DEVELOPMENT OF HYBRID MPI+UPC PARALLEL PROGRAMMING MODEL

Elif ÖZTÜRK

KADIR HAS UNIVERSITY

2011

DEVELOPMENT OF HYBRID MPI+UPC PARALLEL PROGRAMMING MODEL

ELİF ÖZTÜRK

B.S., Computer Engineering, Kadir Has University, 2007

M.S., Computer Engineering, Kadir Has University, 2011

Submitted to the Graduate School of Kadir Has University
in partial fulfillment of the requirements for the degree of
Master of Science
in
Computer Engineering

KADIR HAS UNIVERSITY

2011

KADIR HAS UNIVERSITY

GRADUATE SCHOOL OF SCIENCE AND ENGINEERING

DEVELOPMENT OF HYBRID MPI+UPC PARALLEL PROGRAMMING MODEL

ELİF ÖZTÜRK

APPROVED BY:

Asst. Prof. Dr. Zeki Bozkuş _____
(Thesis Supervisor)

Asst. Prof. Dr. Taner Arsan _____

Prof. Dr. Selim Akyokuş _____

APPROVAL DATE: 01/07/2011

DEVELOPMENT OF HYBRID MPI+UPC PARALLEL
PROGRAMMING MODEL

Abstract

Parallel Computing is a form of computation that divides a large set of calculations into tasks and runs on multi-core machines simultaneously. Today, Message Passing Interface (MPI) is the most widely used parallel programming paradigm that provides programming both for symmetric multi-processors (SMPs) which consists of shared memory nodes with several multi-core CPUs connected to a high speed network and among nodes simultaneously. Unified Parallel C (UPC) is an alternative language that supports Partitioned Global Address Space (PGAS) that allows shared memory like programming on distributed memory systems.

In this thesis, we describe the MPI, UPC and hybrid parallel programming paradigm which is designed to combine MPI and UPC programming models. The aim of the hybrid model is to utilize the advantages of MPI and UPC; these are, MPI's data locality control and scalability strengths with UPC's global address space, fine grain parallelism and ease of programming to achieve multiple level parallelism. This thesis presents a detailed description of hybrid model implementation comparing with pure MPI and pure UPC implementations. Experiments showed that the hybrid MPI+UPC model can significantly provide performance increases up to double with pure UPC implementation and up to 20% increases in comparison to pure MPI implementation. Furthermore, an optimization was achieved which improved the hybrid performance an additional 20%.

HİBRİD MPI+UPC PARALEL PROGRAMLAMA
MODELİNİN GELİŞTİRİLMESİ

Özet

Paralel Hesaplama geniş hesap kümelerini görevlere bölen ve bu görevleri çok çekirdekli makinelerde aynı anda çalıştırmaya yarayan hesaplama biçimidir. Bugün, Message Passing Interface (MPI - Mesaj Gönderme Arayüzü) ortak hafıza noktaları ile birçok çok çekirdekli işlemcinin yüksek hızlı bir networke bağlanmasından oluşan simetrik çoklu işlemciler (SMP) ve noktalar (nodes) arasında aynı anda programlamayı sağlayan ve kullanılan en geniş paralel programlama paradigmasıdır. Unified Parallel C (UPC) dağıtık adresli sistemleri ortak hafızalı sistemler gibi programlamaya izin veren Bölünmüş Global Adres Alanı'nı (PGAS) destekleyen alternatif bir dildir.

Bu tezde, MPI, UPC, MPI ve UPC programlama modellerini birleştirmek için tasarlanan hibrid paralel programlama paradigması anlatılmıştır. Hibrid modelin amacı MPI ve UPC' nin avantajlarından faydalanmaktır. Bunlar MPI modelin yerel data kontrolü ve ölçeklenebilirliği ile UPC modelin global adres alanı, ince taneli paralellik ve çoklu seviye paralellik sağlamak için programlama kolaylığı özellikleridir. Bu çalışma hibrid model uygulamasını yalnız MPI ve yalnız UPC uygulamaları ile karşılaştırmak suretiyle ayrıntılı açıklama sunmaktadır. Deneyler hibrid MPI+UPC modelin önemli ölçüde yalnız UPC uygulaması ile iki kat ve yalnız MPI uygulamasında %20' ye kadar performans artışı sağlayabildiğini göstermiştir. Ayrıca hibrid performansı geliştiren bir optimizasyonda ek olarak %20 iyileşme kazanılmıştır.

Acknowledgements

I would like to present my sincere regards to those who encouraged me most to achieve this graduation, which happens to be of great value to me.

First of all, I would like express my deep-felt gratitude to Assistant Professor Taner Arsan, the head of Computer Engineering Department at Kadir Has University, for all the support he provided me through out my all academic and graduate career. Also, I would like to thank my thesis advisor Assistant Professor Zeki Bozkuş for all his expert guidance on my project, motivation and, support and to Assistant Professor Atilla Özmen for all his support and understanding along my assistantship.

I would like to thank my dear friends lecturer Canan Cengiz, my Assistant friend Ecem Sezenler and my all other colleagues who motivated and supported me at every stage of my career.

