount of concurrency in the application problem.

The factors that influence the performance of this problem include: the size of Data Pool; and, the overhead of obtaining the data objects. The scheduling and synchronization overhead, load balancing and concurrency in this problem were not a problem as far as performance degradation was concerned. The size of the data pool restricts the number of

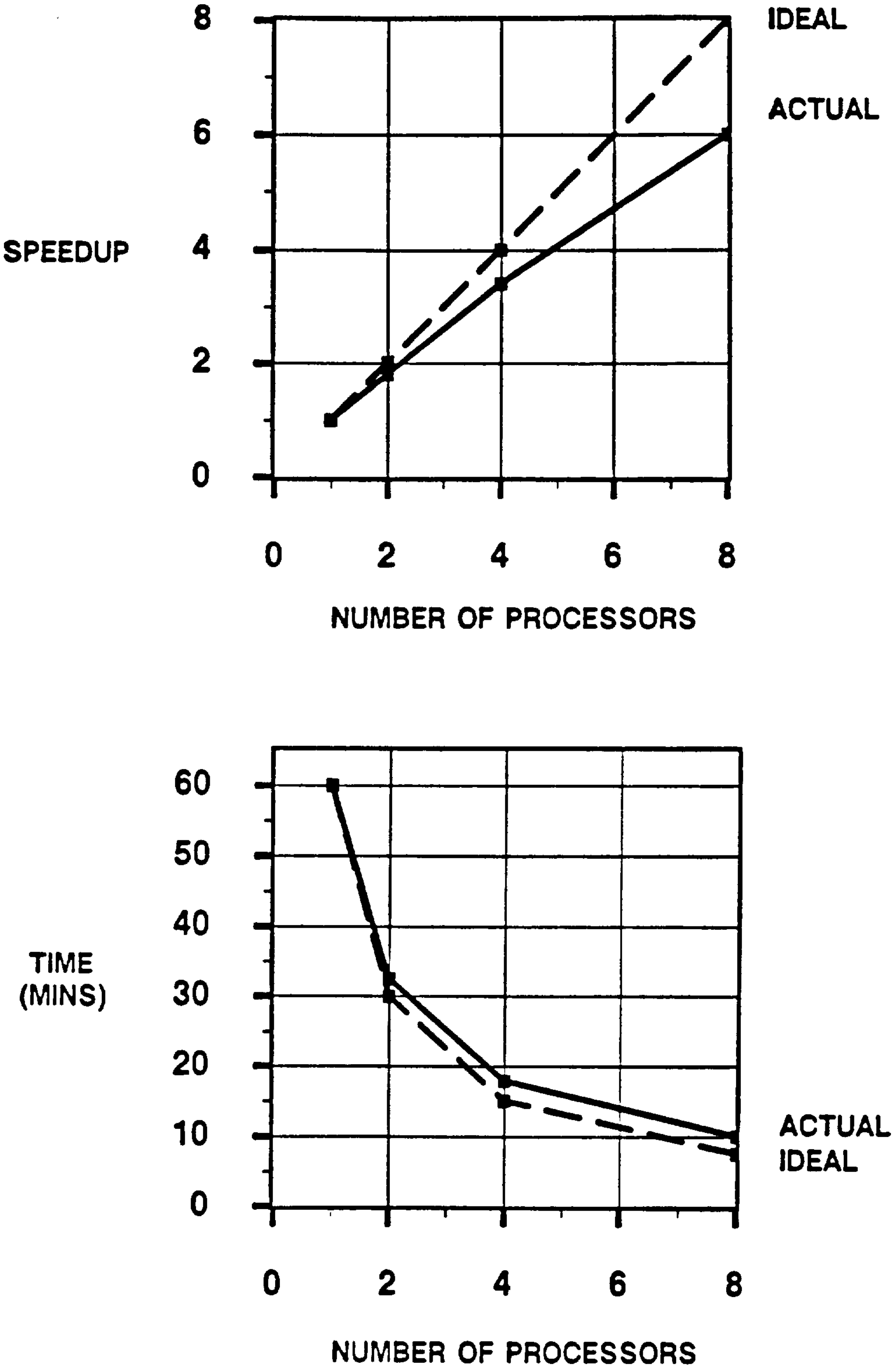


Figure 9.2.3-1 Performance for the Matrix Multiplication Problem

data objects that can be held in memory. When all of the data for the application problem cannot fit into the Data Pool, the individual data objects must be transferred between primary and secondary storage dynamically. This in turn effects the time required to get a data object because the MM has to go over the network.

When one Server is used, the time required to fetch the data objects and to perform a matrix multiplication and add initially is 14 seconds. Eventually, the average time drops to six seconds (5.8 seconds required for a matrix multiplication) because as the operations progress, the objects are already within the pool. The average time per task over the entire run is seven seconds.

As the number of Servers increase, the average time for a task increases because each Server starts with none of the objects in pool. Thus, the overhead of getting the objects from disk accounts for a much larger percentage of the time required to perform the unit of work. The overall time required to perform the matrix product decreases because of the significant gains of concurrency, as more processors are used, fewer tasks are performed per processor.

Each hypermatrix object requires six pages in pool. In order to accommodate every object required for the application, the pool should be approximately 1200 pages. Since the pool is dimensioned to 500 pages, paging occurs when one and two servers are used.

The main factor contributing to the disparity between ideal and actual performance was the communications overhead required to manipulate data objects. Tests were conducted on the token ring network to determine the effective throughput. Sending 1000 word messages showed that the net throughput averaged only 60 kilobytes/s.

During the development phase of the environment, when only the memory manager system was operational, programming the hypermatrix problem took over two weeks to implement. Since the HLL and DM components were missing, a list of tasks was generated at the start of the program and the DM was emulated in fortran routines. In the current system, the programming effort takes less than two hours! The overhead of the DM and HLL interpreter is minimal.

If the c submatrix objects exist when the matrix multiply begins, read only copies of the pointer matrix are effectively propagated on demand to all of the servers immediately. Submatrices that are initially on disk will propagate from the disk to the various processors. Since both the A and B matrices will be used for read only, their submatrices will propagate over the network to other processors on an as-needed basis. The individual c submatrices must be obtained for write-access. Hence, the amount of conflict is a function of the algorithm. The memory management system will automatically handle the conflicts. The effects of the conflicts will show up in the performance.

Assuming that the C matrix exists before the calculations begin is not realistic. In most sparse matrix applications C would be generated as the multiply progresses. Thus, both the pointer matrix and the c submatrix would need to be accessed with write privileges. This would result in the pointers being passed around the network as individual processors created new submatrices. Conflicts would occur and the solution would be somewhat slower.

