EFFICIENT LOAD BALANCING STRATEGIES ON A NETWORK OF COMPUTERS: A CASE STUDY WITH TWO SCIENTIFIC COMPUTING PROBLEMS

by

Shony Abraham

A dissertation submitted in partial fulfillment of the
requirements for the degree of

Master of Science

Department of Electrical and Computer Engineering
Faculty of Graduate Studies
University of Manitoba

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

*****

COPYRIGHT PERMISSION

Efficient Load Balancing Strategies on a Network of Computers: A Case Study with Two Scientific Computing Problems

BY

Shony Abraham

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University of Manitoba in partial fulfillment of the requirement of the degree

Of

Master of Science

Abstract

Parallel computing has been recently taken over by distributed computing technologies especially in academia due to the cheaper and low cost maintenance of these machines. However, algorithm design techniques for cluster or heterogeneous machines still follow the approaches used in parallel computers, such as on shared memory machines. One of the crucial and critical issues in distributed computing is load balancing the tasks among the various processors. Load balancing is a difficult and challenging problem. The strategies/techniques vary greatly based on the applications.

In this thesis, we consider two scientific computing applications. Fast Fourier Transform (FFT) and multidimensional matrix multiplication. We design a parallel/distributed algorithm for the FFT problem by exploiting data locality so as to minimize communication. We implemented the algorithm on a Beowulf cluster. In comparison to the well known existing parallel algorithm, our algorithm, produces 15% speedup.

For the multidimensional matrix multiplication problem there are two issues to consider: representing the multidimensional data structure efficiently so as to minimize storage space and adopting a load balancing strategy for the partitioning of the data onto the heterogeneous machines. We will develop two strategies, static and dynamic. In static partitioning, the data is distributed before runtime. Dynamic partitioning follows a hierarchical manager-worker model where the tasks are allocated by the manager on-demand. The results indicate the dynamic strategy produces better results since it considers the underlying architecture of the heterogeneous machines for efficient load balancing.

Acknowledgments

I take immense pleasure in acknowledging several people who have supported me in different ways. To all of them, I would like to convey my heartfelt gratitude.

I am indebted to my advisor Dr. Parimala Thulasiraman, without whom this thesis would not be possible. Her continual guidance, suggestions and incredible patience has helped me achieve this goal. I would like to thank Dr. Ruppa k. Thulasiram, for his valuable suggestions and comments on thesis.

I would like to thank my committee members Dr. Pourang Irani and Dr. Ekram Hossain for their invaluable time.

My sincere appreciation to Gilbert Detillieux for his technical assistance. I am grateful for the experiences that I gained over the time I spent at University of Manitoba.

Special thanks to all my friends in Winnipeg for being the surrogate family during my stay. Thanks to my partner in life, Arvind, for supporting me during these years.

Lastly and most importantly, I would like to thank my parents. Their prayers and encouragements helped me complete my thesis in a distant land. I dedicate this thesis to them.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Parallel Computing has its roots in computational problems in physics (fluid flow, structural analysis) and engineering. These applications exhibit a regular structure. New applications have been added to the list over the recent years that are very irregular. These problems are classified as grand challenge problems [57], these problems cannot be solved with today's desktop computing.

Recently cluster and heterogeneous computing environment have gained popularity. These are distributed computing networks and are under study due to the low cost commodity components they utilize. Heterogeneous machines vary in processing speed, operating system and underlying hardware. Both cluster and heterogeneous computers can be classified as network of computers. Computationally intensive applications are suitable on such machines. However, not all problems are computationally intensive. There maybe some interaction in the concurrent tasks in the problem.

In this thesis, we study two fundamental scientific computing problems, the Fast Fourier Transform and multidimensional matrix multiplication algorithm on a cluster and heterogeneous computing environments.

In 1965, Jim Cooley and John Tukey developed an FFT algorithm for computing DFT in *O(n log n)* operations. FFT is a widely used algorithm in the field of engineering and computing science. FFT has been studied extensively as a frequency analysis tool in diverse applications. The FFT has several applications such as audio signal, frequency analysis [15, 35, 40] and several real time applications [21, 48]. FFT is used as a frequency analysis tool in a real time beat tracking system on parallel machines [21]. FFT is used for calculating the frequency spectrum for digitized acoustic signals.

Kumar et al. [22] developed a parallel iterative FFT algorithm based on the Cooley-Tukey method for exploring high performance computing. This algorithm is referred to as the unordered FFT, because the output data values are stored in bit reversed index order. Prior techniques have focused on parallelization using linear arrays [18] and mesh architectures [26]. The parallel FFT has been implemented on several parallel architectures [27, 50]. In parallel implementations, there are two kinds of latencies: synchronization and communication. In order to achieve high performance, requires both the latencies to be either hidden or tolerated. Though it is not a easy task, several techniques have been employed to hide or tolerate such latencies. One such approach is multithreading.

Thulasiraman et al. [50] applied the multithreading technique with a goal of hiding the latencies. Multithreading employs threads to overlap computations with communication. The multithreaded FFT algorithms were implemented on the EARTH (Efficient Architecture for Running THreads) architecture. Their approach was based on the data flow concepts. These parallel FFT algorithms have good performance results for large number of processors.

Recently, the multithreaded FFT algorithm has been implemented and tested for the option pricing problem in computational finance [49]. Using the multithreaded

FFT algorithm, the authors were able to achieve good performance results for large number of processors. In addition, the algorithm extracts full parallelism in the FFT computation. The results from the implemented algorithm clearly indicate that the algorithm could be used in real time applications.

Another recent method to tolerate latency, is by mapping data efficiently onto the processors local memory, that is exploiting data locality. Yeh et al [55] proposed an efficient parallel architecture for FFT in VLSI circuits. Efficient layout for interconnection networks, like butterfly networks are important for building parallel computers. The proposed swap networks [13, 55, 56] have efficient VLSI layouts and packaging properties. Data locality is exploited by having smaller layout areas and minimal link connections.

In this thesis, I consider the work of Yeh et al. [13, 55], to design my parallel FFT algorithm. The FFT is inherently a synchronous algorithm. However, we can improve on the communication latency by providing good data locality. In designing a distributed FFT algorithm for swap networks topology, a block data distribution is applied and data is mapped onto the available processors. This mapping reduces the amount of data swapping between processors significantly.

We discuss the parallel algorithm design of the FFT problem by exploiting data locality and implement the algorithm on a Beowulf cluster. Note that the FFT is nothing but matrix-vector multiplication. We could extend the 1D-FFT problem to 2D-FFT or multidimensional FFT. The underlying data structure for these problems is the multidimensional arrays. Representing multidimensional matrices on a computer is not a trivial task and is quite important to the overall performance of any algorithm. These matrices require abundance storage space. We implement a technique [32, 33] for representing multidimensional matrices. We use this technique in developing a parallel algorithm for multidimensional matrix multiplication.

Typically, the matrices are represented as arrays using the Traditional Matrix Representation (TMR) [31]. Recently, a new approach called the Extended Karnaugh Map Representation (EKMR) has been proposed [32] for representing multidimensional arrays. The efficiency of the parallel algorithms based on the EKMR have been tested on homogeneous parallel systems [33]. The results prove that the parallel algorithms based on EKMR outperforms the TMR based methods. The EKMR based parallel algorithms have been studied on distributed homogeneous memory machines. To our knowledge, the EKMR representation scheme has not been tested on heterogeneous clusters.

We consider heterogeneous machines to implement the parallel matrix multiplication algorithm using EKMR representation. These machines have variable processing speed, different operating systems, and different underlying hardware. It provides a cheaper alternative to buying traditional expensive supercomputers. The increase in low cost hardware technology, fast performance and standard APIs are paving way to widespread use of clusters and network of workstations.

The issue in designing a parallel algorithm on this type of machines is load balancing. This is a challenging problem since we need to distribute the tasks efficiently so as not to overload the system with too slow processing speed with abundant work. Allocating data and tasks to an individual processor depends on the processors speed, network bandwidth, memory constraints and the operating system. In this thesis, we study both static and dynamic load balancing strategies and performance of matrix multiplication for multidimensional arrays using the EKMR approach on heterogeneous machines.

The outline of the thesis is as follows: Chapter 2 gives a detailed introduction to parallel programming environments. In Chapter 3, the proposed parallel FFT algorithm is presented. The performance results of the parallel multidimensional

matrix multiplication algorithm using the EKMR is given in Chapter 4. Finally, conclusion and future extension of this work is presented in Chapter 5.

Chapter 2

Parallel Computing

Parallel computing provides a solution for large computational problems in a reasonable amount of time using more than one processing unit. Many *grand challenge problems* [57] such as quantum chemistry, astrophysics, computational fluid dynamics, medical imaging require high performance computing techniques and resources. There are many key ingredients necessary for parallel computing: operating system, programming model, parallel compilers and parallel algorithms.

The most important component needed to produce an efficient solution to a problem, however is *parallel algorithm design.* In this chapter, we discuss the classification of the parallel computers, parallel programming models and the parallel algorithm design techniques.

2.1 Classification of Parallel Computers

Many large commercial parallel machines are based on Flynn's taxonomy of computer architecture [22]. A computer's classification depends upon the parallelism it exhibits in the instruction stream and data stream. The four main models of compu-

tation are: Single Instruction Single Data Stream (SISD), Multiple Instruction Single Data Stream (MISD), Single Instruction Multiple Data Stream (SIMD) and Multiple Instruction Multiple Data Stream (MIMD).

2.1.1 Single Instruction Single Data (SISD) Model

The SISD model best describes the basic sequential von Neumann machine. The machine is made up of a main memory and a processor. This computational model takes a single sequence of instructions and operates on a single sequence of data. Data and programs are moved between the memory and the processing unit. The algorithm is executed serially on this computer. Modern uniprocessors may still exhibit some concurrency of program execution due to superscalar architectures. Superscalar architectures dynamically select and identify multiple independent operations that might be executed simultaneously. Though this might be an advantage it also has the drawback of limiting the performance of parallel programs due to the speed at which data are transferred from memory to the processor and vice-versa.

2.1.2 Multiple Instruction Single Data (MISD) Model

In the MISD model, the same data stream flows through a linear array of processors. The processors might however execute different instructions. Systolic arrays are MISD architecture. Image processing applications favored systolic arrays due to the inherent pipelined execution of an algorithm. This architecture is no longer in use.

2.1.3 Single Instruction Multiple Data (SIMD) Model

In this model several processors execute a single instruction on multiple sets of data. The SIMD model consists of an array of identical processing elements (PE), that are

controlled by a common global control unit (CU). Each PE stores the data in their local memory. During each instruction cycle, the control unit broadcasts the same instruction to all of its subordinate processing units. Therefore, at a given instance, the PEs operate synchronously by performing the same instructions on their local data sets. The processors work in lock step, thereby requiring a global synchronization mechanism among the processors. Such parallel machines are called *synchronous* programming models. The Massively Parallel Processor [3] and MasParMP-1 [6] are examples of SIMD architectures.

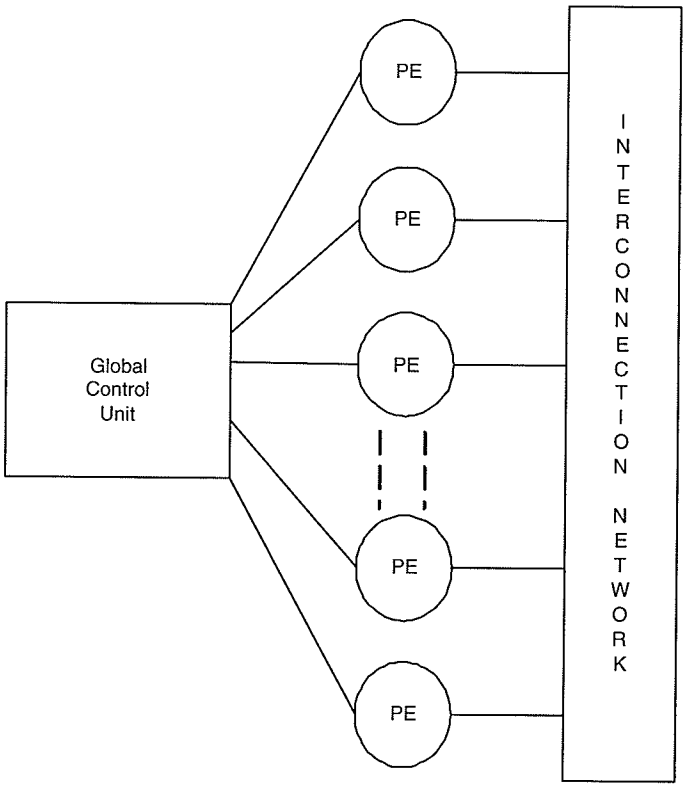

Figure 2.1: Single Instruction Multiple Data (SIMD) Model.

As can be seen from Figure 2.1, the SIMD architecture is nothing but a collection of sequential computers controlled by a global control unit. Processor arrays or vector

computers are SIMD machines. A vector computer is implemented as a sequential computer connected to a set of identical, synchronized elements capable of performing the same operation on different data. SIMD machines are most applicable to data parallel applications.

2.1.4 Multiple Instruction Multiple Data (MIMD) Model

In MIMD machines (Figure 2.2), each of the processing elements have their own control unit and local memory. Thus making it more powerful than the other models. The PEs are connected to each other via an interconnection network. Multiple processors can simultaneously execute different instruction streams manipulating different data streams. That is, each processor is capable of executing a different program independent of other processors. MIMD models are also referred to SPMD (Single Program Multiple Data) models if the same program is executed on every processor but with different data set. These are also called *asynchronous* programming models because they do not have a global clock to synchronize the various operations as in SIMD machine.

The two main classification of the MIMD model are: *shared memory multiprocessors* and *distributed memory multicomputers.* These models are discussed next.

2.1.4.1 Distributed Memory Multicomputers

In Distributed memory multiprocessor (Figure 2.3) model multiple processors operate independently and each has its own private local memory that is not accessible to other processors. In distributed memory machines, the number of processors could vary from few machines to several thousands, thus making it highly scalable and cost efficient. The computers are connected via an interconnection network. The

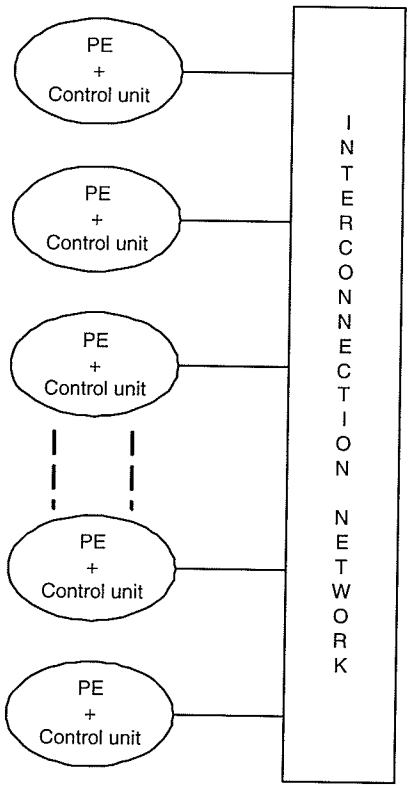

Figure 2.2: Multiple Instruction Multiple Data (MIMD) Model.

interconnection network acts as a bridge to pass messages between the processors.