Finally; I would like to thank to my mother and father who brought me to this day.

Table of Contents

List of Tables

List of Figures

Chapter 1
Introduction

Recently, there is a growing demand for parallel models of High Performance Computing (HPC) infrastructures for solving operations large data sets because of their scalability and performance. Parallel hardware and software technologies are serving to this demand however users had to effort to find the best suited programming paradigm for the underlying computer architecture. Most people are interested in hybrid models that merge the advantages of two different models. There is many examples of models that combines MPI library with shared memory model OpenMP.

Message Passing Interface is the most commonly used parallel programming model for parallel computing [9]. MPI is usually used on distributed memory and this model provides portability, good scalability and significant flexibility in parallel programming. However, today MPI is used by many scientific applications, MPI requires explicit communications with large granularity which renders programming and programming problematic. Partitioned Global Address Space (PGAS) languages supports that a single program can be able to run across the shared and distributed memory features of the machine [1]. One of the most popular PGAS language is Unified Parallel C (UPC) is a parallel programming language which facilitate us to use the distributed memory as the shared memory and save us from using explicit communication via simplified statements like read/ write to remote memory.

This study aimed to exploit the complementary strengths of both models by providing a hybrid model that combines MPI and UPC models. This hybrid model reduces the communications overhead by lowering data movements within nodes. In addition the goal of this hybrid model is to offer the fine granularly parallelism of the

UPC, partitioned address space and it's benefit of simplified programming. Hybrid model adds the strengths of the MPI's good scalability, portability and coarse grain parallelism with a larger message size. The recent trend in high performance computer architecture is to increase cores on nodes and hence decrease the memory per core at nodes, consequently encouraging us to explore different programming paradigms such as an MPI+UPC hybrid on a large scale distributed platform.

In this study, we presented and described a new hybrid parallel programming model that combines advantages of MPI and UPC model to increase the performance and scalability of operations comparing with pure MPI and pure UPC implementations. We selected a funneled approach for our hybrid model, meaning that all interactions between the MPI and UPC are controlled by a master thread and only the master thread calls MPI routines. In this manner, all communication process between nodes is made by a single thread-master thread- of the node, so we can say all communication is funneled to the master thread.

In this thesis, we developed three codes to probe the efficiency and scalability of the models on distributed multi-core systems with the Cannon matrix multiplication algorithm, which was chosen to exploit some of the advanced features of MPI. We used MPI virtual topology to benefit from regional locality, as the UPC has only local or shared (global) objects and in this way the hybrid model will enhance UPC program performance with regional locality. In addition, we utilized an optimization to overlap MPI communications with UPC computations on the hybrid model and this optimized benchmark performance up to 30% on some data sets.

The rest of the thesis organized as follows. Chapter 2 presents an overview of MPI and UPC parallel programming models on parallel platforms, briefly describing their strengths, weaknesses and most known implementations, and Chapter 3 examines hybrid models and describes the funneled model in further detail by the

implementation. In Chapter 4 we give an overview of memory types that parallel programming is being utilizing and we used for this study and Chapter 5 presents detailed explanation of Cannon matrix multiplication with a sample example of multiplication with figures. Chapter 6 explains four different implementations of the Cannon's Algorithm that we used for our experiments in Chapter 7. Related Work and Conclusion are presented in Chapter 8 and 9 respectively.

Chapter 2
Overview of MPI and UPC

This chapter provides an overview of MPI and UPC parallel programming model and examines the advantages and disadvantages of each model for the construction of a hybrid programming model.

2.1 MPI Model

A group of computer vendors, computer science researchers, application scientists from government laboratories, universities and industry came together at a workshop and decided to cooperate for the message passing model about parallel computing in April 1992. Sixty people from forty organizations attended MPI standardization effort and these people were mainly from United States and Europe. As a result of this workshop MPI Forum is emerged and accepted a primary model of high performance computing environment. MPI (Message Passing Interface) is resulted from these deliberations and now it is synonymous with the parallel computing model itself. In June of 1995, the first product of these efforts MPI 1.1 released [2]. Membership of the MPI Forum has been open to all high performance community members.

Message Passing Interface specification has been used for a wide range of compute systems from general purpose operating systems such as Windows NT and UNIX, to high performance computer systems such as Intel Paragon, IBM SP1.

MPI is not a language, operations of MPI are called as functions, methods according to language features for example C, C++ and Fortran-77 and 95.

MPI Standard is developed for people (application developers that studies on parallel machines) who want to write high level portable programs in C++, C and Fortran. For world wide usage, the standard must present a simple and easy to use interface for advanced machines. 2D finite difference, molecular dynamics and atmosphere-modeling problem are examples of parallel programming model.