9.3 APPLICATION 2: BLOCK CHOLESKY DECOMPOSITION

9.3.1 Description of the Problem

The solution of a linear system of equations is obtained using the Cholesky algorithm. The equations are generated by a finite element program. The equation is:

$$[K]\{u\} = \{P\} \qquad \text{(equation 9.3.1-1)}$$

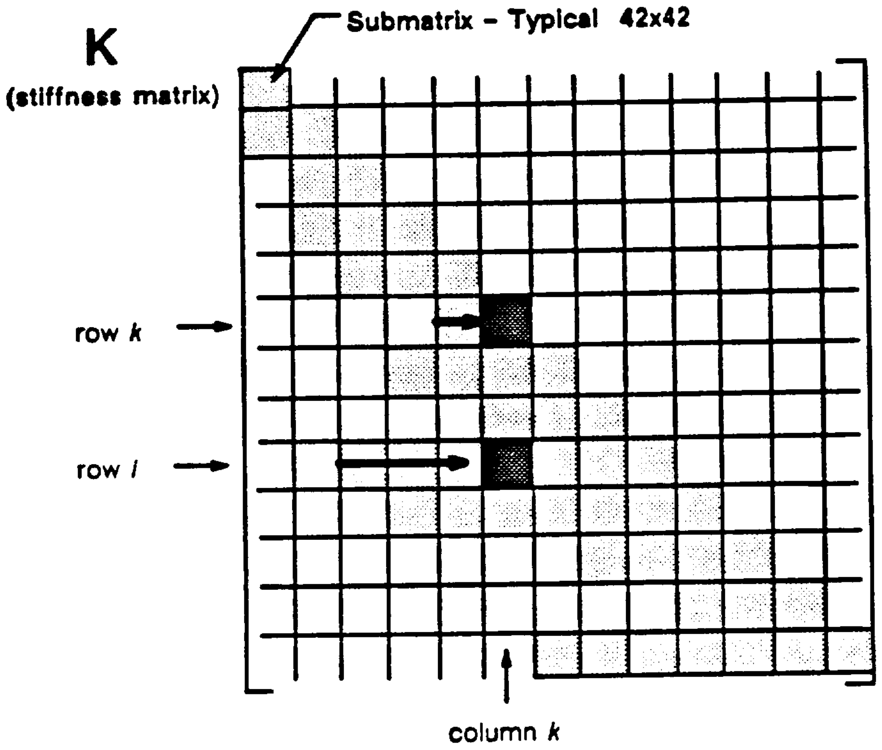
where, [K] is the global stiffness matrix, {u} is a vector of nodal displacements and {P} is the vector of applied nodal loads. In the displacement method used in the finite element procedure, equation 9.3.1-1is solved for {u}. [K] is formed by summing the stiffness contribution of each element. In typical structural analysis problems, [K] is sufficiently large and sparse to make inversion impractical. Therefore alternate techniques must be employed to compute {u}. The most common direct approach is to use the Gaussian Elimination or Cholesky decomposition. Gaussian Elimination is a much more general approach whereas Cholesky decomposition takes advantage of the symmetry in the [K] matrix. For structural analysis problems [K] is usually symmetric, and hence the Cholesky method is most often used. Other alternatives to solving the stiffness matrix include iterative methods, an example of such approach is the conjugate gradient method.

For this application problem, the cholesky decomposition is used. The basic procedure is to decompose [K] into a $[L][L]^T$ form, where [L] is a lower triangular matrix. The solution procedure incorporates a hyper–matrix approach. Hence, each hypermatrix term K_{ij} consists of a submatrix.

The equations that describe the decomposition procedure are shown in figure 9.3.1–1. They are based on earlier work by Fuchs and Schrem [32]. There are two equations shown in the figure. The first equation corresponds to operations performed for terms on the diagonal. The second equation corresponds to operations performed for off–diagonal terms. The hypermatrix or block Cholesky algorithm is very similar to a regular Cholesky decomposition. The difference lies in the types of operations that are performed; instead of multiplying and dividing single elements, one has to perform the equivalent operations at the matrix level. Elements within the hypermatrices are manipulated to perform the matrix multiplication and matrix "division" operations. In the algorithm that is illustrated in figure 9.3.1–1, the terms in the matrix are transformed columnwise. Columns are processed from left to right. Hence, for a term in a given column, all the elements to the left of the column have been transformed to their corresponding term in [L].

Referring to figure 9.3.1–1, there are two basic steps involved in the algorithm. In the first step, each term K in a given column *k* is transformed to a K^* term. The term K^*_{ik} is the original term K_{ik} minus the matrix dot product of: the terms (L) in row *i* and the transpose of the terms (L^T) in row *k*. The second step consists of computing the L_{ik} term given K^*_{ik}. For a term on the diagonal ($i = k$), a cholesky decomposition is performed at the element level on K^*_{kk}. For a term on the off diagonal ($i \neq k$), the off diagonal term L_{ik} is obtained from K^*_{il} by performing the equivalent of the operations used to do a "forward pass" on the right hand side vectors.

In the algorithm just described, the computation of terms in a given column is dependent on the operations performed on terms in the previous columns. Hence, columns must be processed sequentially. However operations within the column can be performed concurrently. The potential for concurrency exists at two levels. The first level of concurrency involves the computation of the K^* terms within a given column. The second level of concurrency occurs within the computation of the individual K^* terms. Although the off diagonal terms L_{ik} can be computed concurrently, these computations must be preceded by the



$$(1) \quad L_{kk} \; L^{T}_{kk} = K_{kk} - \sum_{j=1}^{k-1} L_{kj} \; L^{T}_{kj} = K^{*}_{kk}$$

$$(2) \quad L_{ik} \; L^{T}_{kk} = K_{ik} - \sum_{j=1}^{i-1} L_{ij} \; L^{T}_{kj} = K^{*}_{ik}$$

Figure 9.3.1-1 Block Cholesky Decomposition Equations

decomposition of the diagonal term because the computation of L_{Ik} involves the transpose of diagonal term (L^{T}_{KK}.)

9.3.2 Programming notes

The data structure used for the application problem is similar to that used in the hypermatrix problem. In this case, the submatrix size for this problem was set at 42 by 42 double precision — this was the maximum size that could be set in the program that generated the system of equations.The data structure was organized so that for any given row, the first set of contiguous zero data blocks were not stored. The data structure is shown in figure 9.3.2-1.

The main routine in the HLL program that was developed for this application is shown in figure 9.3.2-2 . The statements containing "loop I" and "cond i " are annotation for further discussion, and are not part of the program. The matrix dot product or "K* " operation is performed in a subroutine called "form_k_star". It is shown in figure 9.3.2-2. Both figures contain the constructs, highlighted in bold text, that were used to transform the application from a serial program to a parallel program. The discussion of the program begins with the explanation of the serial version. It is followed by the description of various ways to exploit the concurrency.

A double nested do-loop is used to process the elements in the matrix. The outer loop (loop 1) sets the column number to be processed. Each column in the matrix is referenced in this loop. The inner loop (loop 2) sets the row element within the column that will be trans-