Parallel programming on such machines is a bit more difficult than in shared memory machines. It is the users responsibility to partition the data, distribute and map the data among the processors efficiently. Since the machines do not share global (common) memory, the processors communicate by means of message passing. Parallel Virtual Machine (PVM) and Message Passing Interface (MPI) are two such message passing libraries. Intel Paragon [25], CM-5 [30], ncube [16] are few examples of distributed machines.

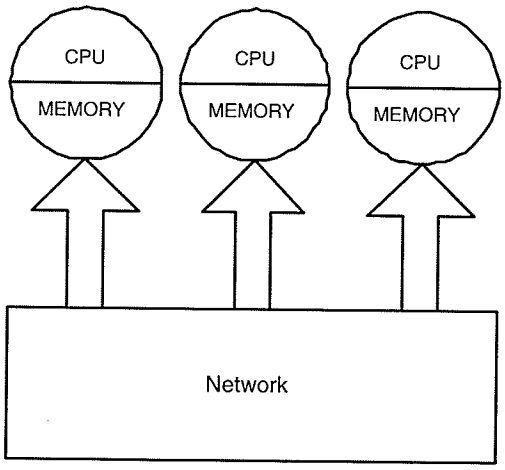

Figure 2.3: Distributed Memory Architecture.

2.1.4.2 Shared Memory Multiprocessors

Shared memory machines are basically an extension of the single processor model. To a user, the machine looks like a sequential machine with multiple processes running in parallel. In a shared memory multiprocessor model, all the processors share a common global memory via a interconnection network [22]. Shared memory machines employ a single address space concept, where each location in the whole memory has a unique address used by each processor to access the location. The shared memory architecture is shown in Figure 2.4.

Such machines are easy to build and program. Examples of such machines are SGI Origin 3000 [45], BBN Butterfly [8]. The users need not worry about partitioning their data among the processors and only partition the computations. The drawback of shared memory machines is that due to single address space concept, variables may be shared, and resource sharing may limit the speed of the computation due to synchronization primitives. In general, these machines use high speed caches.

There are three main categories of shared memory models: *Uniform Memory*

Access (UMA), Non-Uniform Memory Access (NUMA), and *Cache Only Memory Architectures (COMA)* multiprocessors. These models differ in how the memory and peripheral resources are shared and distributed.

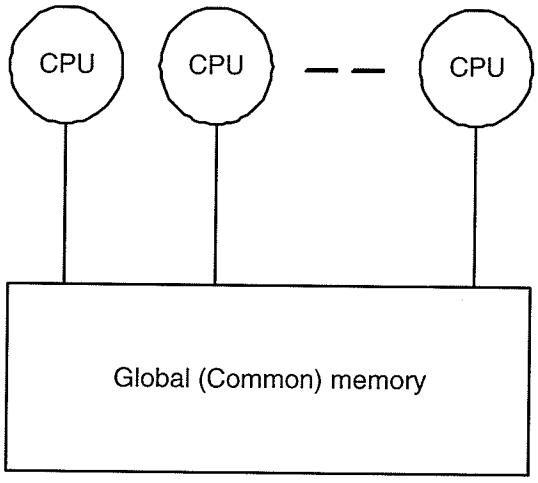

Figure 2.4: Shared Memory Architecture.

In UMA model (Figure 2.5), the time taken by each of the processor to access memory locations are uniform. As the data accessing times are uniform, programming is relatively easier in such models. Such machines are also called a *Symmetric Multiprocessors* (SMP). The SMP machines may vary from 2 to 64 processors. Since the number of processors are small, these machines are very cost effective. The interconnection is a bus, crossbar or network.

The disadvantage of the UMA machines is that they do not scale well. Caches are used to reduce access time to memory. However, this leads to cache coherency problems. That is, when two processors A and B read the same location from main memory, the values are stored in their respective caches. Processor B could modify its value and write the new value to memory. Processor A is unaware of this and has stale value. Protocols such as snooping [17] is used to solve the cache coherency

problem.

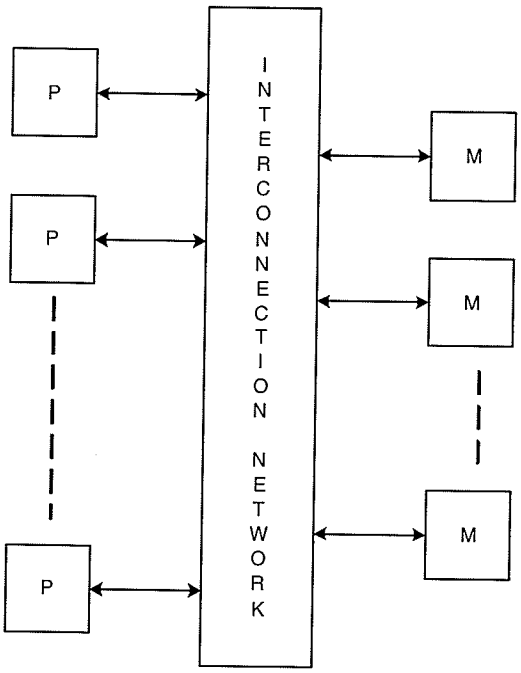

Figure 2.5: Uniform Memory Access.

UMA machines are suitable for general purpose and time sharing applications by many users. Large shared systems have hierarchical or distributed memory structure. These machines are called Non-Uniform Memory Access (NUMA) machines. In NUMA machines (Figure 2.6), each memory is closer to some processors than others. Each processor has its own local memory that is accessed faster than the remote memories. The shared memory is physically distributed to all processors called *local memories.* However, the machine operates as a shared memory machine and appears so from the programmerś perspective. The collection of local memories forms a global address space accessible to all processors. NUMA machines are also called *distributed shared memory* machines. The interconnection network is a grid or hypercube. These machines scale to more processors, but locality is harder to program

since local memory is faster to access via the interconnection network than remote memories.

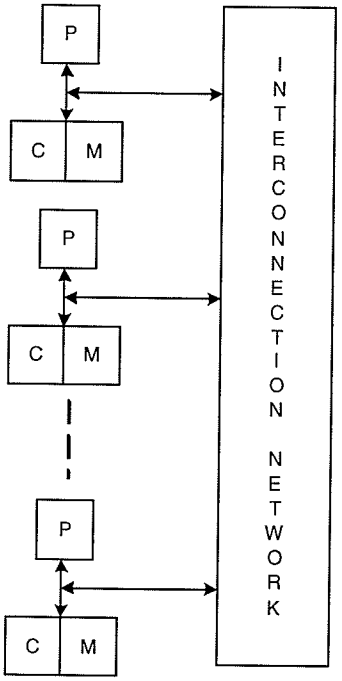

Figure 2.6: Non-Uniform Memory Access.

In NUMA machines, the access time between local and remote memory grows with number of processors. Therefore, caches become important. A variant of NUMA machines is Cache Coherent-NUMA (CC-NUMA). The cache coherency problem becomes difficult, however, and is alleviated by sophisticated cache coherency protocols such as the directory based protocol. SGI Origin 2000 and Sun Ultra HPC servers are some examples of the NUMA multiprocessors model.

COMA (Cache Only Memory Access) machines (Figure 2.7), are a special case of NUMA machines. The distributed memory is converted into cache. All caches form a global address space. Remote cache accesses are assisted by distributed cache directories. KSR-1 [41] is one example of COMA machines.

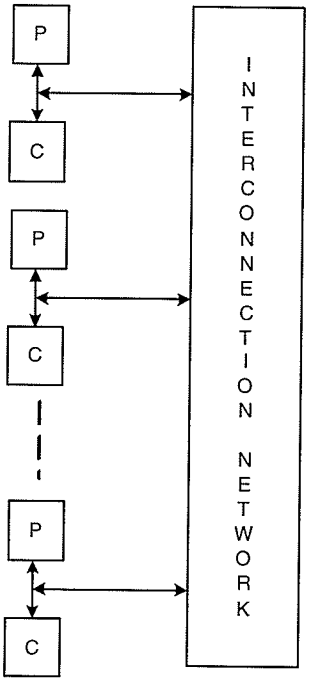

Figure 2.7: Cache Only Memory Access.

2.2 Multithreaded Paradigm

Massively parallel processors perform asynchronous computation by message passing through an interconnection network. In 1987, Arvind and Iannucci [1] realized two fundamental latency problems that asynchronocity triggers: *communication* and *synchronization*. Communication, or remote access causes a processor to be idle until the necessary data is fetched from the destination processor. In order to realize that the necessary data is ready at the destination processor, the processors must synchronize.

There are many techniques to hide, reduce or tolerate latencies in hardware and software. The most general technique is *multithreading* which tries to overlap computation with communication by means of threads (a thread is a set of instructions). The underlying principle in multithreading is that as long as there is enough parallelism in the algorithm, the processors are always busy executing threads.

There are different types of multithreaded constructs (languages, libraries or hardware). Tera [9] is a commercial multithreaded hardware architecture available at the San Diego Supercomputing Center. There are other research oriented multithreaded models such as Cilk [7, 44] which is a fine-grained multithreaded language developed at MIT. EARTH [23] is another example which is a fine-grained data flow multithreaded architecture developed at McGill University (a fine-grained thread is a thread with very few instructions). PThreads and Solaris threads are commercially available library routines developed at Sun Microsystems. Java is an object-oriented language that supports threads. These language based threads are coarse-grained (many instructions per thread), operating system dependent and slow.

Currently OpenMP is regarded as the standard language for multithreading. It is designed for shared-memory multiprocessors. OpenMP will be discussed in detail in section 2.3.2. Here, a brief overview of other multithreaded paradigms is presented.

2.2.1 EARTH

The EARTH (Effective Architecture for Running THreads) [23, 46] is a multithreaded program execution model targeted to high performance parallel and distributed multiprocessing. Program code is divided into threads that are scheduled atomically using data flow like synchronization operations. EARTH supports latency tolerance by efficient exploitation of fine grained parallelism by overlapping computation with communication. EARTH nodes consist of an *Execution Unit (EU)* and *Synchronization Unit (SU)*, which share the processors local memory. The *EU* executes the fine grain threads, while the *SU* determines which thread is to be executed and handles communication between the processors. Currently, programs running on EARTH are written in Threaded-C [53], which extends the C language with multi-threading

instructions.

2.2.2 Cilk

Cilk is a fine grained multithreaded language [44] developed at MIT. It provides mechanisms to exploit dynamic, asynchronous parallelism. Cilk is an algorithmic language. The performance of a multithreaded algorithm can be determined through complexity analysis, by using two parameters *work* and *critical path length.* The run-time system employs an algorithmic scheduler based on the concept of work stealing and handles load balancing issues and communication protocols. The construct "cilk" is key in the cilk codes and by removing the constructs a cilk code becomes a pure sequential code. This is very advantageous in building the cilk code in a modular fashion as well as for performance evaluation.

2.2.3 Tera

Tera is a highly scalable multithreaded architecture that supports very fine grained multithreading [9] on a shared memory architecture. Each processor supports 128 instruction streams (threads). The thread scheduling is preemptive. Memory latency is masked by fast context switching between the threads. All the processors have uniform access to the shared memory without caching. Tera supports TeraOS, a Unix based operating system. The runtime system manages the interface between hardware and user programs and Tera's operating system. The first Tera multithreaded machine exists at San Diego Supercomputing Center [37].

2.2.4 Java

Java [39] is an object oriented, Internet aware language, that allows for concurrent programming for both single and multiprocessors system. Java allows users to write platform independent multithreaded programs conveniently by means of threads. Java has a thread scheduler to monitor all running threads. Java scheduler could be preemptive or non-preemptive. Java is not a good choice for fine grained parallelism, because of slow context switching between the threads.

2.2.5 PThreads

PThreads (POSIX threads) are commercially available and was developed at Sun Microsystems [38]. The PThread library calls use pthread, in front of every call. The threads follow a two-level multhreaded model. Programs are written using user level threads. The threads are scheduled onto light weight processes or kernel level threads. The kernel level threads are scheduled to the available processors. This scheduling is transparent to the user. PThreads are widely used by parallel programmers, since it is an IEEE standard.

2.3 Message Passing and Shared Memory Programming

In this section we briefly discuss the Message passing Interface (MPI) and OpenMP.

2.3.1 Message Passing Interface (MPI)

The increasing popularity of distributed computing has produced a need for a standardized portable library. Message Passing Interface (MPI) has evolved from such a

demand. MPI is one of the most widely used programming paradigm for distributed memory models. MPI provides flexibility, by allowing the users to write message passing programs over any hardware with suitable operating system support. The library functions can be implemented using several programming languages like Fortran, C and C++.

Communication is achieved among the several hundreds of processors in the network using MPI library functions. Message passing computing provides a method for creating separate processes for execution on different processors and a method for sending and receiving messages between processors. All processes are statically created in MPI. That is, the processes are created at the start of the program and is therefore, fixed. MPI follows a master-slave model. There is one master process and many slave processes. The same program is executed by many processes on its own local data. It is therefore, a single program multiple data model (SPMD).

MPI provides point to point and collective communications. Point to point communication occurs between two processes only. Collective communication such as a broadcast operation involves more than one process in sending and receiving.

Blocking and non-blocking message passing is available in MPI. Blocking message passing routines do not allow the process to continue with the next statement until the transfer is completed. These routines return when they are locally complete, meaning when the location used to hold the message can be safely altered without affecting the message being sent. Non-blocking routines return whether or not the message has been received. This is accomplished by means of buffers at the receiving process.

Non-blocking routines are attractive because they allow asynchronous computations. A process can continue with other statements following a send operation as long as the statements are not in any way dependent on the location used by the

send operation. When it requires to alter the location, the process has to specifically use test and wait primitives to determine the status of the send operation. These routines try to overlap computations with communications. However, a process may have to spend time continuously testing and waiting wastefully using the processors resources. Multithreading avoids this by decomposing the program into many fine grained independent programs.

The parallel algorithms using the MPI standard do produce desirable results when run on homogeneous clusters. When the same applications are run on heterogeneous machines [28] the performance were shown to degrade. There are several reasons behind such a failure in performance. Homogeneous clusters contain processors with the same processing capabilities. It is not the case with heterogeneous clusters. These clusters could contain processors with varying speed, operating system and network bandwidth. A parallel algorithm is seriously affected by the slowest processor in the cluster. mpC [28] is recently becoming popular and supports programs running on heterogeneous machines.

2.3.2 OpenMP

The OpenMP API comprises of library routines and complier directives. This interface supports programming languages like C, C++ and Fortran. Several compliers support OpenMP specifications. Thereby, allowing parallel programs to be portable across several compliers [28].

The OpenMP parallel programming model is designed for shared and distributed shared memory multiprocessors [11]. This avoids the overhead inherent in message passing systems. The OpenMP compiler directive in C and C++ are defined by #pragma directive. These compiler directives allow the OpenMP programs to be

portable to platforms that do not support OpenMP. On such platforms, the directives of the OpenMP programs are ignored and are treated as a sequential program.