Until today, MPI had several extensions. These are provided in remote-memory access operations, process creation, collective operations and parallel I/O.
Also, MPI is not an implementation, it is a specification and there are several MPI implementations. One of these implementations is MPICH, developed by Argonne National Laboratory and Mississippi State University. The MPICH implementation is well-known world-wide used implementation that provides high portability and efficient usage and serves as a basis for several other MPI implementations. After passing several revisions and improvements it became a good message passing implementation. Layered software architecture is the foundation of its portability [3].
There are three layers of MPICH: API, ADI and Device. MPI standard provides threads that user can call API, ADI functions and device threads. The first layer API provides high level message passing logic and helps for implementing abstractions such as topologies and data types. The second layer ADI (Abstract Device Interface), consists of three queues of pointers that handles communication processes.
The third layer of MPICH is the device layer that consists of communication modules and provides three threads: Sender, Receiver and Terminator.
Sender and receiver threads are responsible for making sure that data can be sent or received. Terminator threads are waiting states for the signal to terminate the process [4].

Portability and ease of use are the main advantages of using message passing standard (MPI). This is important for application developers to benefit these specifications in a distributed memory communication environment because of this

architecture includes higher level routines and lower level message passing routines. In addition to this, MPI provides clearly defined routines that can be implemented efficiently. Main goal of MPI is developing a widely used message passing standard for parallel programming. Therefore MPI should be portable, efficient, practical and flexible.

Designing MPI, developers considered some issues:

- For heterogeneous environment, it's allowed for implementations can be used in.
- To provide efficient communication, avoid memory-to-memory copying.
- Fortran-77, Fortran-95, C and C++ are allowed for the developed interface.
- For a reliable communication interface, communication failures are handled by communication subsystem, so the user need not make an effort to solve these problems.
- Without changes in system software and related communication, the interface can be implemented on many vendors' platforms.
- The interface should support thread safety.
- The interface is designed to be language independent.

MPI is designed for parallel machines and workstation clusters to get high performance and portability. It is widely and freely available. Programs which are including high performance message passing operations may run on shared memory architectures, hybrid architectures and particularly on distributed memory architectures.

To provide multiple application level threads, MPI implementation must be thread safe.

In multithreaded programming, thread - safety is a critical issue. If code can be used in multithreaded application, it can be called as thread safe. In order to provide parallelism by multithreading, thread safety allows code to run in multithreaded environments and process synchronization [6].

MPI provides the processes communicate with other processes by calling library routines with send and receive messages. To collaborate and communicate processes with each other, MPI-1 provides library routines and includes two-sided send/receive operation for exchanging data between process pairs. MPI functions are called by threads. Some basic functions that are being used in MPI are defined below:

MPI_Init: A function that helps the system to do need setups for other calls for MPI library. It's not necessary to use this call in the first executable statement but it must be used before any other MPI function.

MPI_Finalize: If the functions are completed, the function MPI_Finalize is called. This free ups the allocated resources for MPI.

MPI_Reduce: This collective communication MPI function is responsible for reduction operations. Reduction operations are finding minimum/maximum, and/or, multiplication and summation [7].

MPI_Barrier: This function is one of the collective communication functions that allows barrier synchronization.

MPI_Bcast: A collective communication function that provides to broadcast one process to other processes in a communicator.

MPI_Gather: A collective communication function that performs gathering operation.

MPI_Scatter: A function that delivers data to different processes.

MPI_Isend: Collective communication operation that performs transmission of the message. Computation thread sends a request using this function for accessing remote memory [5].

MPI_Irecv: The *MPI_Isend* request that is sent by computation thread is received by a communication thread with *MPI_Irecv* in the remote node.

MPI-2 standard is developed to support one sided communication, parallel I/O and dynamic process management. Three one sided communication operations, PUT, GET and ACCUMULATE are supported by MPI-2. These operations are used for writing to remote memory, reading from remote memory and a reduction operation on the same memory across a number of processes. MPI-2 is similar to UPC's global address space programming model because MPI-2 model supports remote process access to data without help from the user. But it's more inhibitive than UPC's global address space model because o cache coherence and synchronization features.

The advantages of MPI model are:

- Process synchronization
- User's complete control of data distribution
- Allowing the optimization of data locality
- Clear (explicit) communication

These characteristics give MPI standard scalability and high performance. Unfortunately these are also made MPI difficult to program and debugging skills.

The disadvantages of MPI model are:

- Not allow incremental development
- Difficult to write program
- Difficult to debug

Developer efforts to reconfigure the existing sequential applications to adapt MPI parallelization.

2.2 UPC Model

There is a growing demand for parallel runtime systems with multi-core processors. Scalable, efficient multi-core systems are increasing their popularity. To meet the growing demand of new programming models that supports these architectures, UPC is shown an alternative to parallel programming models. UPC adds global memory access, parallelism and keeps characteristics of C provides ability to read and write remote memory with simple statements and understanding of what is remote and local for memory access.

High computing vendors and users are interested for the simplicity, usability and performance of UPC and they spend effort to develop and commercialize UPC compilers. Through the efforts of a consortium of industry, government an academia, the first product, UPC specification V1.0 is released in February 2001. In Michigan Technical University and University of California Berkeley, there are available open source implementations such as MuCP and BUPC [8].