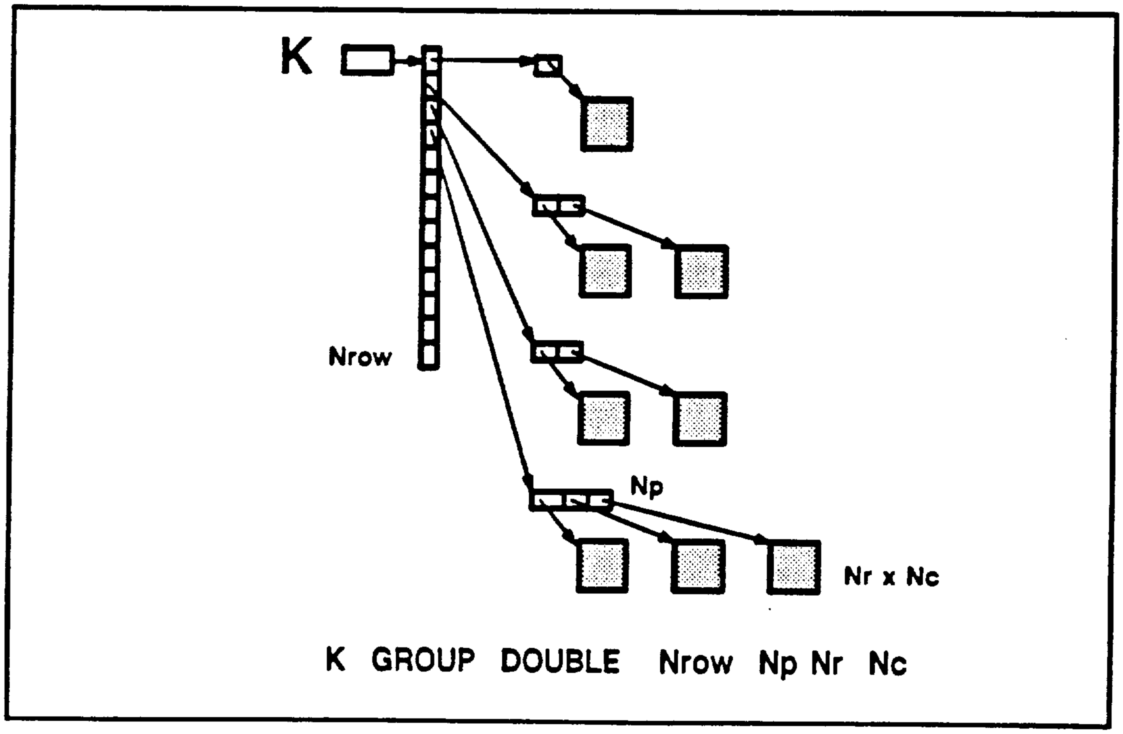


Figure 9.3.2-1 Data Structure for the Stiffness Matrix

```
SUBGRAMMAR TRIANGULATE
loop 1   DO FOR COL = 1, NEQ
              UNAVAILABLE ( HOLD:wi)
loop 2        DO FOR ROW = COL, NEQ   IN PARALLEL GRAN 100
                DEFAULT ACCESS WRITE
                IS_IT_THERE ( K (ROW,1):ri , OK)
cond 1          IF *OK THEN
                      CALL FORM_K_STAR
cond 2                IF *COMPARE(ROW,COL) THEN
                        DECOMPOSE ( K( ROW, REL_ROW_COL, 1, 1) )
                         AVAILABLE ( HOLD:ri )
                      ELSE
                        TOUCH ( HOLD:ri )
                        FINISH_OFF_DIAGONAL( K( COL, REL_COL_COL,1,1),
                                      K( ROW, REL_ROW_COL,1,1 ):wi)
                      ENDIF
                 ENDIF
              ENDDO
         ENDDO
 RETURN
 END OF SUBGRAMMAR
```

Figure 9.3.2-2 Main Routine (algorithm 1) for Cholesky Decomposition

```
SUBGRAMMAR  FORM_K_STAR

     DEFAULT ACCESS READ

     INNER_LOOP_LIMITS,

     DO FOR K_COL = LOW_TERM, UP_TERM   IN PARALLEL GRAN 10

          IF ( FIRST_ITERATION ) THEN
             CLEAR
          ENDIF

          EXISTENCE( K ( ROW, 1 ), K ( COL, 1 ), OK )

          IF *OK  THEN
              MULTIPLY_SUM  ( K ( ROW, REL_ROW_COL, 1, 1 ),
                                   K ( COL, REL_COL_COL, 1, 1 ))
          ENDIF

          IF  ( LAST_ITERATION ) THEN
             GET_RELATIVE_VALUES,
                SUBTRACT( K ( ROW, REL_ROW_COL, 1, 1 ):wi)
          ENDIF

     ENDDO

     RETURN

END OF SUBGRAMMAR
```

Figure 9.3.2-3 HLL Code for the K^* Operation

formed. The inner loop starts at the term on the diagonal and ends at the last term in the column. The HLL operation *is_it_there* checks if the term at the specified instance of row and column values in the matrix exists. If the term exists, the logical variable *OK* set to true. In the first conditional statement (cond 1), if *OK* is true, the K^* computation (matrix dot product calculation) is performed by calling the HLL subroutine *form_k_star*. After computing K^*, one more operation remains; if the term is on the diagonal, it is *decomposed*, otherwise the operation *finish_off_diagonal* is performed. Note that the finish_off_diagonal operation cannot proceed until the decompose operation is complete.

The preceding identified the concurrency and data dependencies within this application. There are two approaches that can be used to exploit the concurrency and manage the data dependencies. The first approach is to maintain the structure of the serial program that was just described by employing synchronization primitives to handle the data dependency. This approach is used in the first algorithm as shown in figure 9.3.2-2. It is a natural solution to the dependency problem. The second approach is to modify the structure of the program, thereby implicitly handling the data dependency. Two algorithms were considered for the second approach, they are described later.

In order to implement the first algorithm, the High Level Language was enhanced with features to explicitly synchronize the computation of the diagonal and off-diagonal terms. The high level operations are called **unavailable, available** and **touch.** The first two functions are analogous to semaphore operations. The **unavailable(x)** operation is used to deny access of object **x** to any Server. The **available(x)** function is used to make the object **x** available to the Servers. The **touch(x)** operation simply attempts to access the object **x**. If the object is unavailable at the time the *touch* operation is invoked, the Server will wait until another Server makes the object available.

Figure 9.3.2-2 illustrates how the new features were implemented in the program. In order to insure that the *finish_off_diagonal* operation is not performed before the *decompose* operation, the the object *hold* is made unavailable prior to beginning the inner do loop (loop 2). This loop permits K^* operations to proceed concurrently. The body of the loop also contains the conditional statement (cond 2) that is used to perform either the *decompose* or the *finish_off_diagonal* operation.

Since the time required for K^* computations vary depending on the sparcity of the matrix, It is not possible to determine when a Server will reach the conditional statement (cond 2). When a term on the diagonal is encountered, it is decomposed and the object *hold* is made available. When an off-diagonal term is encountered, the Server will attempt to touch *hold*. If the operation is successful, the server can proceed with *finish_off_diagonal*. This implies that the diagonal term has already been decomposed. If the diagonal has not been decomposed, the Server will wait at the *touch* operation. These new functions will insure that the correct results are obtained. They are easy to use and were easily implemented.

The difficulty with the above approach is that for problems in which the band is nearly uniform in size, calculation of the diagonal K^* obviously takes longer than the corresponding calculation for off diagonal terms. This almost guarantees that servers will wait for the *hold* object to be made available. Other algorithms might be more appropriate.

In the second approach, the program was restructured to implicitly enforce the data dependency. Two alternatives were considered. The algorithms are shown if figure 9.3.2-4, and are referred to as the second and third algorithm.

In the second algorithm (figure 9.3.2-4a) the K^* computation for the diagonal element is performed first. Concurrency is exploited within the K^* computation. After decomposing the diagonal term, the remainder of the terms within the column are processed concurrently in a separate parallel do-loop. There is no conflict when the *finish_off_diagonal* operation is encountered because the diagonal term is decomposed prior to scheduling the calculations for the off-diagonal terms in the column.

In the third algorithm (figure 9.3.2-4b) two parallel do-loops are used. In the first loop, the K^* computations are performed simultaneously for all the terms in the column. Furthermore, when the diagonal term is encountered, it is decomposed. In the second loop, the *finish_off_diagonal* operations are performed concurrently on all off-diagonal terms in the column. The data dependency is implicitly enforced by the synchronization of the two parallel do loops -- the second loop cannot start until the first one is completed.

All three algorithms use the same version of the *form_k_star* subroutine. In order to facilitate programming this subroutine so that concurrency could be exploited, new features were added to the HLL. In order to efficiently operate in concurrent mode, it is necessary to minimize the amount of time that a Server holds a shared data object for exclusive use. If this is not done, other Servers may have to wait for a substantial amount of time before they obtain access to the object, this in turn can degrade performance considerably. This is particularly important in the K^* operation in which the matrix dot product may be split among several Servers. Each Server has to subtract from the term in the column K_{ik}, the partial product. Hence each Server competes for access to the term K_{ik}. Performance will degrade considerably if the term is held by the Server for the duration of the partial product.