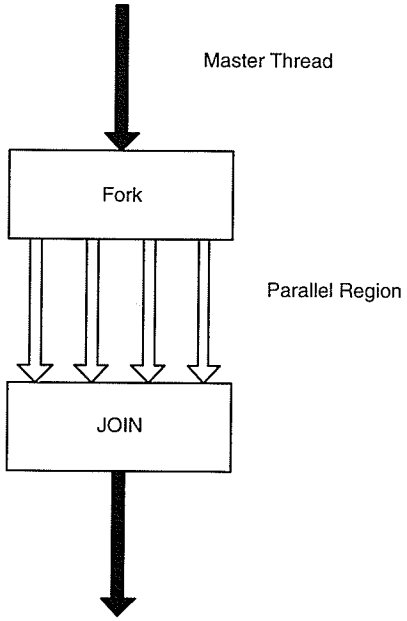

Figure 2.8: Fork-Join Model.

In general, an OpenMP program executes sequentially on a single thread (called a master thread), just like a serial program. The execution model is based on the fork and join model (Figure 2.8) where the *master* thread forks many *slave* threads. When the program encounters the parallel constructs, a group of parallel threads (called slave threads) are created (fork operation) and each thread executes a copy of the body of code enclosed within the parallel directive. When each slave thread completes execution (join operation), the master thread gains control of the program and executes the program sequentially until it encounters another parallel directive.

A simple OpenMP *parallel* directive:

! *begins with one thread*

! *#PRAGMA OMP PARALLEL*

! *code is replicated over*

! *a team of threads*

! *end of parallel region*

! *all threads join to form one thread*

OpenMP exploits both loop level parallelism and parallel regions. In loop level parallelism, each iteration of a loop is considered as a thread. For example,

$\#pragma\ omp\ parallel\ \ for(i = 0; i < N; i++)$

{

body

}

The master thread forks N independent threads (slave threads). Each thread executes the body of the code. N threads are executed concurrently. The variable i is private by default. Loop level parallelism exploits fine grained parallelism. There is an implicit barrier at the end of the *for* loop. The master thread resumes execution once all the slave threads have completed (join). At this point the slave threads die and the memory allocated for these threads are removed. The master thread resumes. Parallel regions in OpenMP allow SPMD programming. Any program written in MPI can be rewritten in OpenMP using parallel regions. The threads used by parallel regions are coarse grained. Also, the slave threads created in parallel regions do not die. They get suspended until another parallel region is encountered.

Loop level and parallel region parallelisms exploit domain decomposition. OpenMP also allows functional decomposition by means of *section*. OpenMP has become popular among parallel programmers due to its flexibility and ease in programming. IBM Seaborg [24] is an example of machine that employs hybrid parallel programming. It

consists of 6080 distributed memory computers. There are 380 nodes with 16 processors per node. MPI is used between the nodes and OpenMP is used within the processors, since the processors are in shared memory paradigm.

2.4 Parallel Algorithm Design

There are applications that can be classified as *embarrassingly parallel* problems. In these applications, there is no communication between processors except at the beginning when the data is distributed and at the end, when the data or solution is gathered by one central (master) processor. The processors execute the same algorithm on their individual local data. For such problems, the SPMD model is appropriate. Monte Carlo simulation is one example of an embarrassingly parallel problem.

Designing parallel algorithms for embarrassingly parallel problems is very simple. There are applications that require much more sophisticated design techniques.

When designing parallel algorithms, we have to keep four attributes in mind: concurrency, scalability, locality and modularity. Concurrency is the ability to perform many actions in parallel. As the number of processors increases, the problem also has to scale well. The key performance on high performance multicomputers is locality-to make sure there is high ratio of local memory access to remote memory access. And finally using software engineering techniques to decompose complex problems into simpler components.

The algorithm design follows four phases: partitioning, communication, agglomeration and mapping. Partitioning and communication deal with concurrency and scalability. Agglomeration and mapping deals with locality and performance issues.

In the partitioning phase, we try to expose maximum parallelism. A problem

can be partitioned using domain decomposition or functional decomposition. In domain decomposition, data is first divided into approximately equal sizes and then the computations are associated with the data. In functional decomposition, the computations are divided into smaller tasks. The tasks are executed simultaneously to reduce execution time on the entire problem.

In the communication phase we have to decide the type of communication involved in the problem. In local communication, each task communicates with a small set of neighboring tasks. In global communication, each task communicates with several tasks. The communication pattern could depend on the structure of the tasks. For structured communication, a task and its neighbors comprise a regular structure (grid). In unstructured communication, the networks maybe arbitrary graphs (sparse matrix).

In static communication neighboring tasks do not change over time. Dynamic communication is more complex, as the partners change over time (determined at runtime). When there is coordination between the producer and the consumer, then communication pattern is synchronous. For asynchronous communication, there is no coordination between the producer and consumer. The producers cannot predict when consumers require the data. The data is placed in the queue when it is ready. The consumer keeps checking the queue continuously to retrieve the data.

In the agglomeration phase, we try to reduce the communication and software engineering costs. Communication overhead can be minimized by combining all the tasks that communicate with each other. When agglomerating tasks for parallel design, care should be taken not to combine too many tasks limiting the scalability of the algorithm. Once a sequential code is carefully agglomerated, parallelization of the code will be less time consuming and cost efficient.

In the final mapping phase, we try to minimize total execution time. The proces-

sor utilization is maximum when the computation is balanced evenly and executed concurrently on all the processors. Placing the tasks that communicate frequently on the same processor increases locality.

Efficient parallel algorithm design depends on types of interactions between the tasks for data sharing and synchronization information. The efficacy of parallel algorithm could vary on different programming paradigms, depending on the nature of interactions among the tasks. The type of interactions could be static or dynamic, regular or irregular.

In static type of interaction, the exact time of interactions between the tasks and the sets of tasks to interact are known in advance. Programming for static interactions is easy in message passing paradigms (MPI). In dynamic type of interaction, the time of interaction between the tasks is undetermined before the execution of the program. Due to the unpredictable nature of interactions between the tasks, designing efficient algorithms becomes more difficult.

Applications can be classified as either regular or irregular. Data structures such as a dense matrix are regarded as having a regular structure. Applications that pose regularity are very easy for algorithmic design. Irregular applications require sophisticated data structures such as trees, unstructured meshes etc. The algorithms for these applications are usually asynchronous. The pattern of data distributions is non-uniform. To design parallel algorithms for such applications requires facilities for dynamic creation of work and dynamic load balancing.

Irregular problems pose challenging research issues. With the advent of heterogeneous distributed environments, scheduling of tasks has become very difficult, especially for irregular problems.

In this thesis, I will look at both a regular and irregular applications on a homogeneous and heterogeneous computing environment.

2.5 Summary

In this chapter we presented a brief introduction to the Single Instruction Single Data (SISD), Multiple Instruction Single Data (MISD), Single Instruction Multiple Data (SIMD) and Multiple Instruction Multiple Data (MIMD) parallel programming models. A brief introduction to the different multithreading paradigms such as EARTH, Tera, Cilk, Java and PThreads were discussed. Some of the widely used parallel programming paradigms, like OpenMP and MPI were discussed. A brief introduction to the parallel algorithm design was also presented. We use these concepts in the following chapters where we design algorithm for some selected problems.

Chapter 3

Parallel Fast Fourier Algorithm

In this chapter, we introduce a general idea of Discrete Fourier Transform (DFT) and Fast Fourier Transform (FFT). We discuss the design, development and implementation of the FFT algorithm as well.

3.1 Fast Fourier Transform

The Fast Fourier Transform (FFT) is a fast algorithm for computing the Discrete Fourier Transform (DFT). The FFT has diverse applications areas such as audio signal and frequency analysis [15, 35, 40], and several real time applications such as finance [48], oceanography and music [21]. It has been studied as a frequency analysis tool. Before, discussing the FFT, a brief explanation of the DFT will be given.

In a given problem, the amount of data considered is huge. These data could be collected and observed for analysis, such as in oceanic tidal data collected over several years, stock market data collected over several months or communication signals data collected over microseconds. We assume that there are repeated patterns in these data. The DFT tries to determine these cyclic patterns. A pattern may repeat m

times in the data of size N. This is called the Fourier frequency.

The DFT computations can be expressed as a matrix-vector multiplication,

$$Y(x) = \sum_{j=0}^{n-1} a_j x_j \tag{3.1}$$

Where the coefficients of matrix A are complex numbers in the form $(a_0, a_1, ..., a_{n-1})$. If we let $y_k = Y(\omega_n^k)$ then,

$$y_k = Y(\omega_n^k) = \sum_{j=0}^{n-1} a_j \omega_n^{kj} \tag{3.2}$$

where $\omega_n = e^{2\pi i/n}$ and $k = 0, 1, ..., (n-1)$.

We assume n is a power of 2. ω_n is defined as the twiddle factor and is the n^{th} root of unity, a complex number. There are exactly n complex n^{th} roots of unity: $\omega_n^0, \omega_n^1, ..., \omega_n^{n-1}$. All complex roots of unity can be expressed as,

$$\omega_N^x = \omega_N^{x \ mod \ N} \tag{3.3}$$

The twiddle factor is periodic and the vector is symmetric. The vectors are equally spaced around the circle with spacing $\Delta\omega$,

$$\Delta\omega = \frac{2\pi}{n}$$

For example, with $n = 8$, there are exactly 8 twiddle factors placed evenly around the circle of unit radius centered at the origin of the complex plane as shown in Figure 3.1.

Note that $\omega_8^5 = -\omega_8^1$ and $\omega_8^7 = -\omega_8^3$. For $n > 8$, we can use the equation 3.3 to determine the twiddle factor. For example, $\omega_8^{24} = \omega_8^{24 mod 8} = \omega_8^0$. Equation 3.2 can be represented as:

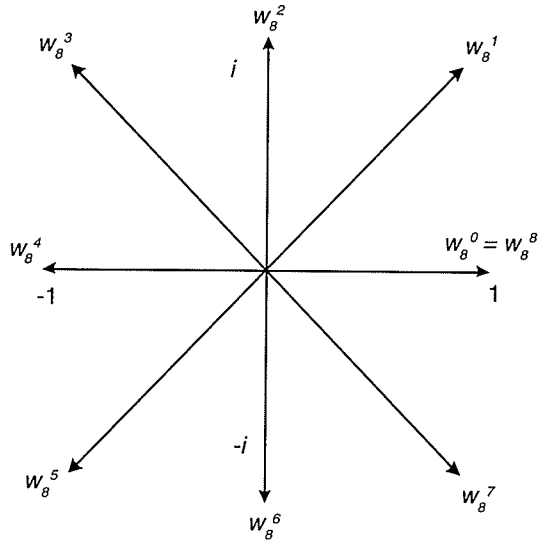

Figure 3.1: Complex roots in complex plane.

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \omega_8^1 & \omega_8^2 & \omega_8^3 & \omega_8^4 & \omega_8^5 & \omega_8^6 & \omega_8^7 \\ 1 & \omega_8^2 & \omega_8^4 & \omega_8^6 & \omega_8^8 & \omega_8^{10} & \omega_8^{12} & \omega_8^{14} \\ 1 & \omega_8^3 & \omega_8^6 & \omega_8^9 & \omega_8^{12} & \omega_8^{15} & \omega_8^{18} & \omega_8^{21} \\ 1 & \omega_8^4 & \omega_8^8 & \omega_8^{12} & \omega_8^{16} & \omega_8^{20} & \omega_8^{24} & \omega_8^{28} \\ 1 & \omega_8^5 & \omega_8^{10} & \omega_8^{15} & \omega_8^{20} & \omega_8^{25} & \omega_8^{30} & \omega_8^{35} \\ 1 & \omega_8^6 & \omega_8^{12} & \omega_8^{18} & \omega_8^{24} & \omega_8^{30} & \omega_8^{36} & \omega_8^{42} \\ 1 & \omega_8^7 & \omega_8^{14} & \omega_8^{21} & \omega_8^{28} & \omega_8^{35} & \omega_8^{42} & \omega_8^{49} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \end{pmatrix} \tag{3.4}$$

Given a data size of n, there are n equations. This requires $O(n^2)$ computations. However, let us take a second look at the equations. Let us consider two coefficients y_1 and y_5, simplified using the relation 3.3

$\omega_8^4 = -\omega_8^0$, $\omega_8^{12} = \omega_8^4 = -\omega_8^0$, $\omega_8^{16} = -\omega_8^0$, $\omega_8^{24} = \omega_8^0 = 1$, $\omega_8^{28} = \omega_8^4 = -\omega_8^0$,

$\omega_8^6 = -\omega_8^2$, $\omega_8^7 = -\omega_8^3$, $\omega_8^5 = -\omega_8^1$, $\omega_8^4 = -\omega_8^0$.

$y_1 = [1 \quad \omega_8^1 \quad \omega_8^2 \quad \omega_8^3 \quad \omega_8^4 \quad \omega_8^5 \quad \omega_8^6 \quad \omega_8^7][a]$

$y_5 = [1 \quad -\omega_8^1 \quad \omega_8^2 \quad -\omega_8^3 \quad \omega_8^4 \quad -\omega_8^5 \quad \omega_8^6 \quad -\omega_8^7][a]$

If we group the odd and even coefficients,

$y_1 = (a_0 + a_2\omega_8^2 + a_4\omega_8^4 + a_6\omega_8^6) + (a_1\omega_8^1 + a_3\omega_8^3 + a_5\omega_8^5 + a_7\omega_8^7)$ or

$y_5 = (a_0 + a_2\omega_8^2 + a_4\omega_8^4 + a_6\omega_8^6) - (a_1\omega_8^1 + a_3\omega_8^3 + a_5\omega_8^5 + a_7\omega_8^7)$

If we let,

$C = (a_0 + a_2\omega_8^2 + a_4\omega_8^4 + a_6\omega_8^6)$ and

$D = (a_1\omega_8^1 + a_3\omega_8^3 + a_5\omega_8^5 + a_7\omega_8^7)$

then,

$y_1 = C + D$

$y_5 = C - D$

So y_1 is the summation of even and odd parts, while y_5 is the difference of the two parts. This will be visible for y_0 and y_4, y_2 and y_6, y_3 and y_7.

This implies that if we calculate $\frac{n}{2}$ of y_k values $(k = 0, ..., \frac{n}{2} - 1)$, the rest of $y_{\frac{k+n}{2}}$ values $(k = 0, ..., \frac{n}{2} - 1)$ can be easily calculated through a difference of the even and odd parts.

We can recursively solve the above equations or subproblems. The above computation is called the butterfly computation (Figure 3.2) and can be conceptually described as follows: a and b are points or complex numbers. The upper part of the butterfly operation computes the summation of a and b with a twiddle factor ω while the lower part computes the difference. In each iteration, there are $\frac{N}{2}$ summations and $\frac{N}{2}$ differences. The sequential FFT can be executed in $O(NlogN)$.

Figure 3.3 shows an example of the FFT decomposition. A 8 point signal is decomposed through 3 stages. There are therefore $logN$ stages. The data is separated into odd and even numbered samples. Since there are n equations, each equation requires $log\ n$ steps. The complexity of the algorithm is $n\ log\ n$.

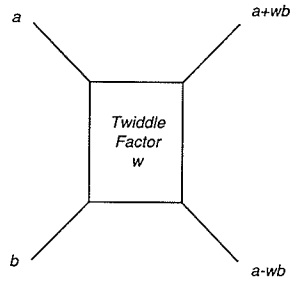

Figure 3.2: Butterfly Computation.