UPC (Unified Parallel C) is an extension of C programming language that provides support to application development on distributed, shared and hybrid architectures and simplifies the programmers' problems with logically partitioned address space (PGAS) also known as the distributed shared memory model. This memory model allocates to partitions the memory into available memory domains and also is similar to shared memory model in terms of data locality. Languages which support PGAS, showed that in demonstrations, they provide increased productivity, better performance and high level control over data locality. However, UPC model provides memory coherence, fine -grained, asynchronous communication and dynamic distributed data structures. Every shared data element has affinity (logical association) to a thread by means of UPC. This data locality information is denoted to the user. By this way user can get increased performance.

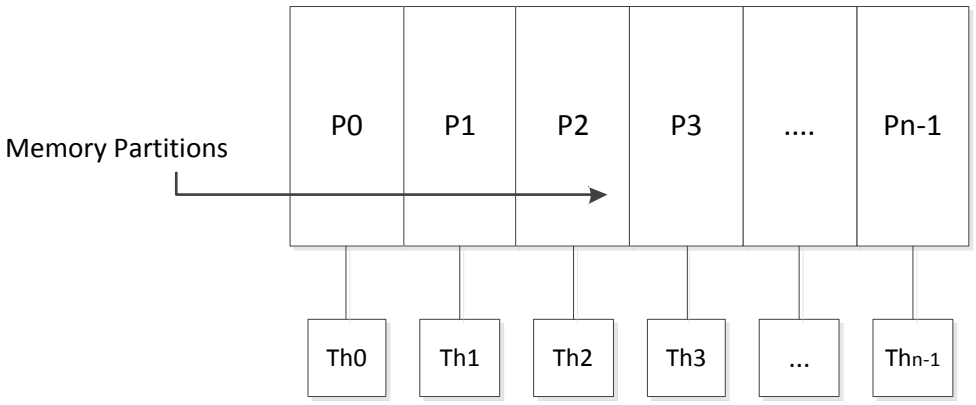

Figure 2.1 Shared Memory Model in UPC

Memory is divided into partitions where each partition P*i* has affinity to thread Th*i*, shown in Figure 2.1. UPC memory is divided into two partitions: Private and Shared. For accessing private part of the memory is similar in C language. To access the shared part, the user should use "shared" qualifier

As we mentioned, UPC has two kinds of data; shared and private. Private Memory space is allocated for an object by UPC when a private object declared. Private objects that are created by a thread is accessible only by this thread. In addition to this, shared memory space is partitioned for shared objects which have affinity with a thread and all data in the global address space can be reached by all operations without help from user.

It's important for data locality to utilize data distribution and work sharing in UPC. This model provides work sharing by *upc_forall* function for distributing tasks to each thread.

Usage of distributed shared memory model simplifies data distribution in UPC. This model provides sharing data by unsophisticated statements. For example to share an

array of size N equally in UPC, user declares the array as “shared” and UPC distributes array in round robin fashion among threads.

UPC has a library that provides parallel programming functions that supports collective (that means data is sent and received from many nodes synchronously) operations such as broadcast, gather, scatter etc. for user to access and manipulate shared and private data by threads in a collective way. This library is also called as UPC Collective Library.

Some basic keywords that are being used in UPC are defined below:

THREADS : THREADS keyword is const int variable that can be signified at compile time or runtime. It defines the number of threads which the current UPC program is used.

MYTHREAD : MYTHREAD keyword is const int variable that used for signify the current thread number that is currently being executed and initialized between 0 and THREADS-1 indexes.

upc_forall : *upc_forall* statement is a collective parallel statement and looks similar to a traditional for loop but adds a fourth parameter, that defines the affinity. This field determines the current iteration of the loop should be run by thread.

upc_barrier : *upc_barrier* is the parallel statement that makes all threads to wait at barrier until all threads has reached it. It provides synchronizing of the threads when data dependency appears between threads.

upc_all_broadcast : This function is used to copy a block of memory which has assigned to a thread, to a block of shared memory on each thread as shown in Figure 2.2.

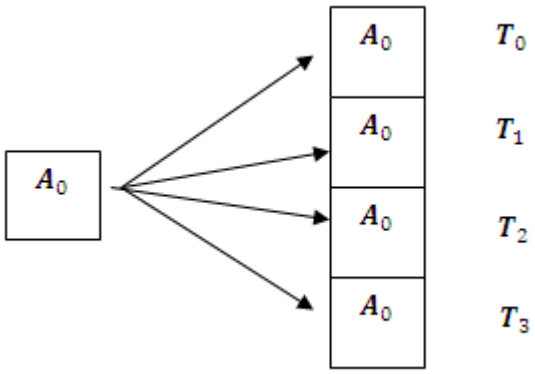

Figure 2.2 upc_all_broadcast

upc_all_scatter : This operation is used for copying an *ith* block of shared memory which has an affinity with a thread, to a block of shared memory which has affinity with *ith* thread as shown in Figure 2.3.