Ideally, all Servers should be able to proceed with the dot product computations and then in one operation modify the term in the column. In this way, the Servers only hold the

```
loop on columns

    calc k_star for diagonal term in column concurrently
    decompose diagonal term

    do in parallel for remainder of elements in column
        calc k_star
        finish off diagonal
    enddo

end loop on columns
```

(a) algorithm 2

```
loop on columns

    do in parallel for all elements in column
        calc_k_star (concurrency optional)
        if (diag. term) decompose diagonal term
    enddo

    do in parallel for all off-diagonal terms in column
        finish off diagonal
    enddo

end loop on columns
```

(b) algorithm 3

Figure 9.3.2-4 Alternative Algorithms for Cholesky Decomposition

object for the duration of the subtract process. This can be accomplished by performing the matrix dot product and storing the results in a temporary matrix stored in the common area. After the dot product is performed, the server will get the column element and subtract the results accumulated in the temporary matrix. This implies that the programmer knows when the first and last operations in the matrix dot product are encountered by the Server.

In order to provide the programmer with this information, the **first** and **last iteration** flags were incorporated into the HLL. The use of these flags is shown in figure 9.3.2-3. The HLL operation *Inner_loop_limits* sets the values of the variables *low_term* and *up_term.* During the execution of the program, the Client will assign to the Server, one or more iterations of this do-loop. During the first iteration, the local matrix is cleared (initialized with zeros). For each iteration, the sum of the matrix dot product is accumulated in the temporary matrix. In the final iteration, the subtract operation takes place. This procedure minimizes the amount of time that the global data object is needed by any given processor while exploiting the concurrency within the application.

9.3.3 Results and Observations

The system of equations was generated by a finite element program. This was accomplished by defining a cantilever structure consisting of 350 elements and 1129 nodes. Each node had six degrees of freedom. Twelve DOF's were constrained to be zero. A blocking factor of 42 was specified for this problem, and therefore resulted in 162 hyper-rows (equations). The average half-bandwidth was 477 nodes or 12 hypermatrices.

For this problem, the algorithms described earlier were modified slightly; the upper limit of the parallel do-loop (loop 2 of figure 9.3.2-2) was changed so that the loop variable *row* stopped at the boundary of the bandwidth instead of at the end of the column.

This application problem had a relatively small bandwidth and consequently this had important ramifications on the load balancing aspect of the scheduling and synchronization of tasks. Figure 9.3.3-1 shows that the number of hyper-elements associated with a given row decreases as one proceeds down a column. Hence the amount of calculations involved in the *k_star* operation decreases as terms further away from the diagonal are processed. This has a profound effect on the performance of the system. All three algorithms were affected.

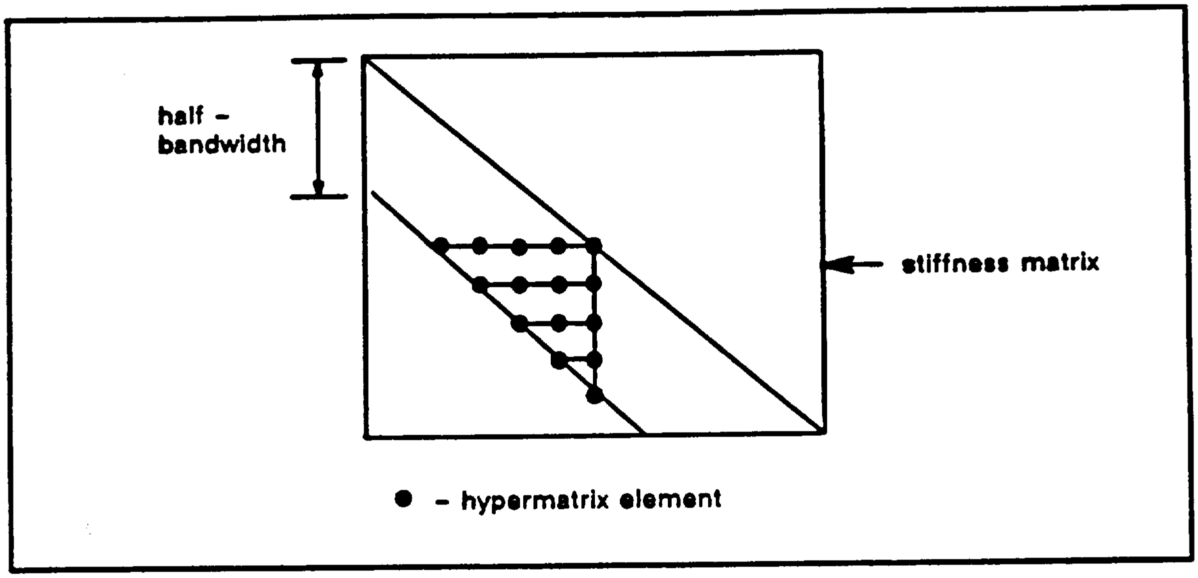


Figure 9.3.3-1 Terms Involved for the K* Operation for a Given Column

In the first algorithm, K* computations for the elements in a given column were performed concurrently. When multiple Servers were used, those servers that processed off-diagonal terms usually had to wait for the first Server to decompose the term on the diagonal. Some servers had to wait a considerable amount of time because they were assigned tasks with a very small amount of computation.

The solution to the problem in the first algorithm was addressed in the second algorithm (figure 9.3.2-4a). In this approach, the program was designed to process the term on the diagonal term first before proceeding with the other terms. In this approach, the concurrency in the K* operation was exploited. The relative granularity was set so that each Server would get at least one hypermatrix operation. This granularity was too small and furthermore the overhead of getting the data only exacerbated the situation.

The best results were obtained with the third algorithm (see figure 9.3.2-4b.). However, problems with load balancing and data communications overhead were still encountered. This algorithm worked slightly better than the other because a Server that was assigned a term with few operations could get some more work without having to wait. The results obtained for this application problem are shown in table 9.3.3-1 below. The data is shown graphically in figure 9.3.3-2.

Table 9.3.3-1 Performance of the Cholesky Decomposition Problem

number of processors	time (hours)	speedup
1	7.33	1
2	4.22	1.7
3	3.22	2.3
4	2.71	2.7
6	2.30	3.2
8	2.30	3.2

The speedup curve shows that there was no point in using more than six processors because we couldn't take advantage of them. The results obtained are consistent with the observations of Ortega [62]. It has been shown that the Cholesky decomposition of stiffness matrices that have a relatively small bandwidth perform poorly on multiprocessors. Despite the poor speedup curve, the results are encouraging. Considering the facts that the data base was approximately 35 MB, the processor memories were 4 MB, the network throughput was 60 kilobits/s, only one disk manager was used, and the system of equations had a small bandwidth, the system was used to reduce a 7.33 hour problem by five hours using an additional five workstations.

In an experiment to examine the amount of time spent getting data objects, the computation involved in the actual matrix multiplication was suppressed. It was found that for eight processors, a wall clock time of 80 minutes was spent just moving data objects across the network!