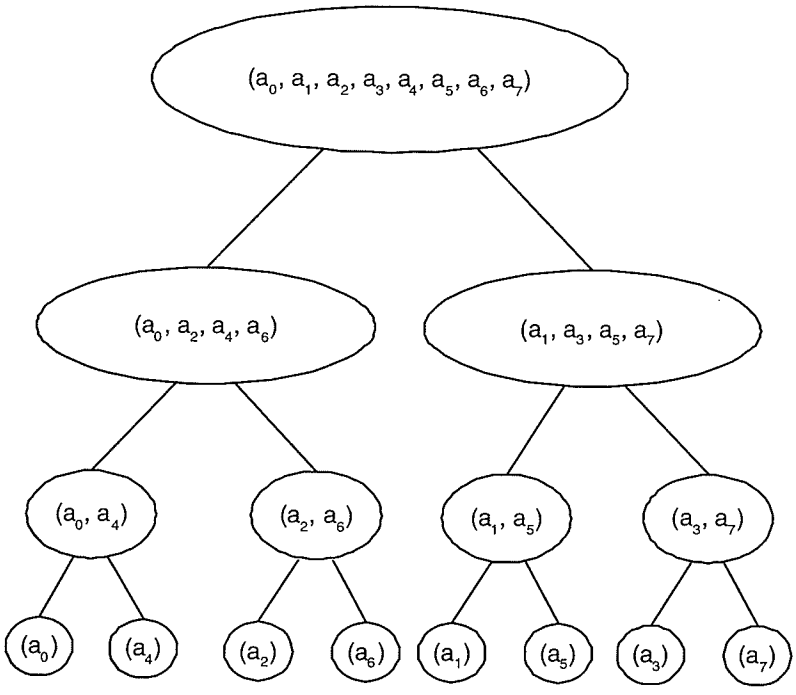

Figure 3.3: Tree structure for recursive FFT.

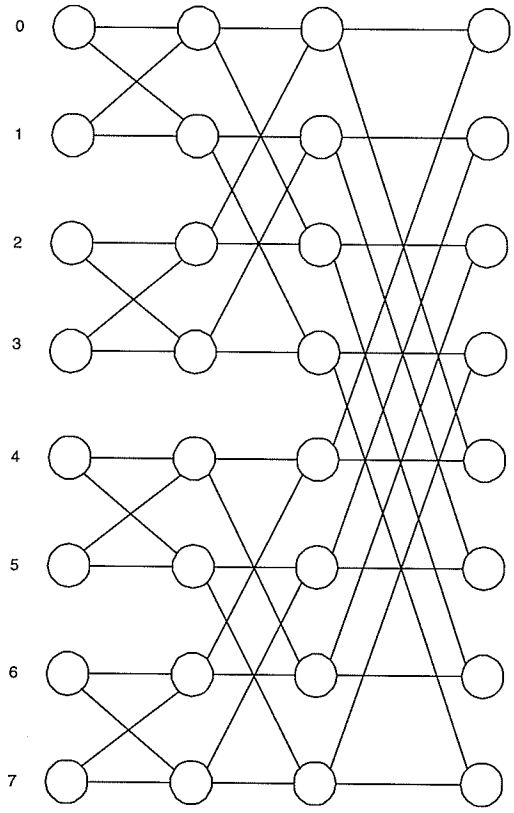

Figure 3.4: Butterfly Network.

3.2 Parallel FFT

First, before discussing the parallel FFT developed by Kumar et al. [22], let us briefly consider the sequential DFT algorithm and how to parallelize this algorithm:

$for(k = 0;\ \ k < n;\ \ k++)$

$\{\ \ Y[k] = 0;$

$for(j = 0;\ \ j < n;\ \ j++)$

$Y[k] = Y[k] +\ \ \omega^{j*k}x[j];$

$\}$

We can apply the master/slave approach and create n slaves to calculate each *Y[k]*. Each slave computes the inner loop j for its value k. The master broadcasts the vector x and ω to each slave. Slave 0 gets ω^0, slave 1 gets ω^1 and slave n gets ω^n (Figure 3.5). Slave k can compute ω^{jk} by using $\omega^{(j-1)k}$ (the value that was calculated in the previous iteration). Since $\omega^{jk} = \omega^k\omega^{(j-1)k}$, we could therefore avoid redundant computation of ω^{jk} at every iteration in this way. We can also cut down the number of slaves to $\frac{n}{2}$, if we allow symmetry.

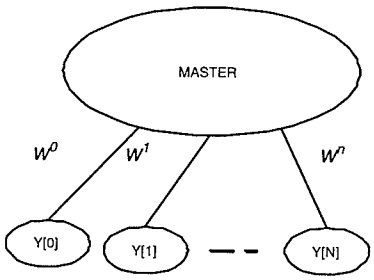

Figure 3.5: Master/Slave approach.

In this algorithm, the amount of computation performed by each slave is n. The amount of communication is in distributing ω^{jk} to each slave and getting the result back from each of the n slaves. Therefore, the total communication is $2n$ with some

startup cost associated with each send operation. The startup cost depends on the interconnection network.

Now, let us look at the FFT algorithm. We assume the recursive approach, the simplest case is when each slave computes a Y value recursively. This is a coarse grained algorithm. If n is large, the number of computation per slave increases. We could come up with a different technique by distributing the data among the processors and allow each processor to compute a subset of Y's. Usually we do not have n number of processors. However, in this scenario, since the algorithm requires the even and odd parts of the vector and $\omega's$, the processor has to determine the location of these data. This involves further communication. In a distributed memory machine the recursive approach is usually not very efficient. We therefore, discuss the iterative approach of the FFT algorithm.

In general, iterative version of the FFT algorithm is more suitable for distributed memory parallel machine. Let us assume we have $N(N = 2^m)$ data elements and $P(P = 2^p)$ processors. A *butterfly* (Figure 3.2) computation is performed on each of the data points in every iteration and there are altogether log N iterations.

In general, a parallel algorithm for FFT, with blocked data distribution of N elements on P processors, involves communication for *log P* iterations and terminates after *log N* iterations. The first *log N - log P* iterations require no communication and a sequential FFT algorithm can be used inside each processor. At the end of the (*log N - log P*) iteration, the latest computed values for $\frac{N}{P}$ data points exist in each processor. The rest of the *log P* iterations require remote communication.

In the butterfly network (Figure 3.6), each processor needs to communicate $\frac{N}{P}$ values to its partner data points which are not located within its own local memory. The design of a parallel FFT algorithm is driven by the data distribution onto processors rather than the actual physical connection of the processors. There are

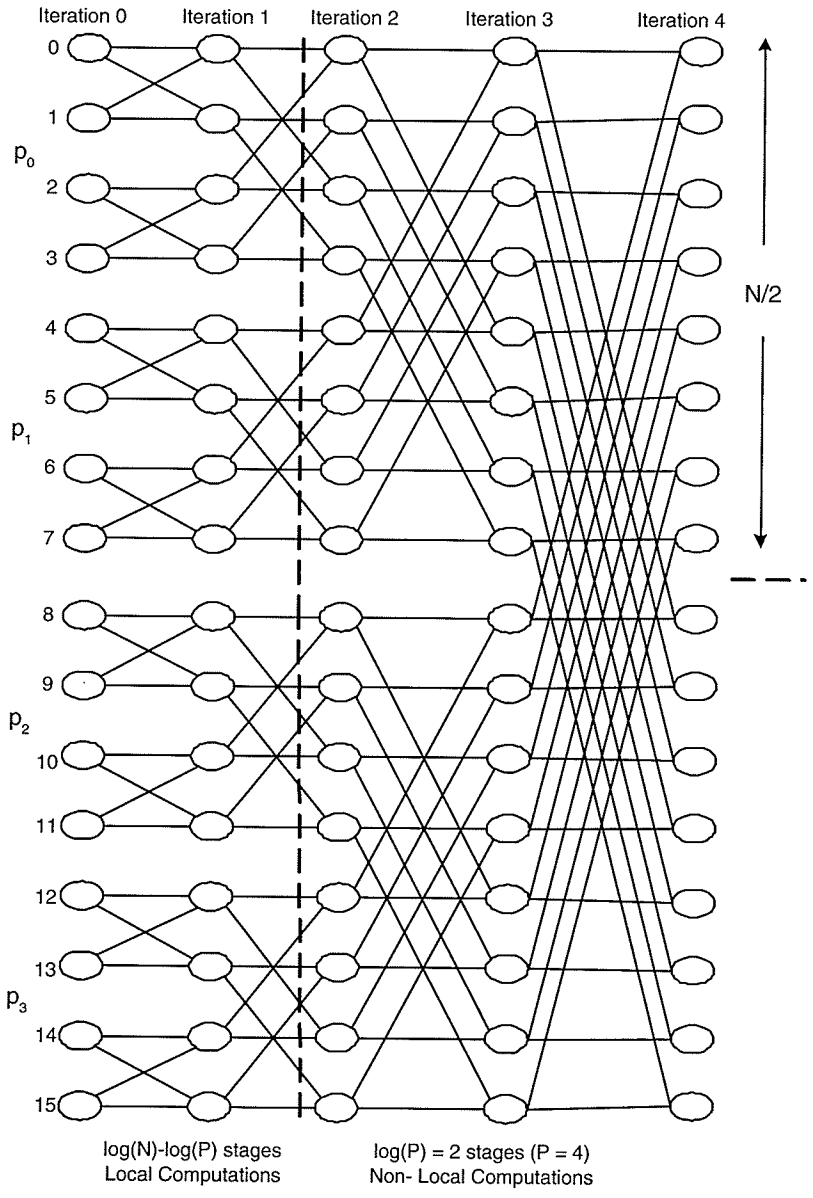

Figure 3.6: FFT Network.

two fundamental latencies in the FFT: synchronization and communication. At every iteration of the algorithm, each data point is dependent on the value of the previous iteration, leading to synchronization. To ensure the data is readily available at the corresponding processors, the values have to be communicated. To tolerate these two latencies, one approach is multithreading [50]. The other approach is to have good mapping of the data such that the data is available in the local memory of the processors. From the work conducted by Yeh et al. [13] in VLSI circuits, we have designed and developed an FFT algorithm that exploits data locality, by means of a swapping technique.

3.3 Localized Swapping Algorithm

Indirect Swap Networks (ISN) [13, 55] is a recently proposed network. The ISN network has smaller layout areas and minimal link connections. The ISN has efficient VLSI layout and packaging properties [55, 56]. The communication pattern in ISN is more localized than the butterfly network.

The ISN can be transformed into a butterfly network by modifying some of the stages that are connected by swap links. Figure 3.7 shows a $4X4$ ISN network and its $4X4$ butterfly transformation.

In the ISN network (Figure 3.8), let us consider $N(N = 2^m)$ data elements and $P(P = 2^p)$ processors. For block distribution the N data elements as before are distributed over P processors. In Figure 3.8, we see that ($log\ N$ - $log\ P$) iterations are local computations and the $log\ P$ iterations require remote communications. The difference, however from the previous parallel FFT techniques (Figure 3.6) is that data is swapped at every iteration.

For example, iteration 2 in the Figure, processors 0 and 1 exchange data points

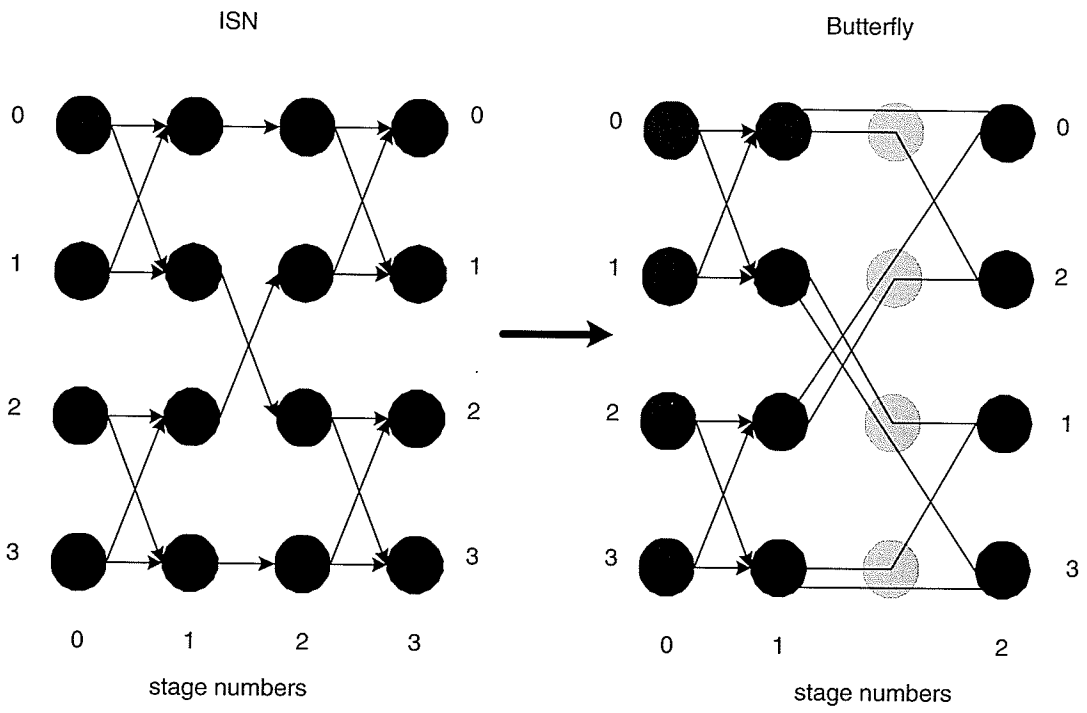

Figure 3.7: Transforming of ISN into Butterfly Network.

2, 3 and 4,5 respectively. In iteration 3, processors 0 and 2 exchange 4,5 and 8,9 respectively. In general for a given N and P processors, $\frac{N}{2P}$ data points are swapped between two processors. This results in half the reduced communication of the butterfly network in Figure 3.6.

In the next section, we will see that this new swapping technique produces better performance than the butterfly network.

3.4 Experimental Results

The swap parallel algorithm is implemented on a Beowulf cluster running MPI. Each node in the Beowulf cluster consists of 2 processors each with Pentium III processors running at 501.146 MHz and has a memory of 526 MB with cache size of 512 KB.

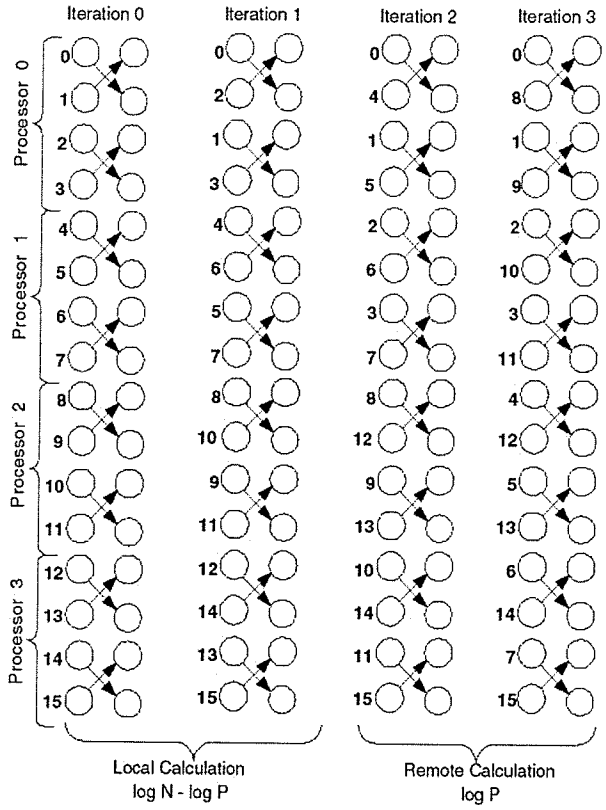

Figure 3.8: ISN Network.