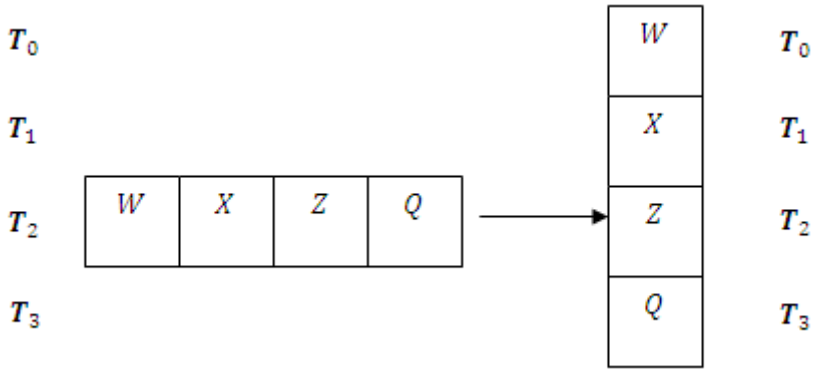

Figure 2.3 upc_all_scatter

upc_all_gather : This operation is used to copy a block of shared memory with affinity to *ith* thread, to *ith* block of shared memory which has affinity to a thread [10] as shown in Figure 2.4.

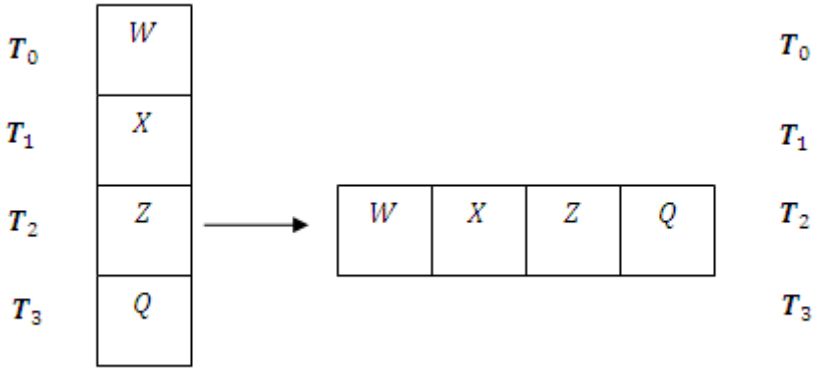

Figure 2.4 upc_all_gather

upc_all_reduce : The *upc_all_reduce* function is a computation operation which is used to execute a user specified operation such as *upc_add* on the threats and return the result to a single thread.

upc_all_sort : The *upc_all_sort* function is a computation operation which is used to perform a sorting operation of a taken shared array in ascending order.

In UPC model requirements of the user to develop application is, analyzing the association of programming model and making effort for data locality and memory consistency.

User don't need to handle complex language structures for high level programming because UPC provides user an easy mapping with low level instructions and also this model presents to user a common syntax for parallel programming in C.

Providing of converting a sequential program to a simple shared- memory implementation easily, performing incremental parallelization of applications, ability and efficient mapping to machine architecture, minimization of thread communication, usage of simple statements are the key features of UPC model. Also pointers and arrays can easily tied to addresses and provides easy usage to

programmer. But like other systems UPC has disadvantages too. UPC does not support thread groups and distributions on arrays do not provide flexibility.

Chapter 3
Hybrid MPI+UPC Model

Hybrid programming model is preferred because of offering reduced communication, improved load balance, memory consumption and also improved level of parallelism.

MPI is an API based library that provides flexible and efficient programming environment the programmer and it can be linked C, C++ or Fortran languages. On the other hand, UPC is an extension of C programming language and supports distributed, shared memory systems for parallel programming. Both MPI and UPC use a Single Program Multiple Data (SPMD) model. SMPD is a high level programming model that provides a single program is executed by all tasks. This model also allows to execute different data on all tasks. Thus, the UPC program calls MPI libraries to form a hybrid program with an SPMD model.

The objective of the hybrid MPI+UPC programming model is to combine the strengths of MPI's locality control and scalability with UPC's fine grain parallelism and ease of programming and UPC's partitioned global address space features. The hybrid model consists of UPC - an extension of C language - , MPI library, so we can say the model is simply a UPC program that calls MPI library and program is compiled with the UPC compiler and linked with MPI libraries [16].

MPI demands large granularity and small messages are expensive because every communication has a fixed startup overhead latency. Thus, the hybrid model will use MPI for the outer parallelism and UPC for inner parallelism.