In an effort to maintain data locality, the scheduling system in the Client was modified to a static scheduling scheme. That is In this scheme, a given server is responsible for predefined set of iterations. The set of iterations corresponds to the server number and every iteration whose stride is the number of servers from the initial iteration. Hence, if there are four servers, the first server is responsible for the set {1, 5, 9, 13 ... }, while the fourth server will be assigned {4, 8, 12, ...}. This did not improve the performance because of the load balancing problem again. Those servers that finished quickly still had to wait because noth-

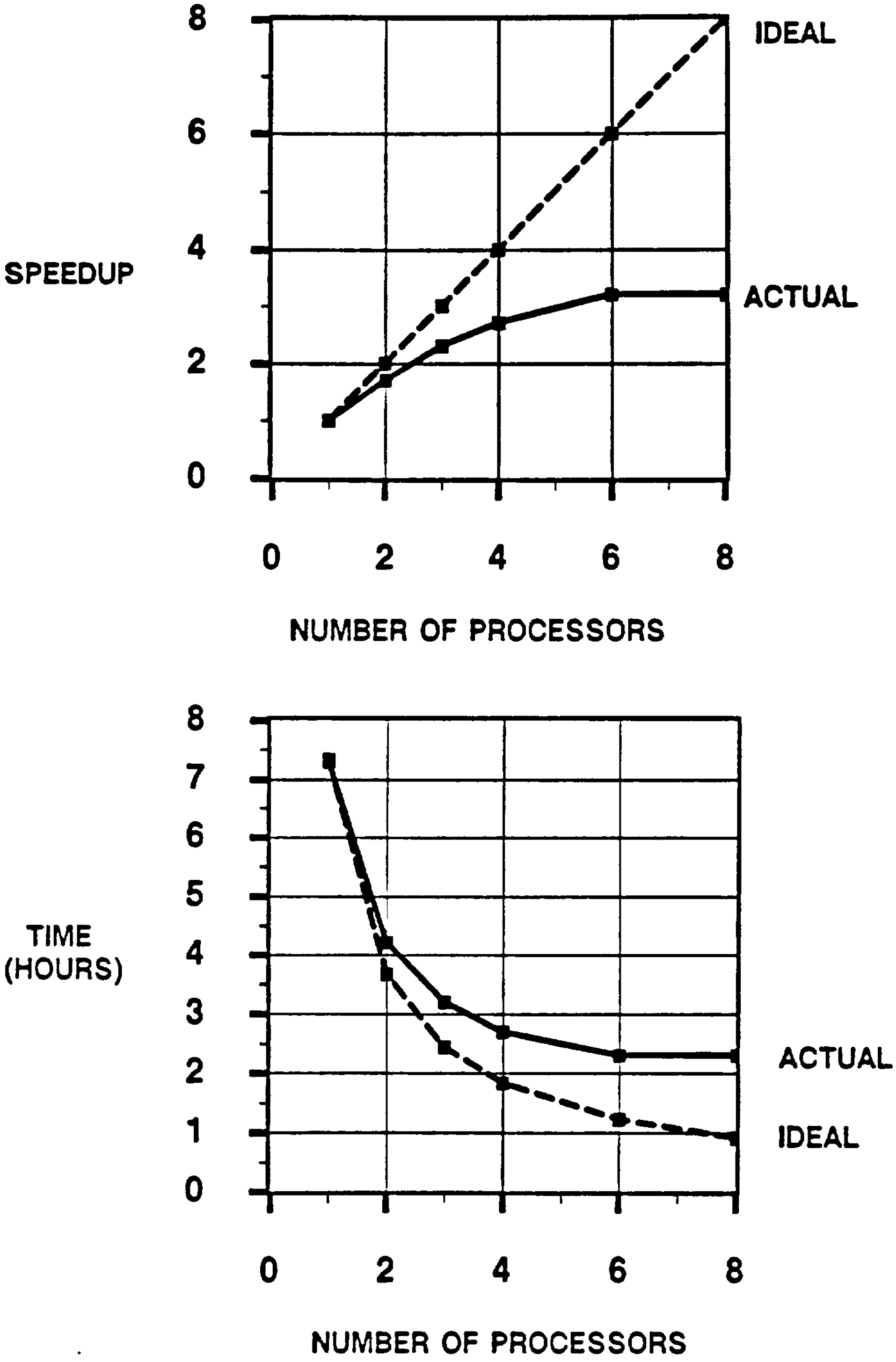


Figure 9.3.3-1 Performance of the Cholesky Decomposition Application

ing more was assigned to them. For example if there are 12 terms and 8 servers, server 8 only has iteration 8 to do. while server 1 has to do iteration 1 and 9. Iteration 1 takes a long time compared to iteration 8. Thus server 8 has to wait until the other servers finishes their iterations.

This scheduling algorithm enforces data locality because a given server will not have to switch row data with any other server. This approach leads to less data communication for the entire network than the first algorithm, but does not guarantee improved overall performance or utilization. The trade-off between data locality and load balancing even out.

In conclusion, this application problem demonstrated the flexibility of the HLL approach: programs are easily developed and modified; new features can be implemented without much difficulty. Programs can be developed in an architecture independent form. Despite the slow communications in the network, the results obtained are encouraging. Clearly, the performance is heavily influenced by the slow communications network used in the prototype system.

CHAPTER 10

CONCLUSIONS AND RECOMMENDATIONS

This chapter is divided into two parts. The first part is a summary of the observations and conclusions that were made based on the development and implementation of the conceptual design of the environment, and the results obtained from the application programs. The second part identifies the potential for future work and the enhancements for the programming environment, particularly those aspects related to improving the performance of the run-time system and porting the environment to other MIMD computers.

10.1 SUMMARY OF OBSERVATIONS AND CONCLUSIONS

This report identifies the need to address the issues related to developing software for compute-intensive engineering applications. Engineers invest a significant amount of time and money developing large sophisticated systems for such applications. Rapid changes are taking place relative to the computing technology used by engineers. The latest type of hardware technology that is commercially available today is known as Multiple Instruction stream Multiple Data stream (MIMD) computers. MIMD computers have the potential to allow engineers to investigate and analyze very complex and compute-intensive problems — problems significantly larger than those encountered in the past.

MIMD computers have existed for approximately twenty years. However, it is only recently that engineers have employed them for production work. MIMD computers were not used in the past because they were expensive and very difficult to program. Advances in the microelectronics industry have made parallel processing hardware much more affordable; programming environments will make parallel processing software easier to develop.

Programming languages and environments evolved as the hardware technology changed. Programming Languages are suited to exploiting various features of the hardware while environments facilitate the development of application programs. Unlike the FORTRAN model of computing, there is no standard programming language that will map easily to the variety of hardware that is available. Hence, new software development tools are needed to help engineers develop large scale architecture-independent software for MIMD computers.

This research has shown that it is possible to develop an architecture-independent model for parallel computing that facilitates the development of large scale engineering software systems on MIMD computers. The development of such systems is necessary if progress is to be made in solving the large engineering problems of the future.

Architecture-independent software is important because new machines are constantly being developed. Some of them will succeed commercially and others will fail. A case in point was the recent failure of both the ETA-10 and FLEX computer companies. Developing large scale engineering software for a specific computer can be very costly, especially if that computer fails commercially. Architecture-independent software facilitates software portability so that it is possible to take advantage of a variety of computers. It also improves the longevity and utility of the software product and provides some form of insurance for the investment of time and money that goes into developing the software.

This report describes the design and implementation of a prototype software environment that facilitates developing engineering software on a variety of MIMD computers. The conceptual design was based on the configuration of several hybrid, shared, and distributed memory computers. A prototype system was implemented on a distributed memory system for demonstration purposes.