The experiments are run on 16 processors (the maximum available at the University of Manitoba).

Figure 3.9 shows the execution time for various number of processors. We see from the figure that for 8 processors and data size of 2^{20}, the execution time of the algorithm is much less than that of 2 processors. This shows that the processor utilization on efficiency is significant. The reason for this is that each processor gets $\frac{N}{2P} = \frac{2^{20}}{8} = 2^{17}$ amount of data while on 2 processors, the amount of data per processor increases to 2^{19}. The smaller number of procesors perform more computations that the larger number of processors. This figure suggests that having too coarse grained algorithm is not always efficient. By using 8 processors we decrease the workload among the

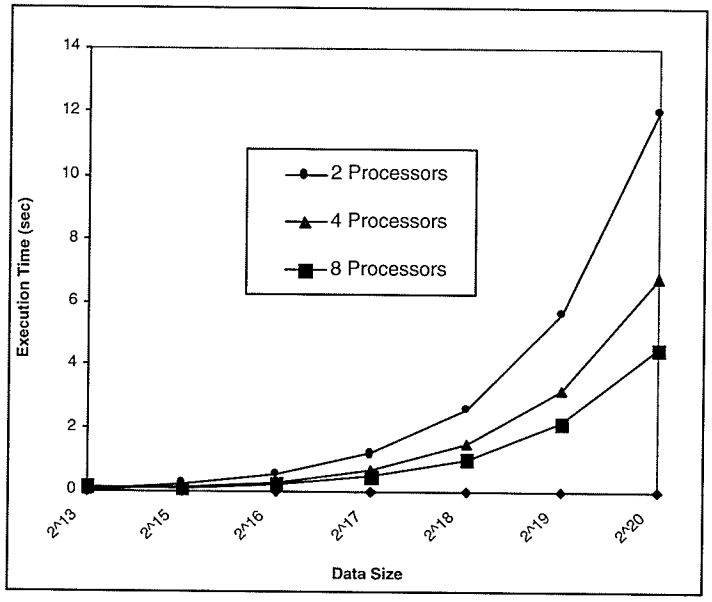

Figure 3.9: Performance results with varying processor size.

processors.

Figure 3.10 shows the execution time for varying data size. As the machine size increases, we see a gradual decrease in execution time for all data sizes. For P processors, the amount of data to be communicated is $\frac{N}{2P}$ per processor for $log\ P$ iterations. For large data size, 2^{20}, the amount of communication is about 2^{18} and 2^{16} on 2 and 8 processors respectively. In Figure 3.8, notice that a processor P_i exchanges data with only one other processor P_j. Therefore, each processor can send $\frac{N}{2P}$ data set in one single send operation, by packing the $\frac{N}{2P}$ in an array. In MPI, packing and unpacking data takes substantial amount of time. So, a slight improvement in terms of communication, produces significant performance.

Figure 3.11 shows the comparison between the butterfly and the swap network. We observe some performance improvement for large data sizes 2^{20}. We note that a slight improvement in enhancing the performance through algorithm design has

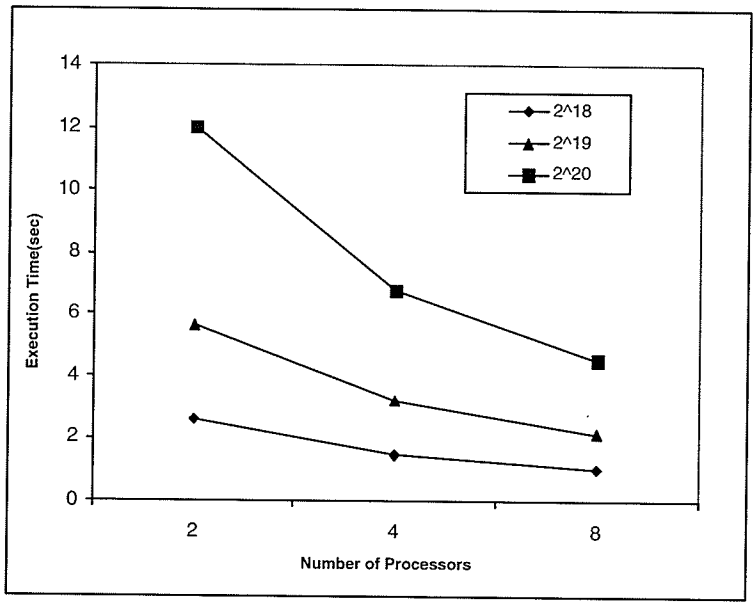

Figure 3.10: Performance results with varying problem size.

considerable effect on FFT algorithm. Algorithm design, therefore, is an important component for all applications.

In the distributed implementation, we observe that swapping of data at every iteration does not degrade the performance. In fact, data locality is improved by bringing the data closer to the processors and hence allowing faster computation. Also, this enhanced data locality incurs fewer communication overhead. In the FFT problem, it is difficult to control the synchronization at every iteration but communication overhead can be controlled to an extent.

3.5 Summary

In this chapter, we started by discussing the DFT and the FFT algorithm in detail. The DFT has a complexity of $O(n^2)$, while the FFT has a complexity of $O(nlogn)$.

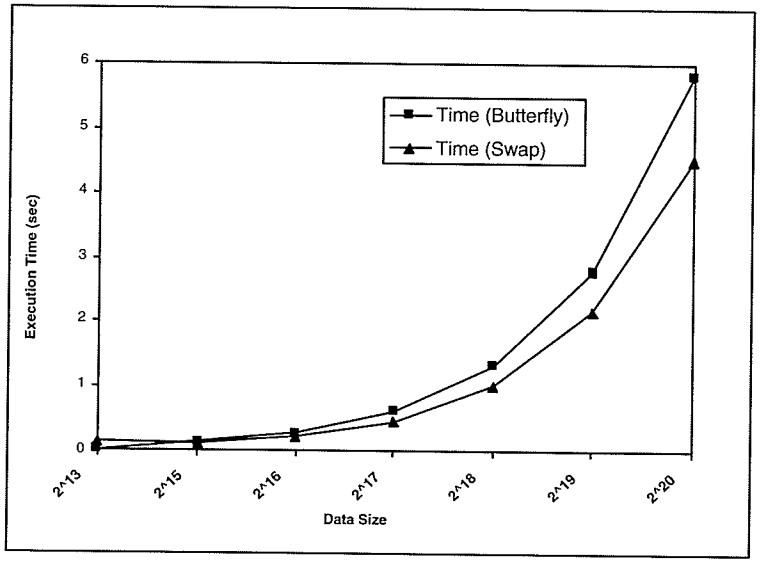

Figure 3.11: Comparison of the Butterfly Network and Swap.

We showed the parallel algorithm design of the DFT using a master/slave approach. The FFT can be computed recursively or iteratively. We showed the parallel FFT algorithm for the iterative approach by Kumar et al. [22]. We showed that this parallel algorithm experiences communication overhead that can be eliminated to some extent by exploiting data locality. We discussed our new localized algorithm by using swapping technique that brings data closer to memory. We implemented the algorithm on Beowulf cluster with a performance improvement of approximately 15% from the original butterfly.

Chapter 4

Multidimensional Matrix Representation

Matrix multiplication is a fundamental and important concept in linear algebra. Implementing matrix multiplication provides an understanding on how to implement more challenging linear algebraic problems on heterogeneous networks. Matrix arrays are used to represent data sets either in 2D or multidimensional arrays. Most research reported in literature has focused on simple 2D array representations [5, 12, 46]. Efficient representation using multidimensional arrays is considered as one of the most challenging and difficult problems.

4.1 Introduction

Linear algebra plays an important role in scientific and engineering applications. One of the fundamental concept in linear algebra is matrix array computations. Matrix arrays are used to represent data sets for various applications. There are several applications that require higher dimension matrices to represent large data sets. Many

engineering problems represent data in multidimensional arrays. Environmental scientists, for example, store satellite images of the earth in multidimensional arrays. They use these data to study climatic changes [20] and the simulations results [36] on such data create large amount of floating point numbers in the form of multidimensional arrays. Other applications that require large storage space and represent data in multidimensional arrays are in monitoring astronomical bodies in space [34], finance [47], computational chemistry [43], computational biology [2] and medical imaging [42]. Before going into a detailed study of the multidimensional array representation, a brief introduction to traditional matrix representation is given.

Let us consider a polynomial,

$$a_i = \sum_{1}^{n} x_{ij} y_j \tag{4.1}$$

where $i = 1, 2, ..., m$

The coefficient x_{ij} can be arranged in a rectangular array.

$$\begin{pmatrix} x_{11} & x_{12} & ..... & x_{1n} \\ x_{21} & x_{22} & ..... & x_{2n} \\ x_{31} & . & . & . \\ . & & & . \\ . & & & . \\ x_{m1} & x_{m2} & ..... & x_{mn} \end{pmatrix} \tag{4.2}$$

This array representation is referred to as matrix $(m * n)$, with rows *(m)* and columns *(n)*. This is the general 2D matrix representation. Figure 4.1 shows the mapping of an 2D array A[3][3] onto sequential memory locations. Mapping 2D arrays is relatively easier than mapping higher dimensional arrays.

Typically, the multidimensional matrices are represented as arrays using Traditional Matrix Representation (TMR) (Figure 4.2). There are some disadvantages

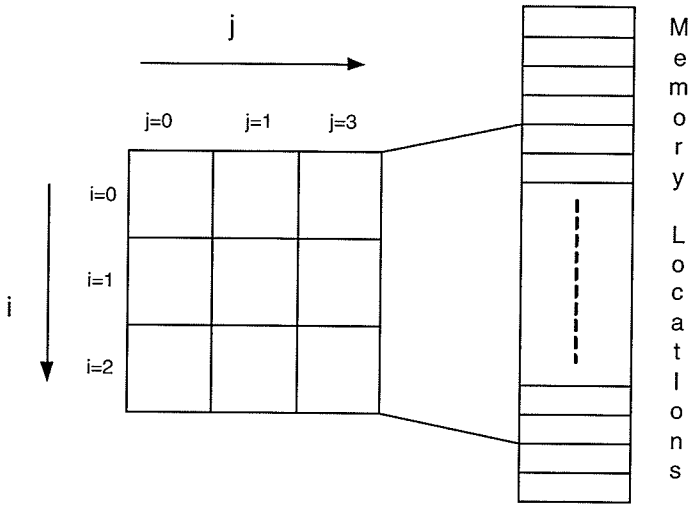

Figure 4.1: 2D Mapping.

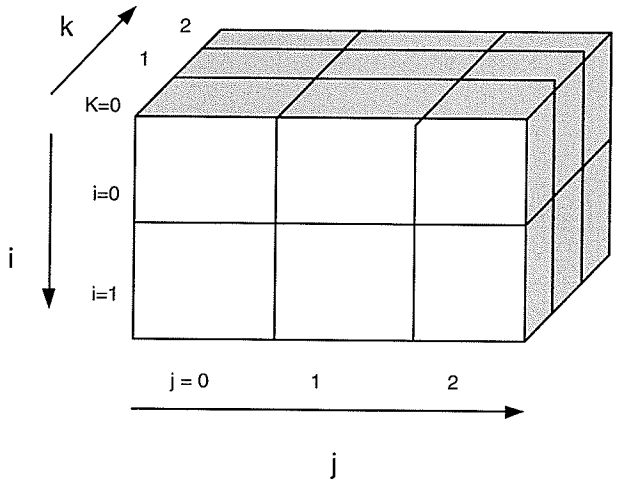

Figure 4.2: TMR 3x2x3 array representation.

in representing higher dimensional data using the TMR [32]. The main drawback, is that it increases the cost of index computations of the array elements. Another drawback, is that there is an increased cache miss rates for array computations [32]. Recently, a new approach has been proposed [31] for multidimensional arrays called Extended Karnaugh Map Representation (EKMR).

Fernando and Emilio [52], designed and studied the performance of a simple parallel matrix multiplication algorithm on both homogeneous and heterogeneous clusters. Their proposed data distribution scheme focuses on the various processing power of the heterogeneous machines. The data distribution algorithm performs more efficiently over the parallel algorithm provided by the ScaLAPACK Library [14]. Oliver et al. [4] developed an efficient load balancing algorithm for 2-D matrix multiplication on heterogeneous machines. A polynomial column-based heuristics was developed using a suitable load balancing strategy. The algorithms were tested on a heterogeneous network of workstations and on two smaller heterogeneous clusters having a myrinet link between them. Their algorithm was simulated using the standard MPI library.

J.R.de Souza et al. [19] implemented the 2-D matrix multiplication algorithm on a network of heterogeneous cluster of workstations by employing a master/worker (M/W) model. In the M/W model the master processor is in-charge of task allocation of optimum sized data and collection of resulting data from the worker processors. The algorithm was based on static, dynamic and serial data allocation schemes. Their algorithm was implemented in MPI, because of ease of portability over the inter-clusters processors. Based on the results from their experimentation, they proposed an efficient task scheduling algorithm model with a combination of both static and dynamic data distribution over heterogeneous networks.

Yuan Lin et al. [31, 32] developed an efficient way to represent the Multidimensional arrays by a set of 2-dimensional arrays. Their representation (EKMR) is based

on the Karnaugh map method. The EKMR scheme has been theoretically and experimentally proven to give better performance than the TMR scheme. The scheme has been implemented on a matrix multiplication algorithm and has been proven to be a cost effective approach for various array operations compared to the TMR. The authors have also studied the performance of the EKMR scheme for row, column and 2D meshes data distribution schemes on distributed memory multicomputers [14].

4.2 Extended Karnaugh Map Representation

In this section, we give a brief description of EKMR [32]. Using the Karnaugh map representation, the EKMR is developed. Karnaugh map (k-map) is used as a minimizing method employed in boolean algebra. The k-map is a simplification tool for logical expressions. The input variables forms the rows and columns of the k-map. The ordering of the input variables are based on the use of gray code. Figure 4.3 shows an example for up to 4-input k-map.

In the example, for the 1-input function, there are 2 possible input variables stored in the k-map. For the 2-input function (B:row, A:column), there are 4 input variables stored and 3-input function (C:row, BA:column) has 8 variables. It follows a similar pattern for higher number of input variables. We can see for n-input variables there are 2^n combinations. For $n = 1$ and $n = 2$ the EKMR follows the same method as in TMR. However for higher dimensions, the EKMR differs.