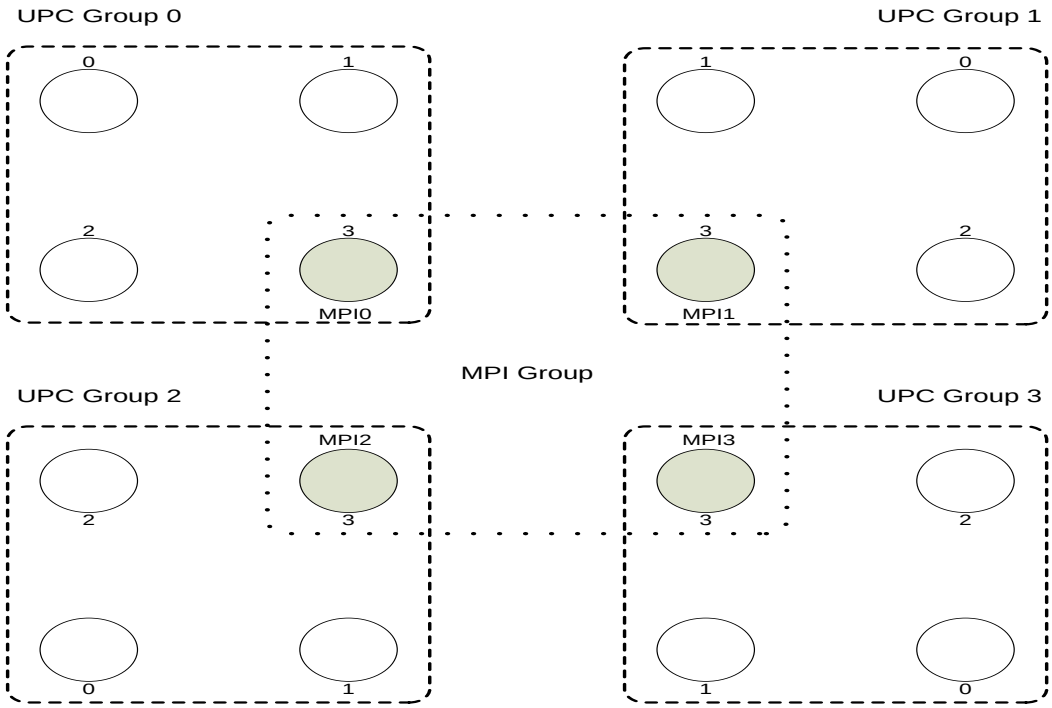

Figure 3.1 The Funneled Hybrid MPI+UPC Model

Figure 3.1 shows a hybrid model in which multiple UPC groups are combined with one outer MPI group. In this figure, gray circles represent the hybrid MPI+UPC master thread and white circles represent UPC threads. UPC threads can communicate with each other within their group while MPI is used for intergroup communication. There is only one master thread in each UPC group, such as the last thread (MYTHREAD == THREADS-1) which can participate in MPI communication; i.e., responsibility of all communication is on the master thread. Our model is very similar to MPI_THREAD_FUNNELED support in which MPI implementation may be multithreaded but only one of those threads (main thread) makes MPI calls [17,18]. We can call this hybrid model as the funneled model.

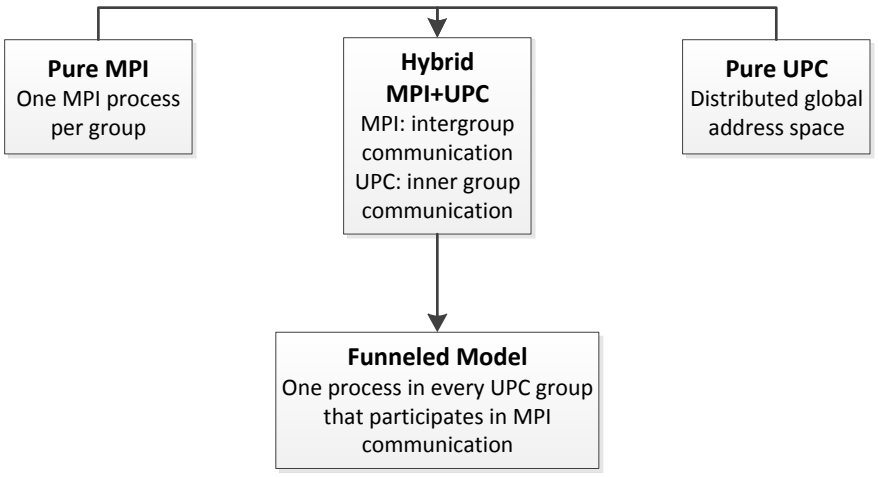

Figure 3.2: Scheme of parallel programming model on hybrid platforms.

As shown in Figure 3.2, hybrid model offers more acceptable model than MPI one sided communication because it's able to utilize UPC's programming features and tunable performance model. Also this model enables the processing of larger problems by using shared global address space for memory constrained MPI codes. For locality constrained UPC codes, this model can provide locality through MPI connections between UPC groups.

Chapter 4
Memory Types

There are three fundamental memory types which multi-processors have been using.

- Shared Memory
- Distributed Memory
- Distributed Shared Memory

We believe it is important to understand these memory types for parallel programming especially, hybrid parallel programming which combines the programming paradigms of different memory types.

4.1 Shared Memory

Shared Memory expression defines a computer architecture that all CPU's can access to a common main memory where the physical memory actually exists [12]. This means addresses of different CPU's are located in same memory location. Also we can call this architecture as Uniform Memory Access (UMA) Multiprocessor or Symmetric Multiprocessor (SMP).