Two application programs were developed using the software environment to illustrate the capabilities of the system. The application problems were chosen from parts of the finite element problem. The finite element method was chosen as a focus problem to provide some insight into the types of problems that an engineer would typically encounter when developing large scale engineering software. It was shown that, architecture-independent engineering application programs can be developed once some of the conceptual problems are resolved. Algorithms can be easily modified for experimental purposes. The performance of the application problems was reasonable considering the type of hardware that was used (engineering workstations on a token ring network).

The development of software applications is facilitated by alleviating many of the programming details related to: inter-processor communications; the scheduling and synchronization of tasks within the application program; data base management; and memory management. The software environment allows engineers to focus on developing their application problems rather than on the programming details associated with message passing, dynamic memory allocation, and data base management.

Unlike some of the common techniques used for shared memory machines, the software environment does not perform a data dependency analysis of the program in order to schedule and synchronize the instructions for the processors. Such a technique is infeasible for loosely coupled systems (distributed memory computers).

However, the software environment makes the programming of distributed memory systems similar to that of shared memory systems. This is accomplished by using a virtual machine concept that is mapped to the hardware provided by the target machine. The engineer sees the same virtual machine regardless of the underlying hardware. Application programs for the virtual machine are developed using a language layer concept.

The use of the language layer concept combined with the relative granularity mechanism permits the programmer to define sophisticated data structures and express multiple levels of concurrency within the application program. The run-time system automatically determines how to exploit the concurrency within the application based on the type of hardware that is used.

The High Level Language (HLL) is architecture-independent and flexible. It can be readily enhanced with new features. The HLL provides the engineer with a variety of control structures to develop a procedural-type program. The HLL program is used to define tasks that will operate on data objects. The scheduling and synchronization of the tasks is managed by the run-time system.

The Data Definition Language is also architecture-independent and very easy to use. It permits the engineer to develop both hierarchical and relational type data structures. The data structures and data objects can be defined independently of the application problem. During the execution of a program the data objects referenced in the HLL program are automatically made available in the memory of the processor on demand. The run-time system will enforce data coherence and will manage the communications necessary for getting the data object into appropriate memory module.

The actions in the HLL program are supported by FORTRAN subprograms. These subprograms can be fine tuned to the target hardware in order to exploit the features of the target machine. For example, if vector processors are used, the subprograms can be written to take advantage of vectorization within the application.

The conceptual model as implemented on the token ring network is biased towards exploiting concurrency at medium and coarse levels within the HLL application program because of the performance of the communication network and the processor granularity of the hardware used. The prototype system currently relies on the target machine to exploit the fine grained concurrency.

The data management and memory management components maintain data coherence at the data object level. Hence, only one processor can modify a data object at any given time. A virtual memory system that incorporates a paged-segmentation approach is supported by the environment. Segmentation facilitates data coherence while pages are an ideal unit for managing data and communication purposes.

The virtual machine concept manifests itself as three types of programs, namely the *Client*, *Server* and *Disk Manager.* The user only interacts with one program — the *Client.* This program also manages the scheduling and synchronization of tasks in the application program. The *Server* programs actually perform the tasks. The *Disk Manager* program provides disk file management. All three of these programs are necessarily multi-tasked; the modules within them have been developed so that this concept is easily mapped to a variety of hardware systems.

Two application problems were used to demonstrate that developing programs using the environment is quite easy. In the matrix multiplication problem, an engineer was able to save himself two weeks of work by using the features of the environment instead of developing the same application using just the low level programming facilities. This application problem demonstrated the shortcomings of programming a distributed memory computer system linked by an ordinary(1 megabyte/s) token ring network; the communications between processors is too slow for production work.

The Cholesky Decomposition problem demonstrated that the prototype system was capable of handling "out of core problems" -- typical of real engineering problems. It also showed how new algorithms can be implemented and that new features can readily be added to the High Level Language. The speedup results indicate that the relatively small bandwidth of the stiffness matrix contributes to poor load balancing when using the cholesky decomposition algorithm.

10.2 RECOMMENDATIONS FOR FUTURE WORK

The main emphasis of this work was to develop a conceptual model of the software environment and to implement a prototype. Having demonstrated that the concept is operational, it is necessary to spend some time examining the performance of various aspects of the system. Hence, it is necessary to develop more applications and tests that will provide insight to how this system really behaves.

The experience gained from the first two application problems has led to the following factors that should be considered for implementation:

1. the specification of relative granularity in terms of variable names or mathematical expressions;
2. the ability to use different scheduling strategies for different parallel do–loops;
3. a scheduling technique where the *Servers* take part of the work when they encounter a parallel stream. In this way they do not have to wait for the *Client* to respond;
4. a control system incorporated in the *Client* program whereby the performance of the processors is monitored. This information can be used to help schedule tasks to processors that work in a multiprogrammed environment by dynamically changing the processor granularity values.

There are other factors that effect the performance of the system and application programs that must be considered. They include:

1. *distribution of data bases.* How does this effect access to data objects that are striped?
2. *data pool size.* How many objects can be maintained in memory? Should the data pool be segmented in shared memory machines?

3. *page size.* What is the ideal size for communications and object management?

4. *data granularity.* How does this influence data coherence and the amount of concurrency that can be exploited in the application?

5. *scheduling algorithm.* What is the overhead? How does it effect the load balancing of tasks in the application program? What is the best strategy for a given application? How well can data locality be enforced by the algorithm?

Further experience with different problems will undoubtedly lead to the development of features that facilitate parallel programming. The experience should provide greater insight to parallel programming.

In order to perform production-type work on distributed memory computers, it is necessary to use systems that have much faster interprocessor communications than that of a general purpose token ring network. The alternative systems include the Ncube, and Transputer based systems. The specifications of the Ncube indicate that the system is capable of actual data transfer rates of up to 2 MB/s. This is a significant improvement over the token ring network that was used in the prototype. The throughput in the token ring network averaged 60 KB/s. The communication facilities in the Transputer offer improved performance compared to the token ring network. The actual speeds were not available to the author at this writing. The beauty of Transputer based systems is that they may represent the first "supercomputer on a desktop."

To demonstrate that the application programs are truly architecture-independent, it is necessary to port the system to a shared memory machine. The main emphasis of this work will be on modifying the memory manager and possibly the disk manager. All the other components remain the same. The implementation of the system on shared memory machine can be done in two ways: (1) segment the global memory so that each processor has its own data pool, or (2) use one memory manager and one global data pool. Both approaches have their limitations, it is not clear which is the better of the two.

With the advent of object oriented programming and recent trends in software environments, it is possible that the concept developed herein can be considered as a underlying support mechanism to perform operations on objects identified at higher level of data abstraction. The procedural component of a given data object can be programmed using the HLL.

Finally, although the finite element method was chosen as a the focus problem, neither the conceptual model nor its implementation is restricted to solving finite element problems only. Other application problems such as the boundary integral or finite difference method can be considered.

THIS PAGE INTENTIONALLY LEFT BLANK

REFERENCES

[1] Adams, L. M. and R. Voigt, "A Methodology For Exploiting Parallelism in the Finite Element Process," *High Speed Computing*. J.S. Kowalik (ed). NATO ASI Series, Vol 7, Springer-Verlag 1983.

[2] Allik, H., et al, "Implementation of Finite Element Methods on the Butterfly Multiprocessor," *Proceedings of the 1985 ASME International Computers in Engineering Conference and Exhibition*, Volume III, pp 393-400, American Society of Mechanical Engineers, New York, August 1985.

[3] Allik H., and S. Moore, "Finite Element Software on a Multiprocessor," *Electronic Computation. Proceedings of the 9th Conference (ASCE)*, 1986.