For example, let's consider a 3-D array $X[k][i][j]$ with dimensions i, j and k, where k^{th} dimension represents the 3^{rd} dimension of the array. Figure 4.4(a) shows the 2-D row major layout for the TMR. The placement of variables in the memory follows layout given by [31].

$$L_{RM}(k, i, j : 3, 2, 3) = k * 2 * 3 + i * 3 + j; \tag{4.3}$$

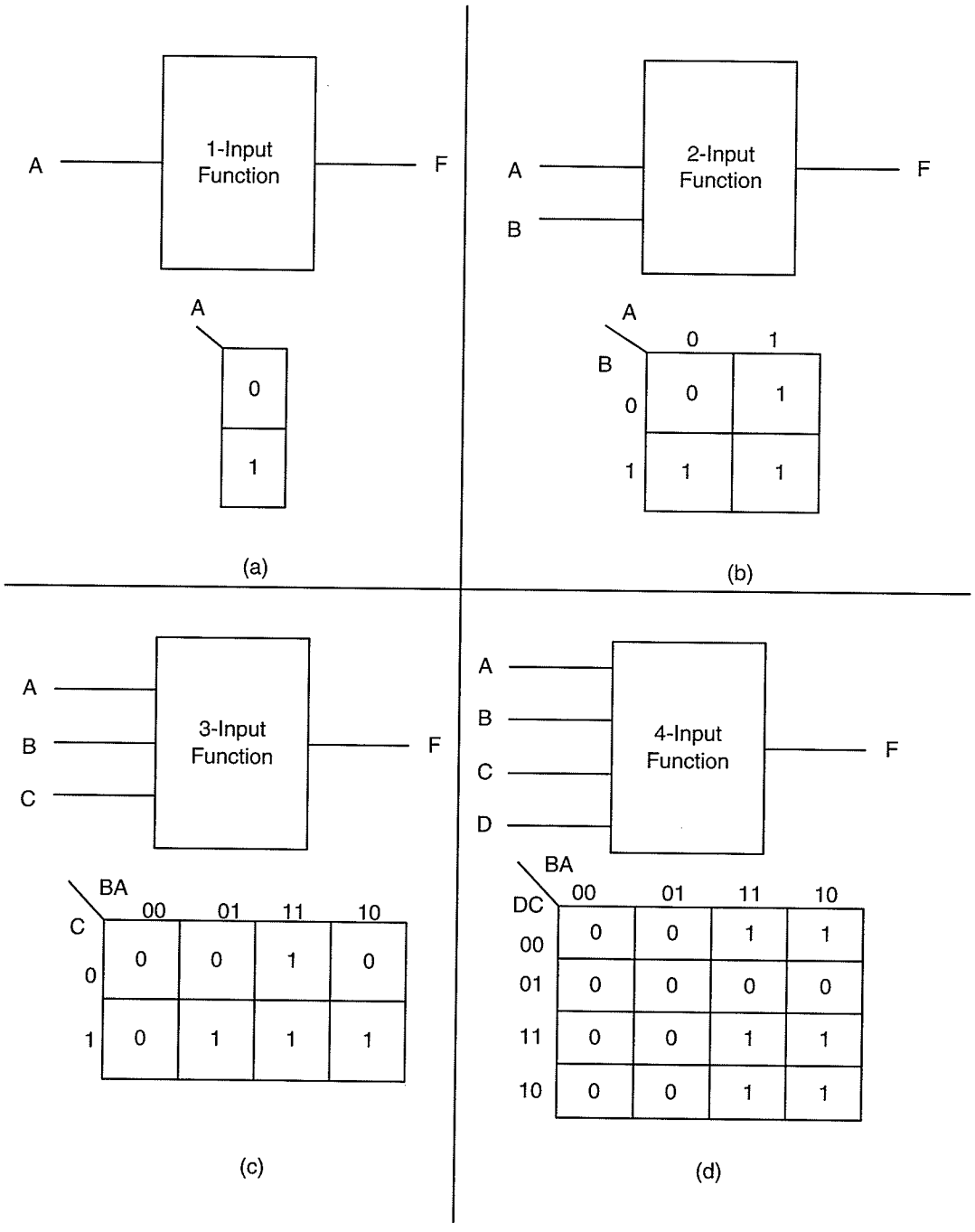

Figure 4.3: Karnaugh map examples: (a) 1-input function F = A, (b) 2-input function F = A+B, (c) 3-input function F = AB'+A'B, (d) 4-input function F = BD+BC'

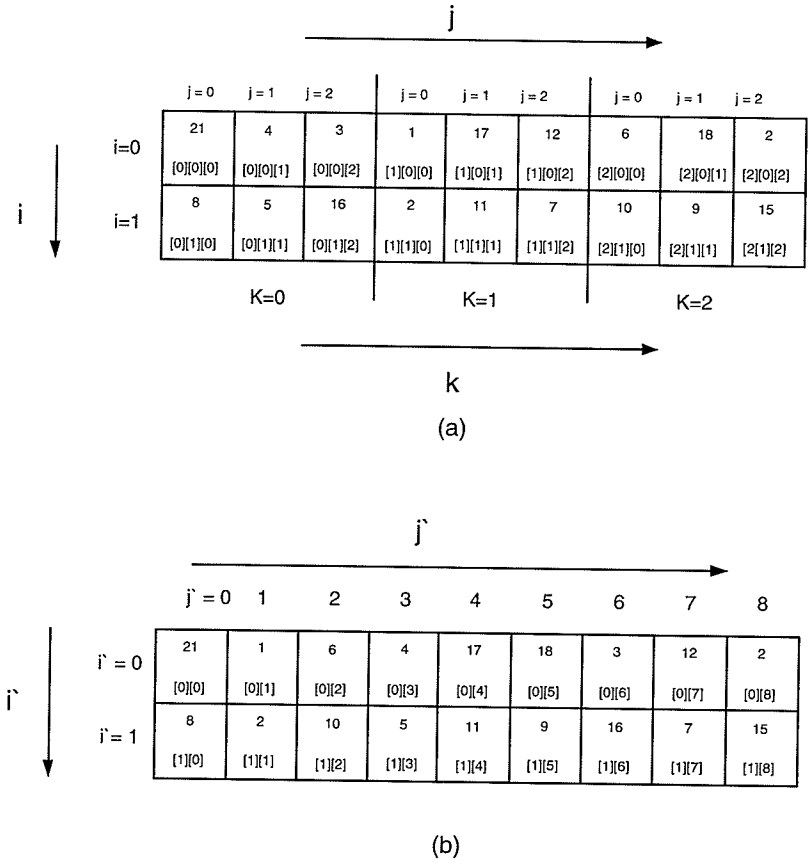

Figure 4.4: (a) TMR (b) EKMR

Figure 4.4(b) show the 2-D EKMR representation of the 3-D array. The size of the array is defined by the index variables $d*r*c$, where d is the third dimension and r and c represent row and column respectively. Let X' denote the two dimensional array in EKMR. We look into the row-major data layout. The mapping of TMR to EKMR [31] can be defined as:

$$L_{RM}(k,i,j:3,2,3) -> L'_{RM}(i',j':r,cd), \tag{4.4}$$

where, $i' = i; j' = j*d+k;$

Note that in EKMR, the array is a matrix unlike in TMR, where it is a three dimen-

sional matrix. In EKMR the array can be represented as $X[i'][j']$. For example, using equation(4.4), the array element $X[1][0][2]$ corresponds to $X'[0][7]$.

Retrieving and manipulating data in multidimensional arrays are very time consuming tasks, even in an efficient multidimensional representation of the arrays. One way to speedup the processing time is by using multiple computers to solve a given problem. The increase in low cost hardware technology, fast network performance and standard APIs are now paving the way to widespread use of clusters [10, 54]. Load balancing plays a very important role in obtaining good performance by reducing communication overhead and thereby increasing computational workload among the processors to avoid processor idling. On a heterogeneous network [51, 52] load balancing is a challenging issue. Allocating data and tasks to an individual processor depends on the processors speed, network bandwidth, memory constraints and the operating system.

EKMR has several advantages over the TMR in representing higher dimensional arrays. Therefore, EKMR seems to be an appropriate choice to test on heterogeneous machines for multi-dimensional array operations.

4.3 Performance on Shared Memory Machines

We first studied the performance results of the EKMR scheme to multidimensional matrix multiplication algorithm on a shared memory machine using OpenMP. The shared memory machine has 8 processors, each processor can handle 72 threads. One main advantage in shared memory programming, is that, the data partitioning, data distribution and data scheduling is handled by OpenMP scheduling policies.

OpenMP provides different types of scheduling, that is used to specify the division of the *for* loop interactions amongst the threads [11]. The different types of scheduling

are: *dynamic*, *static*, *guided* and *runtime* scheduling. Iterations are separated into different chunks, defined by the *chunk size*.

In static scheduling, chunks are statistically assigned to threads in a round robin fashion. If the chunk size is not defined in advance, the threads are assigned chunks that are approximately equal in size. In dynamic scheduling, the chunks are dynamically assigned to threads at runtime. If the chunk size is not specified, the default value is taken as 1 for all the chunks. In guided scheduling, the threads are assigned with chunk sizes that decreases exponentially. Threads are assigned chunks dynamically, until no chunks remain to be assigned. In runtime scheduling, the chunk size and schedule type is assigned at runtime specified by the value of the environment variable OMP_SCHEDULE.

We have tested the EKMR parallel algorithm with the simple static scheduling. The chunk size can be defined as N/P, where N is the trip count of the parallel loop and P is the number of threads that is executing the parallel loop. Figure 4.5 shows the execution times for TMR and EKMR with various number of threads. All the threads have approximately the same amount of work. From the graph, we can see that the execution time decreases as the number of threads are increased. The EKMR algorithm proves to have better performance in the shared memory machines.

4.4 Performance on Heterogeneous machines

The multidimensional matrix multiplication algorithm is implemented on a network of heterogeneous machines. The specifications of these machines are shown in Table 4.1. The interconnection network is a high speed 100BaseTX Ethernet(100 Mb/s) network. The MPI message passing standard was used for the parallel implementation.

We randomly created 3-D matrices of various sizes and stored them as 2-D matrices

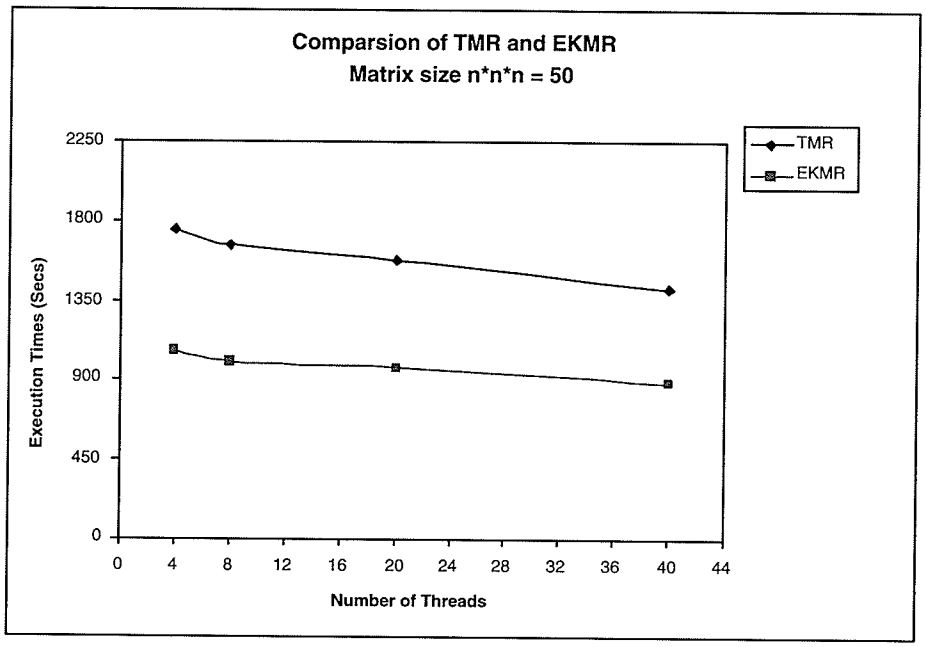

Figure 4.5: Execution Time for TMR and EKMR

Name	Operating System	System Description	Clock Rate	Main Memory	Quantity
Cadmium	Sun OS 5.8	UltraSparc 3	750 MHz	512 MB	7
Calcium	AIX 5.2	IBM RS/6000	400 MHz	256 MB	7
Canary	Red Hat Linux 7.2	Pentium III (Coppermine)	864 MHz	256 MB	8
Linux 16	Red Hat Linux 9.0	Intel(R) Pentium(R)III	1000MHz	512 MB	16

Table 4.1: Heterogeneous Workstations

using the EKMR. Given three matrices X, Y, and Z, the algorithm computes $X * Y = Z$. Using row major layout, the matrix is represented in 2-D $X'[i'][j']$, where i' and

j' are as in equation 4.4. The matrix Y is broad casted to all the processors, and the rows of matrix X' are evenly distributed among the processors. We experimented the algorithm with both static and dynamic load balancing strategies.

4.4.1 Static Distribution

In this scheme, a manager/worker model is applied. The manager initially distributes an equal amount of data to the other processors (workers) before the execution of the algorithm. This is a one time partition. The manager is responsible for distributing and collecting tasks among the workers.

4.4.1.1 Dedicated Machines

For our first experimentation, we used 7 Sun OS, 7 AIX and 7 Red Hat 7.2 machines as workers. One of the Linux machine is selected as the manager, since it's processor speed is faster. This model is shown is Figure 4.6. We call these machines dedicated machines because we were the sole users of these machines. We wanted to determine the efficiency of these machines before involving other machines into the group.

The manager partitions the data statistically among the workers initially. Figure 4.7 shows the execution time of the algorithm for various matrix sizes. As we can see, the execution time tends to increase with the size of the problem. This is a normal occurrence, since the workload among the processors increases with size. However, when compared with the parallel algorithm with TMR on heterogeneous machines [Table 4.2], we see that the parallel algorithm with the EKMR technique does better on heterogeneous machines. For a matrix of size 30x30x30, the execution time for the parallel algorithm with EKMR is only 1.5 secs compared to 30 secs for the parallel algorithm with TMR.