As usual in a uni-processor, CPU is connected to a primary memory and I/O device by a bus. As an extension to this architecture, in shared memory, multiple CPUs are connected to a bus and they share same primary memory. All CPUs have the same access time to memory. A CPU can write a value into the primary memory easily and all other CPUs can read the value. CPU is kept busy by cache memory while data is taken from memory. The architecture is showed below in Figure 4.1 :

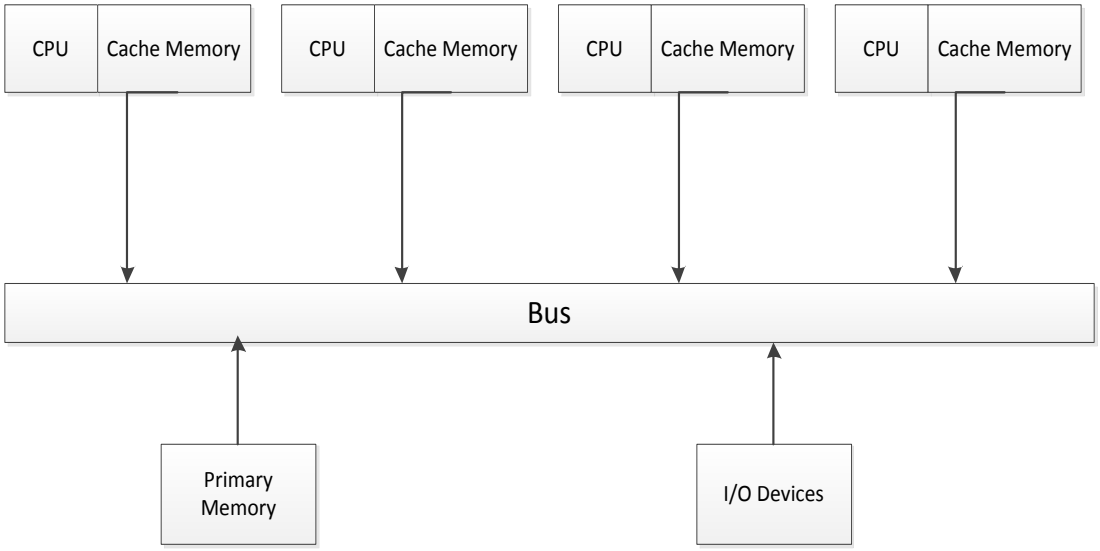

Figure 4.1 Structure of Shared Memory

There are some advantages and disadvantages of this architecture. Having a single primary memory provides a user friendly programming environment and data sharing between processors are fast and uniform. On the other hand, when user wants to add more CPUs, the traffic on the shared memory - CPU bus increases and becomes bottleneck so, to design and produce machines that have increasing number of CPUs is expensive and difficult. Most of shared memory machines have ten or fewer processors because of scalability problem [13].

4.2 Distributed Memory

Shared memory has limits for existence of processors in the architecture. We mentioned that if the user adds more processors to the existing system, there can occur bottleneck problem. To overcome this problem, distributing primary memory builds an alternative architecture called as distributed memory and also called nonuniform memory access (NUMA) micro-processor. In distributed memory, each CPU has its own local memory and I/O. Distributed memories work independently.

Changes in the local memory don't effect other processor's memories. Memory access time changes from process to process whether address is located in that processors local memory or remote processors local memory. If a processor is needed to accessed by an another processor for getting data from it, user must define the communication. In addition to this, synchronization of processes is one of the users responsibilities.

Distributed memory architecture is scalable. User can increase the number of CPUs and memories. Each processor can easily compute with its own memory. However user is responsible for many skills about data communication when programming and converting an existing structure that is mapped with shared memory architecture can be difficult. Distributing data over memories is an important point in programming. Distributed Memory architecture is showed in Figure 4.2 :

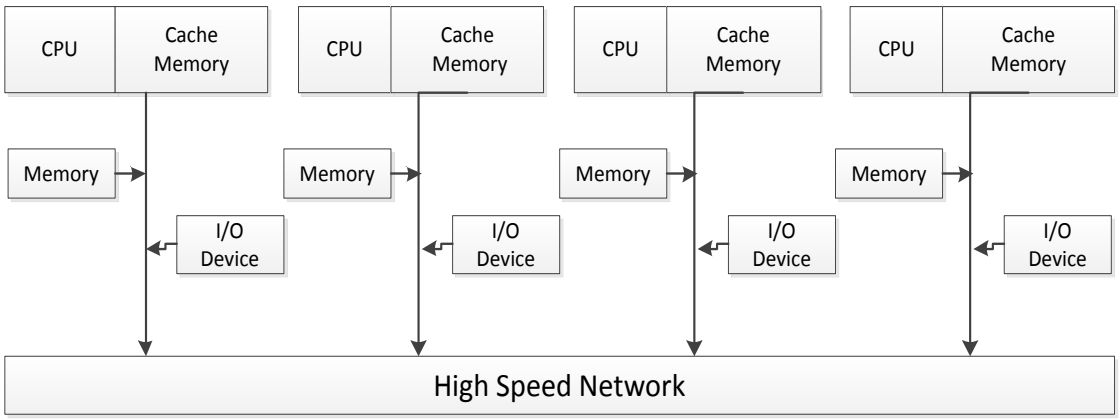

Figure 4.2 Structure of Distributed Memory

4.3 Distributed Shared Memory

Distributed shared memory (DSM) also known as distributed global address space (DGAS) or partitioned global address space (PGAS). This hybrid architecture combines shared memory architecture and distributed memory architecture. This system utilizes both distributed memory and shared memory architectures and improves flexibility and performance.