[4] Alliant Computer Systems Corporation,"FX/Series Architecture Manual," *Part Number 300-00001-B*, Alliant Computer systems Corporation, 1985.

[5] Almasi, G.S., "Overview of parallel processing," *Parallel Computing*, Vol 2, 1985, pp 191-203.

[6] Apollo Computer Corp., "Concurrent Programming support," Part No. 010233, June 1987.

[7] Backus, J., "Can Programming Be Liberated from the Von Neumann Style? A Functional Style and it Algebra of Problems," *Comm. ACM* , Vol. 21, No. 8, Aug 1978, pp. 613-641.

[8] Balance Technical Summary, "*MAN-0110-00 Sequent Computer Systems*, Inc.," Nov. 1986.

[9] Berger Ph., D. Comte and Ch. Fraboul, "MIMD Supercomputers for numerical applications," *Proceedings of the International Workshop on Supercomputers in Theoretical and Experimental Science*, Ed. Devreese, J.T. and Camp P.V., Plenum Press, New York, 1985, pp 115-142.

[10] Berger, Ph., P. Brouaye, and J.C. Syre, "A mesh coloring method for efficient MIMD processing in finite element problems," *Proceedings of the Int. Conf. on Parallel Processing*, 1982.

PRECEDING PAGE BLANK NOT FILMED

[11] Berkling, K. J., "A Computing Machine Based On Tree Structure," *IEEE Trans.Computers* Vol C–20, No. 4, Apr. 1971, pp 404–418.

[12] Berry, M. and R. Plemmons, "Algorithms and Experiments for Structural Mechanics on High Performance Architectures," *CSRD Report No. 602*, University of Illinois, September 1986.

[13] Bhuyan, L.N., "Interconnection Networks for Parallel and Distributed Processing," *IEEE Computer*, Vol. 20, No. 6, June 1987, pp 9-12.

[14] Carey, G.F., "Parallelism in Finite Element Modelling," Communications in *Applied Numerical Methods*, Vol 2, No. 3, May–June 1986, pp 281-287.

[15] Carey, G.F. and E. Barragy, "Finite Element Analysis Using Advanced Processors," *Proceeding of the Fall Joint Computer Conference*, Texas, Nov. 1986, pp 535-539.

[16] Cowther, W., et al, "Performance Measurements on a 128-Node Butterfly Parallel Processor," *Intl. Conf. on Parallel Processing*, Aug. 1985, pp 531-540.

[17] Cytron, R., D.J. Kuck and A.V. Veidenbaum, "The effect of Restructuring Compilers on Program Performance for High-Speed Computers," *Vector and Parallel Processors in Computational Science II*, Oxford, Aug. 1984.

[18] Daley, R.C. and J.B. Dennis, "Virtual Memory, Processes, and Sharing in MULTICS," *Communications of the ACM*, Vol 11, No. 5, May 1968, pp 306-312.

[19] Deitel, H.M., "An Introduction to Operating Systems," Addison-Wesley, 1984.

[20] Denning, P.J., "Virtual Memory," *ACM Computing Surveys*, Sept. 1970, Vol 2, No. 3, pp 153-190.

[21] Dennis, J.B., "First Version of a Data Flow Procedure Language," *Lecture Notes in Computer Science*, Vol 19, Springer-Verlag, Berlin, 1974, pp 362-376.

[22] Dijkstra, E.W., "Solution to a Problem in Concurrent Programming," *Communications of the ACM*, Sept. 1965.

[23] Dodds, R.H. Jr., L.A. Lopez and D.A. Pecknold, "Numerical and Software Requirements for General Nonlinear Finite Element Analysis," *Technical Report – Structural Research Series No.454*, UILU-ENG-78-2020 UIUC, Dept of Civil Engineering, Sept. 1978.

[24] Dongarra, J. and D. Sorensen, "SCHEDULE: Tools for Developing and Analyzing Parallel Fortran Programs," *Tech. Memo. 86*, Argonne National Lab., November 1986.

[25] Duller, A.W.G. and D.J. Paddon, "Processor Arrays and the Finite Element Method," *Parallel Computing 83, Proceedings of the International Conference on Parallel Computing*. Berlin, Sept. 1983.

[26] Edler, J., Gotlieb, A. and Lipkis, J., "Operating System Considerations for Large-Scale MIMD Machines," *Proceedings of the 1985 ASME International Computers in Engineering Conference and Exhibition*, Volume III, pp 199-208, American Society of Mechanical Engineers, New York, August 1985.

[27] "ETA10 System Overview," *PUB-1006* ETA Systems Inc, Feb 1986.

[28] Farhat, C., E. Wilson and G. Powell, "Solution of Finite Element Systems on Concurrent Processing Computers," *Engineering with Computers*, Vol 2, 1987, pp 157- 165.

[29] Feng, T., "Some Characteristics of Associative/Parallel Processing," *Proceedings of the 1972 Sagamore Computer Conference*, Syracuse Univ., 1972, pp 5-16.

[30] Flynn, W. J., "Some Computer Organizations and their Effectiveness," *IEEE Trans. on Computers*, Vol L-21, No. 9, Sept. 1972, pp 948-960.

[31] Fredrickson, P.O. et al, "Synchronization and Control of parallel algorithms," *Parallel Computing*, Vol 2, No. 3, 1985, pp 255-264.

[32] Fuchs, J. R., and E. Schrem, "Hypermatrix Solution of Large Sets of Symmetric Positive-Definite Linear Equations," *Computer Methods in Applied Mechanics and Engineering*, Vol 1, 1972, pp 197-216.

[33] Gajski, D.D. and J-K Peir, "Essential Issues in Multiprocessor Systems," *IEEE Computer* Vol 18, No 6, June 1985, pp 9-27.

[34] Gajski, D.D. et al "CEDAR - A Large Scale Multiprocessor," *Proceedings of the International Conference on Parallel Processing*, Belaire, MI. 1983.

[35] Gustafson, J.L, Montry, G.R., and Benner, R.E., "Development of Parallel Methods For a 1024-Processor Hypercube," *Sandia National Laboratories*, March 1988.

[36] Guzzi, M.D., "Multitasking Runtime Systems for the Cedar Multiprocessor," *CSRD – University of Illinois, Report No. 604,* July 1986.

[37] Haberland, A.N., An introduction to Operating Systems Design, The SRA Computer Science Series, 1976.

[38] Handler, W., "The Impact of Classification Schemes on Computer Architecture," *Proceedings of the 1977 Intl. Conf. on Parallel Processing,* pp 7–15.

[39] Hockney, R.W., "MIMD Computing in the USA – 1984," *Parallel Computing,* Vol 2, 1985, pp 119–136.

[40] Hockney, R. and C. Jesshope, "Parallel Computing," Adam Hilger Ltd., Bristol England, 1981.

[41] Hossfield, F., "Nonlinear Dynamics: A Challenge on High-Speed Computation," *Parallel Computing 83, Proceedings of the International Conference on Parallel Computing,* Berlin, Sept. 1983.

[42] Hwang, K., "Multiprocessor Supercomputers for Scientific/Engineering Applications," *IEEE Computer* Vol 18, No 6.

[43] Jeffcoat, C.E. and D.D. Wilmarth, "Attached Scientific Computers in Structural Analysis," Electronic Computation, *Proceedings of the 9th Conference (ASCE),* 1986, pp 668–679.

[44] Jordan, H.F., M.S. Benten, and N.S. Arenstorf, "Force User's Manual," Dept. of Computer and Electrical Engineering, University of Colorado, October 1986.