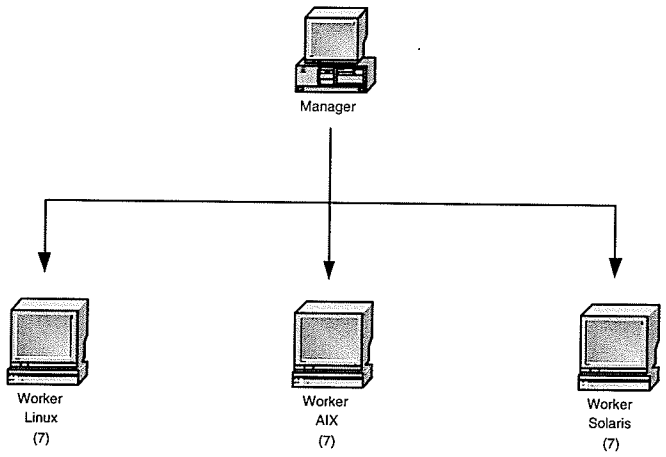

Figure 4.6: Static Data Distribution for Manager/Worker on 21 processors

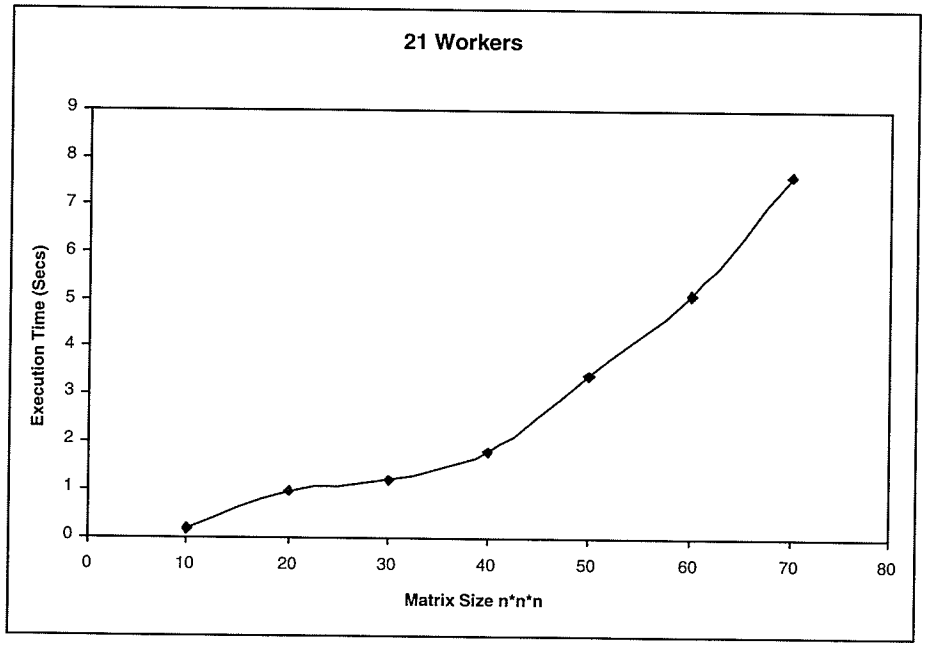

Figure 4.7: Problem Size Vs. Execution Time (Dedicated machines)

Matrix Size	Execution Time(Sec)
5*5*5	0.19503
10*10*10	3.97537
15*15*15	4.23434
20*20*20	13.83291
25*25*25	24.37566
30*30*30	30.184244

Table 4.2: Timing Details for TMR

4.4.1.2 Non-dedicated machines

In this experimentation, we used all the 38 processors, allowing the Linux machine to be the manager (Figure 4.8). The 16 extra Linux machines are non-dedicated machines available in the Computer Science laboratories. These machines are not solely for our purposes but is used by the graduate and undergraduate students for their courses. So, we created a host file in which these Linux machines were included with the other dedicated machines. This host file is very flexible. We can add as many machines as possible as long as we have MPI running on these machines.

As in the experimentation above, the manager partitions the matrix equally among the 37 workers. This is an unfair distribution, since the number of processors for Linux is higher and the speed of Linux is also faster. So, logically we will conclude that the performance of this experimentation should be better than Figure 4.7. However Figure 4.9 shows otherwise.

The reason behind this performance degradation is that the 16 Linux machines are non-dedicated machines. Also, for such an environment MPI is not appropriate [29]. We also notice that from 60*60*60 to 70*70*70 matrix sizes (Figure 4.9), the execution

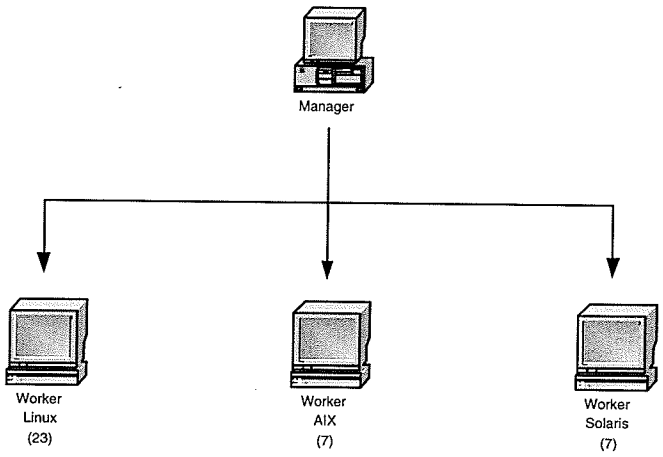

Figure 4.8: Static Data Distribution for Manager/Worker on 37 processor

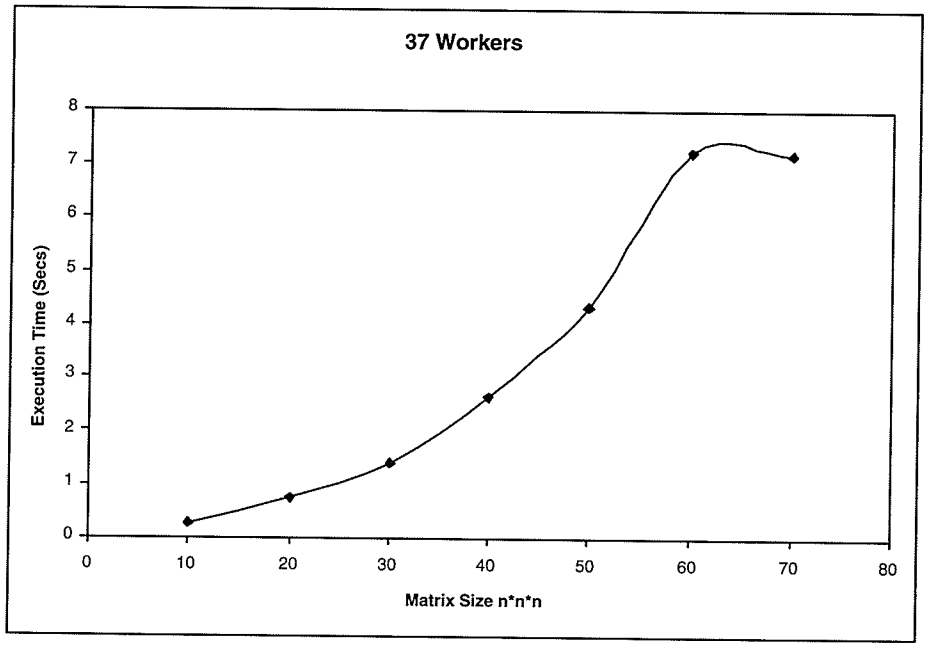

Figure 4.9: Problem Size Vs. Execution Time (Non-Dedicated machines)

time is flat. This experimentation leads us to conclude that maybe MPI is not the right paradigm for heterogeneous machines with a static load balancing scheme. In the next section we develop a dynamic load balancing scheme.

4.4.2 Dynamic Distribution

In this scheme, we have one manager who has the entire dataset (Figure 4.10). The manager allocates work to the processors on-demand. There is a submanager who is in charge of obtaining data from the central manager and distributes this amongst the its workers. The manager gives out work on demand. The submanager asks the manager for X number of rows, where X is the number of worker within its group. Then the submanager distributes this to the X workers. As the computation proceeds, the submanager asks the manager for more work.

Submanager	Time(Sec)
LINUX	0.731417
AIX	0.686881
SOLARIS	0.514936

Table 4.3: Timing Details

Note that, we only experiment with dedicated machines, since we obtained fluctuating results with non-dedicated machines. This is understandable since the non-dedicated machines were fully utilized by other users.

We needed to determine the time it takes for each group of computers to finish its computation allocated to it. This timing is reported in Table 4.3 As we see the Solaris machine is the fastest and the Linux is the slowest. This is understandable, since Linux has more workers. The time for Linux machines to finish their computations, report

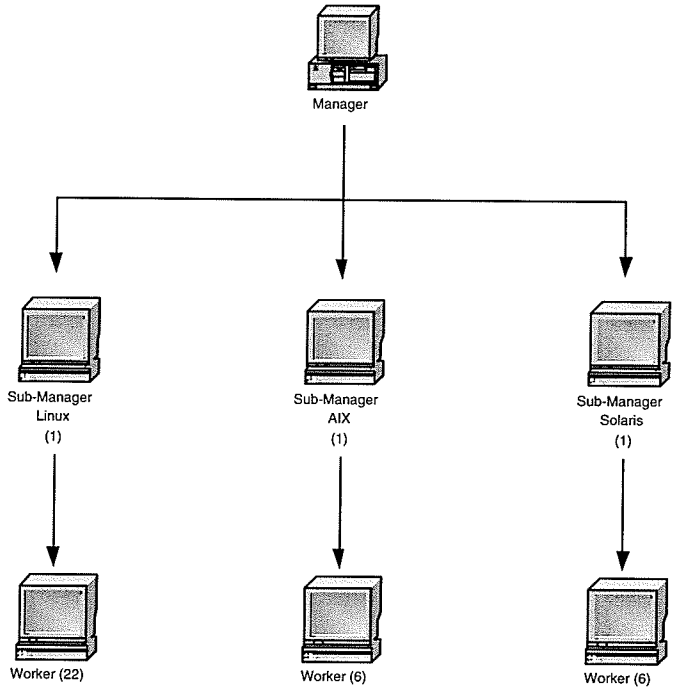

Figure 4.10: Dynamic Data Distribution for Master/Submanager/Worker for n*n*n matrices

to submanager and allow submanager to request more data is significant compared to AIX (7 processors) or SOLARIS (7 processors). This gives us a better understanding of the machines and indicates that giving more work to Linux machines may not necessarily degrade the performance of the algorithm.

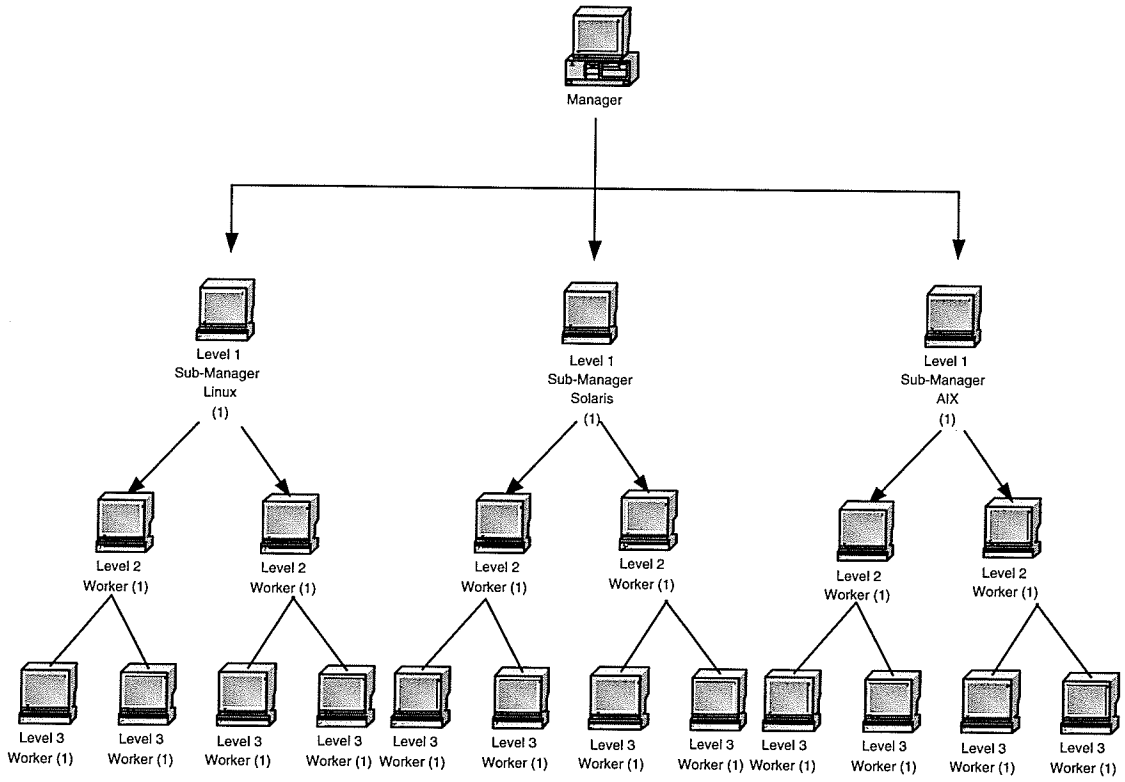

Figure 4.11: Dynamic Data Distribution

With the values determined from table 4.3, we developed a hierarchical worker/manager model for data distribution, shown in Figure 4.11. As with the previous schemes, the Linux machine is the manager. Level 1 has the sub-managers for providing work to the workers in the lower levels. Each of the sub-managers handle two workers, and these workers in turn handles two workers. During the initial distribution, Linux sub-manager receive 30% of the input rows, Solaris sub-manager recieve 20% of the input rows and AIX sub-manager receive 10%. The rest of the rows are stored in the

main Linux manager. Once the sub-managers recieves the rows, it retains 10% of the rows to work on and 10% of rows for future requests. The rest of the rows are sent to the lower level workers. Level 2 workers retains 10% of the rows to work on and 10% of rows for future requests. Level 3 workers do not store any rows.

After the initial distribution, level 3 request more rows to immediate upper level. If level 2 has no rows to send, it in turns requests level 1 manager for more work. Once the work has been completed in all levels and if the main manager has no more rows to distribute, the computation terminates.

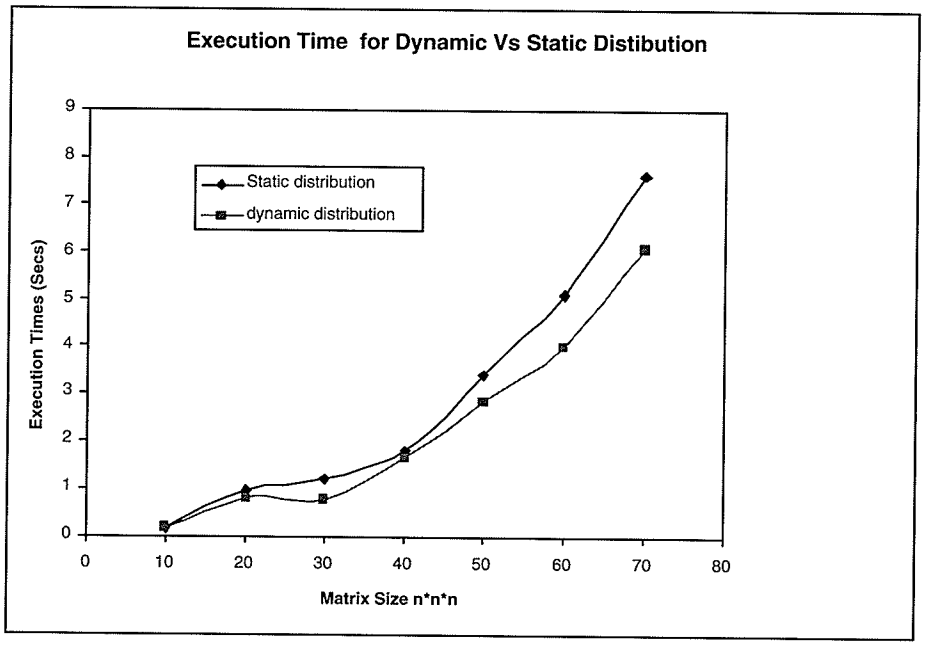

Figure 4.12: Problem Size Vs. Execution Time for Dynamic Data Distribution

From the Figure 4.12, the execution time of the dynamic distribution is better than that of the static distribution. For smaller matrix sizes, the execution times of the static and dynamic distributions are very close. As the data size increases, the dynamic distribution performs better.

4.5 Summary

In this chapter, a brief introduction to the traditional matrix representation is given. The multidimensional EKMR [32] method was described in detail. We implemented the parallel multidimensional matrix multiplication algorithm with the EKMR representation on shared memory machines using static scheduling. The results were impressive. The parallel algorithm was also implemented on heterogeneous machines using static and dynamic load balancing strategies. We tested the algorithm with dedicated and non-dedicated machines. For dynamic load balancing scheme, we developed a hierarchical manager-worker model. The results in this chapter indicate that heterogeneous machines are a good paradigm to consider for future applications. However, MPI may not be the programming model to be used since the partitioning strategies are left to the user with consideration of the underlying architecture. This is quite tedious.