[45] Karp, A. H., "Programming for Parallelism," *IEEE Computer,* Vol 20, No. 5, pp 43–57, May 1987.

[46] Kogge, P.M., "Function-based computing and parallelism: a review," *Parallel Computing* Vol 2, 1985.

[47] Knuth, D.E., "The Art of Computer Programming – Volume 1: Fundamental Algorithms," Addison-Wesley, 1968.

[48] Kuck, D.J. et al, "Parallel Supercomputing Today and the Cedar Approach," *Science,* Vol 231, Feb 28 1986, pp 967–974.

[49] Kutti, S., "Taxonomy of parallel processing and definitions," *Parallel Computing*, Vol 2, 1985, pp 353-359.

[50] Larson J.L., "An Introduction to Multitasking on the Cray X-MP-2 Multiprocessor," *IEEE Computer*, July 1984, pp 62-69.

[51] Li, K. and P. Hudak, "Memory Coherence In Shared Virtual Memory Systems," *Proceedings of the 5th Annual Symposium on Distributed Computing*, ACM, Aug. 1986, pp 229-239.

[52] Lopez, L.A., "Advanced System Architecture for Parallel Computing," *Nasa-Langley, Proposal*, Aug. 1986.

[53] Lopez, L.A., "POLO—Problem Oriented Language Organizer," *International Journal of Computers and Structures*, Vol. 2, No. 4, 1974.

[54] Lopez, L.A., "FINITE: An Approach to Structural Mechanics Systems," *International Journal of Numerical Methods in Engineering*, Vol. 11, No. 5, 1977.

[55] Lusk, E.L., and R.A. Overbeek, "Use of Monitors in Fortran: A Tutorial on the Barrier, Self-Scheduling DO-loops and Askfor Monitors," *Argonne National Lab Workshop on Parallel Processing*, Dec. 1986.

[56] Lusk, E.L., R.A. Overbeek and R. Stevens, "Use of Monitors in Fortran: An Update," *Argonne National Lab.*

[57] Lykos, P., "Working document on Future Computer Systems Needs for Large Scale Computations," *NASA-Ames and IIT*, 1978.

[58] Lyzenga, G.A., Raefsky, A. and Hagar, G.H., "Finite Elements and the Method of Conjugate Gradients on a Concurrent Processor," *Proceedings of the 1985 ASME International Computers in Engineering Conference and Exhibition*, Volume III, pp 401-405, American Society of Mechanical Engineers, New York, August 1985.

[59] McGrath, R.E. and P. Emrath, "Using Memory in the Cedar System," *Proceeding of the International Conference on Supercomputing*, Athens Greece, June 1987.

[60] McGraw, J. and T.S. Axelrod, "Exploiting Multiprocessors: Issues and Options," *UCRL-91734 preprint*, Lawrence Livermore National Laboratory, Oct., 1984.

[61] Norrie, C., "Supercomputer for Superproblems: An Arch. Intro," *IEEE Computer* Vol 17, No. 3, pp 62-75 Mar. 1984.

[62] Ortega, J. "Parallel Linear Equation Solvers for Finite Element Computations," *NASA Workshop on Computational Structural Mechanics 1987* November 18-20 1987, Nancy P. Sykes - Editor, pp 171-202

[63] Osterhaug, A., "Guide to Parallel Programming on Sequent Computer Systems," *Sequent Computer Systems, Inc.* 1986.

[64] Padua, D.A., V.A. Guarna Jr. and D.H. Lawrie., "Supercomputing Environments," *CSRD Report No. 673,* Center for Supercomputing Research and Development, University of Illinois, Urbana, IL., June 1987.

[65] Papamarcos, M. S. and J. H. Patel, "A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories," *Proceedings of the 11th Annual International Symposium on Computer Architecture,* IEEE, 1984, pp 348-354.

[66] "Parallel Processing - Advancing Technologies," Datapro Research Corp., Vol 1, June 1986.

[67] Pfister, G.F., et al, "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture," *Proceedings of the 1985 Intl. Conf. on Parallel Processing,* pp 764-771.

[68] Polychronopoulos, C.D. and D.J. Kuck, "Guided SelfScheduling: A Practical Scheduling Scheme for Parallel Supercompurters," *IEEE Transactions on Computers,* Vol C-36, No. 12, December 1987, pp 1425-1439.

[69] Pratt, T.W. et al, "The FEM-2 Design Method," *International Conference on Parallel Processing* IEEE, 1983, pp 132-134.

[70] Pratt, T.W., "PISCES: An Environment for Parallel Scientific Computation," *NASA-ICASE, Rep. No. 85-12.*

[71] Ratlner, J., "Concurrent Processing: A New Direction in Scientific Computing," *AFIPS Conference Proceedings,* 1985, pp 159-166.

[72] Rehak, D.R. et al, "Evaluation of Finite Element System Architectures," *Advances and Trends in Structures and Dynamics*, Ed. by A.K. Noor and R.J. Hayduk, pp 17-30, Pergamon Press Ltd., 1985.

[73] Rogers, M.W., "Ada as a Scientific Language," *Scientific Ada - Ada Companion Series*, Ed. by B. Ford, J. Kok and M.W. Rogers, Cambridge University Press, 1986, pp 15-27.

[74] Scientific Supercomputer Subcommittee "Software for Supercomputers," *IEEE Computer* Vol 21, No 12.

[75] Smith, A.J., "Cache Memories," *ACM Computing Surveys*, Sept. 1982, Vol 14, No. 3, pp 473-530.

[76] Snyder, L., "Type Archictectures, Shared Memory and the Corollary of Modest Potential" *Department of Computer Science, FR-35* University of Washington. Seattle, WA, TR 86-03-04.

[77] Stone, Harold S., High Performance Computer Architecture, Addison Wesley, 1987.

[78] Swanson, J.A., G.R. Cameron, and J.C. Haberland, "Adapting the Ansys Finite-Element Analysis Program to an Attached Processor," *IEEE Computer*, Vol 15, No. 6, June 1982, pp 85-91.

[79] Sykes, Nancy P., Proceedings of the Workshop on Computational Structural Mechanics, NASA Langley, August 1987.

[80] Teng, P. S. and K. Hwang, "Parallel Preprocessing and Postprocessing in Finite Element Analysis on a Multiprocessor Computer," *Proceedings of the 1986 Fall Joint Computer Conference*, Texas, Nov. 1986, pp 307-314.

[81] "The Flex/32 Multicomputer System Overview," *Doc. No. 030-000-002*, 2nd edition, Flexible Computer Corporation, Mar. 1986.

[82] Treleven, P.C., D.R. Brownbridge, and R.P. Hopkins, "Data-Driven and Demand-Driven Computer Architecture," *Computing Surveys* Vol. 14, No. 1, Mar. 1982, pp 93-143.

[83] Tucker, S.G., "The IBM 3090 System: An Overview," *IBM Systems Journal*, Vol. 25, No. 1, 1986, pp 4-19.

[84] Wallis, P.J.L., "Fortran to Ada - an Example of HighLevel Language Conversion," *Scientific Ada - Ada Companion Series*, Ed. by B. Ford, J. Kok and M.W. Rogers, Cambridge University Press, 1986. pp 28-39.

[85] Wilmarth, D.D. and C.E. Jeffcoat, "Equation Solving on a Parallel Computer," Electronic Computation, *Proceedings of the 9th Conference (ASCE)*, 1986, pp 692-700.

[86] Yew, P.C., "Architecture of the Cedar Parallel Supercomputer," *CSRD Report 609*, Center for Supercomputing Research and Development, University of Illinois, Urbana IL., 1985.