Chapter 5

Conclusion

In this thesis, we have studied the effectiveness of developing parallel algorithms on distributed heterogeneous and cluster computing environments. The parallel algorithm design, implementation and performance analysis issues have been studied. On a distributed network environment, one of the crucial issues in gaining performance is efficiently mapping the problem onto the processors, so as to reduce communication latency. We would like to place tasks that communicate frequently in the same processor and at the same time minimize idle time of the processors. These are the two main objective in mapping. These are conflicting and difficult objectives to maintain in a problem. Load balancing technique try to solve the mapping problem to some extent. However, load balancing strategies differ depending on the application.

Recently cluster and heterogeneous computing have gain popularity. Clusters can be built from low cost commodity components and interconnected through a high speed network. Heterogeneous machines vary in processing speed, operating system and underlying hardware. Both cluster and heterogeneous computers can be classified as network of computers. Computationally intensive applications are suitable on such machines. However, not all problems are computationally intensive. There maybe

some interaction in the concurrent tasks in the problem.

In this thesis, we have studied two fundamental scientific computing problems, the Fast Fourier Transform and multidimensional matrix multiplication algorithm on a cluster and heterogeneous computing environments.

The FFT is an ubiquitous problem since it has many applications such as in image processing, signal processing, medical imagining and recently computational finance. From a parallel algorithm perspective, the FFT can be regarded as a semi-irregular dynamic and synchronous problem. The problem is semi-irregular because there are two inherent latency issues, communication and synchronization. We apply domain decomposition on N data points and distribute $\frac{N}{P}$ data points to the P processors. Two data points are combined at an iteration i , their binary bits differ in the i^{th} bit. The termination condition of the parallel algorithm is simple if we follow this partitioning strategy. The algorithm terminates after $log\ N$ iterations, where the first $log\ N$ - $log\ P$ iterations are sequential and the last $log\ P$ iterations require remote communication.

We try to minimize communication by exploiting data locality. At each iteration, we swap $\frac{N}{2P}$ data between two processors. This swapping technique produces better results since we exchange chunks of data in one send operation as opposed to sending one data at a time, which is costly in MPI. When compared to the algorithm by Kumar et al [22] our parallel algorithm produces 15% speedup. We implemented our parallel algorithm on a beowulf cluster.

Many application utilize multidimensional arrays as data structures including multidimensional FFT. Matrix multiplication algorithms have been used as benchmark applications in the scientific community. Representing multidimensional arrays efficiently is one of the important steps in gaining good speedup. Since it requires lots of memory space. We use the Extended Karnaugh Map Representation (EKMR)

technique to represent multidimensional arrays. We then consider implementing multidimensional matrix multiplication algorithm on heterogeneous machines. We first experimented with the machines to determine the characteristics and features of these machines. We applied two types of load balancing strategies, static and dynamic. Dynamic load balancing produces better results than the static strategy. This is because the dynamic strategy uses a on-demand approach. The manager is responsible for allocating tasks to its submanager on-demand. The submanagers in turn divide work amongst its workers. This is a hierarchical manager-worker model. In the static approach, all workers were given the same amount of work. This is not efficient, since the slowest processors degrade the performance, while the fastest processors sit idle once the work has been accomplished. Therefore, on a heterogeneous machine, since we had to consider the feature of all machines, it was very tedious to come up with an efficient load balancing strategy.

Our experience indicates that we need a better way of designing parallel algorithms on heterogeneous machines. MPI is not very efficient since it does not consider the underlying features of the machines. Recently a new language, mpC [28] has been developed which addresses issues related to heterogeneous computing environments.

Another issue that one needs to worry about is in performance evaluation of parallel algorithms on these environments. For future work, parallel algorithm design using mpC should be investigated. The above two studies form fundamental to many problems in science, engineering, finance and social research areas. Hence, use of the results from this study has wide applicability.

Bibliography

[1] Arvind and Robert A. Iannucci. Two fundamental issues in multiprocessing. In *4th International DFVLR Seminar on Foundations of Engineering Sciences on Parallel Computing in Science and Engineering*, pages 61–88. Springer-Verlag New York, Inc., 1988.

[2] David Bader. Computational biology and high performance computing. *Communications of the ACM*, 47(11):35–41, Nov 2004.

[3] Kenneth E. Batcher. Design of a Massively Parallel Processor. *IEEE Transactions on Computers*, 29(9):836–840, September 1980.

[4] Oliver Beaumont, Vincent Boudet, Fabrice Rastello, and Yves Robert. Matrix Multiplication on Heterogeneous Platforms. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1033–1051, October 2001.

[5] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*, May 1997.

[6] Tom Blank. The MasPar MP-1 architecture. In *Proceedings CompCon*, Feburary 1990.

[7] Robert D. Blumofe, Christopher F. Joerg, Bradley Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 207–216, Santa Barbara, California, July 1995.

[8] Bolt, Beranek, and Newman. www.bbn.com.

[9] S. M. Brunett, J. Thornley, and M. Ellenbecker. An initial evaluation of the Tera multithreaded architecture and programming system using the C3I parallel benchmark suite. In *Proceedings of the Supercomputing '98*, pages 1–19, Orlando, Florida, November 1998.

[10] Rajkumar Buyya. *High Performance Cluster Computing: Architectures and Systems*, volume 1. Prentice Hall PTR, Upper Saddle River, New Jersey, 1999.

[11] Rohit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. *Parallel Programming in OpenMP.* Morgan Kaufmann, San Francisco, CA, 2001.

[12] Amit Chhabra, Parimala Thulasiraman, Mohammad Towhidul Islam, and Ruppa K. Thulasiram. An OpenMP implementation of FTCS method for reduced black-scholes equation. In *The 18th International Conference on High Performance Computing Systems and Applications*, Winnipeg, MB, May 2004.

[13] Chi-Hsiang Yeh. $at^2l^2 \approx n^2/2$ for fast Fourier transform on multilayer VLSI. In *14th annual ACM symposium on parallel algorithms and architectures*, pages 145–146, Winnipeg, Canada, Aug. 2002.

[14] Jaeyoung Choi, Jack J. Dongarra, Roldan Pozo, and David W. Walker. Scalapack: A scalable linear algebra library for distributed memory concurrent computers. In *Fourth Symposium on the Frontiers of Massively Parallel Computation*, pages 120–127, McLean, VA USA, 1992.

[15] J.W. Cooley, P.A. Lewis, and P.D. Welch. The Fast Fourier Transform and its application to time series analysis. In *statistical methods for digital computers*, Wiley,New York, 1977.

[16] NCUBE Corporation. Ncube users handbook, 1987.

[17] David E. Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel Computer Architecture : A Hardware/Software Approach.* Morgan Kaufmann Publishers, San Francisco, 1998.

[18] Thompson C. D. Fourier transform in VLSI. In *IEEE Transcations on Computers*, pages 1047–1057, 1983.

[19] J. R. de Souza, A. Furtado, A. Reboucas, D. Rexachs, and E. Luque. Efficient algorithm execution in a collection of HNOWS. In *International Conference on Computer Science,Engineering Software, Information Technology, E-business and Application*, Brazil, 2002.

[20] J. Dozier and H. K. RamaPriyan. Planning for the EOS data and information system. In *Global Environment Change*, volume 1. Springer-Verlag, 1991.

[21] M. Goto and Y. Muraoka. A beat tracking system for acoustic signals of music. In *Proceedings of the second ACM international conference on Multimedia*, pages 365–372. ACM Press, 1994.

[22] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. *Introduction to Parallel Computing.* Pearson Education Limited, Harlow, England, 2003.

[23] Herbert H. J. Hum, Kevin B. Theobald, and Guang R. Gao. Building multithreaded architectures with off-the-shelf microprocessors. In *Proceedings of the 8th IEEE International Parallel Processing Symposium*, pages 288–294, Cancun, Mexico, 1994.

[24] IBM-Seaborg. http://www.nersc.gov/nusers/resources/sp/.

[25] Intel Corporation. *PARAGON OSF/I, Users guide*, April 1993.

[26] L. H. Jamieson, E. J. Delp, J. N. Patel, C.C. Wang, and A. A. Khokhar. A library-based program development environment for parallel image processing. In *Scalable Parallel Libraries Conference*, pages 187–194, Mississippi State University, Mississippi, October 1993.

[27] R.A. Kamin and G.B. Adams. Fast Fourier Transform algorithm design and tradeoffs on the CM-2. In *Proceedings of the 1987 Workshop on computer architecture for pattern analysis and machine intelligence*, pages 184–191, October 1987.

[28] Alexey Lastovetsky. *Parallel Computing on Heterogeneous Networks.* John Wiley and Sons, New York, 2003.

[29] Alexey Lastovetsky and Ravi Reddy. HMPI:towards a message-passing library for heterogeneous networks of computers. In *Proceedings of the International Symposium on Parallel and Distributed Processing*, 2003.

[30] Charles E. Leiserson, Zahi S. Abuhamdeh, David C. Douglas, Carl R. Feynman, Mahesh N. Ganmukhi, Jeffrey V. Hill, W. Daniel Hillis, Bradley C. Kuszmaul,

Margaret A. St Pierre, David S. Wells, Monica C. Wong-Chan, Shaw-Wen Yang, and Robert Zak. The network architecture of the Connection Machine CM-5. *Journal of Parallel and Distributed Computing*, 33(2):145–158, 1996.

[31] Chun-Yuan Lin, Jen-Shiuh, and Yeh-Ching Chung. Efficient representation scheme for multidimensional array operations. *IEEE Transactions on Computers*, 51(3):327–345, March 2002.

[32] Chun-Yuan Lin, Jen-Shiuh, and Yeh-Ching Chung. Efficient data parallel algorithms for multidimensional array operations based on the EKMR scheme for distributed memory multicomputer. *IEEE Computer Society*, 14(7):625–639, July 2003.

[33] Jen-Shiuh Liu, Jiun-Yuan Lin, and Yeh-Ching Chung. Efficient parallel algorithms for multi-dimensional matrix operations. In *Proceedings of the International Symposium on Parallel Architectures, Algorithms and Networks*, pages 224–229, 2000.

[34] Ying Liu, Wei keng Liao, and Alok Choudhary. Design and evaluation of a parallel hop clustering algorithm for cosmological simulation. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03)*, pages 82–89. IEEE Computer Society, 2003.

[35] C.L. Loan. Computational frameworks for the fast fourier transform. In *SIAM Journal*, Frontiers in Applied Mathematics, 1992.

[36] C. R. Mechoso, C. C. Ma, J.D. Farrare, J. A. Spahr, and R. W. Moore. Parallelization and distribution of a coupled atmosphere-ocean general circulation model. In *Monthly Weather Review*, volume 121, pages 2062–2076, 1992.

[37] Tera MTA. www.tera.com.

[38] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads Programming: A POSIX Standard for Better Multiprocessing.* O'Reilly, Cambridge, MA, 1996.

[39] Scott Oaks and Henry Wong. *Java Threads.* O'Reilly, Cambridge, MA, 1999.

[40] A.V. Oppenheim and A.S. Willsky. *Signals and Systems.* Prentice Hall, Englewood Cliffs,New Jersey, 1983.

[41] Kendall Square Research. KSR-1 principles of operation, 1991.

[42] Alexander E. Souvorov, Alexander E. Bulyshev, Serguei Y. Semenov, Robert H. Svenson, Alexei G. Nazarov, Yuri E. Sizov, and Geortge P. Tatsis. Microwave tomography: A two dimensional newton iterative scheme. *IEEE Transactions on Microwave Theory and Techniques*, 46(11):1654–1659, Nov 1998.

[43] T. P. Straatsma and J. A. McCammon. Load balancing of molecular dynamics simulation with nwchem. *IBM Syst. J.*, 40(2):328–341, Febuary 2001.

[44] Supercomputing Technologies Group, MIT Laboratory for Computer Science. *Cilk 5.3.2 Reference Manual*, November 2001. url: http://supertech.lcs.mit.edu/cilk/manual-5.3.2.pdf.

[45] SGI Technical Publications. *SGI - C Language Reference Manual*, June 2003.

[46] Kevin Theobald, Rishi Kumar, Gagan Agarwal, Gerd Heber, Ruppa Thulasiram, and Guang R. Gao. Implementation and evaluation of a communication intensive application on t he earth multithreaded system. *Concurrency and Computation: Practice and Experience*, 14(3):183–201, 2002.

[47] Ruppa K. Thulasiram and Dmitri Bondarenko. Performance evaluation of parallel algorithms for pricing multidimensional financial derivatives. In *Proceedings of the Fourth International Workshop on High Performance Scientific and Engineering Computing with Applications (HPSECA 2002)*, pages 306–313. IEEE Computer Society, August 2002.

[48] Ruppa K. Thulasiram and Parimala Thulasiraman. A parallel FFT approach for derivative pricing. In *Proceedings of SPIE, Commercial Applications for High Performance Computing (Ed: H.J.Siegel)*, pages 181–192, Denver, CO, August 2001.

[49] Ruppa K. Thulasiram and Parimala Thulasiraman. Performance evaluation of a multithreaded Fast Fourier Transform algorithm for derivative pricing. *The Journal of Supercomputing*, 26(1):43–58, August 2003.

[50] Parimala Thulasiraman, Kevin B. Theobald, Ashfaq A. Khohkar, and Guang R. Gao. Multithreaded algorithms for the Fast Fourier Transform. In *symposium on Parallel Algorithms and Architectures (SPAA)*, Bar Harbor, Maine, June 2000.

[51] Fernando G. Tinetti, Antonio A.Quijano, and Armando De Giusti. Heterogeneous networks of workstations and SPMD scientific computing. In *International Workshop on Parallel Processing*, pages 338–342, Aizu-Wakamatsu, Japan, 1999.

[52] Fernando G. Tinetti and Emilio Luque. Efficient broadcasts and simple algorithms for parallel linear algebra computing in clusters. In *Proceedings of the International Parallel and Distributed Processing Symposium*, page 8pp., 2003.

[53] Guy Tremblay. Threaded-C Release 2.0: Motivation, Description, and Rationale. Technical Report CAPSL, Computer Architecture and Parallel Systems Laboratory, Technical Note 09, University of Delaware, Newark, DE, USA, 2000.

[54] Barry Wilkinson and Michael Allen. *Parallel Programming Techniques and Applications Using Networked Workstations and Parallel Computers.* Prentice Hall, Upper Saddle River, New Jersey, 1999.

[55] Chi-Hsiang Yeh and Behrooz Parhami. A class of parallel architectures for fast fourier transform. In *Proceedings of the Midwest Symposium Circuits and Systems*, pages 856–859, IEEE catalog, 1996.

[56] Chi-Hsiang Yeh, Emmamoel A. Varvarigos, and Behrooz Parhami. Efficient VLSI layouts of hypercubic networks. In *Proceedings Symposium Frontiers of Massively Parallel Computations*, pages 98–105, 1999.

[57] Albert Y. H. Zomaya. *Parallel and distributed computing handbook.* McGraw-Hill, New York, 1996.