The architecture is designed by dividing memory into shared parts among nodes as shown in Figure 4.3.

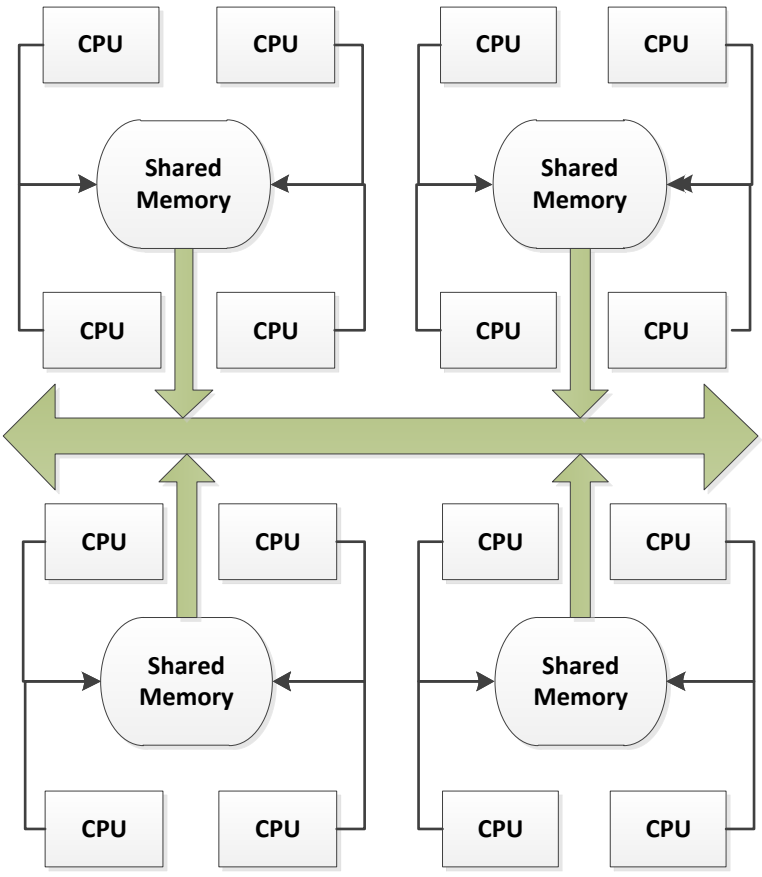

Figure 4.3 Structure of Distributed Shared Memory

Distributed shared memory reduces cost and complexity of developing a program and it becomes easier to integrating different architectures to this architecture [14].

MPI and UPC allow programming on these memory types. However, we targeted our hybrid programming model runs on the distributed shared model. The high performance computer at Kadir Has University has the distributed shared memory model.

Chapter 5
Cannon's Algorithm

Many scientific and engineering problems such as in signal processing and chemistry need to use large matrices and matrix multiplication can be required for solution. There are many matrix multiplication algorithms. One of them is Cannon's algorithm also it can be called Fox's algorithm [11].

Cannon algorithm is first described in 1969 by Lynn Elliot Cannon and this algorithm provides memory efficiency in parallel programming and less communication between processes.

We are interested in performing **C = A x B** multiplication where **C**, **A**, **B** are n x n square matrices. We assume that these three matrices are decomposed into 2 dimensional square sub blocks. Let P is a square number and n will be the multiple of $\sqrt{P}$.

In Cannon algorithm, a processor is responsible for (n / $\sqrt{P}$) x (n / $\sqrt{P}$) block of C. The steps of matrix multiplication are explained for the case of 4x4 matrices as an example below in Figure 5.1 .

$C_{0,0}$	$C_{0,1}$	$C_{0,2}$	$C_{0,3}$
$C_{1,0}$	$C_{1,1}$	$C_{1,2}$	$C_{1,3}$
$C_{2,0}$	$C_{2,1}$	$C_{2,2}$	$C_{2,3}$
$C_{3,0}$	$C_{3,1}$	$C_{3,2}$	$C_{3,3}$

=

$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$A_{0,3}$
$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	$A_{1,3}$
$A_{2,0}$	$A_{2,1}$	$A_{2,2}$	$A_{2,3}$
$A_{3,0}$	$A_{3,1}$	$A_{3,2}$	$A_{3,3}$

x

$B_{0,0}$	$B_{0,1}$	$B_{0,2}$	$B_{0,3}$
$B_{1,0}$	$B_{1,1}$	$B_{1,2}$	$B_{1,3}$
$B_{2,0}$	$B_{2,1}$	$B_{2,2}$	$B_{2,3}$
$B_{3,0}$	$B_{3,1}$	$B_{3,2}$	$B_{3,3}$

C = A x B

Figure 5.1 Matrix Multiplication

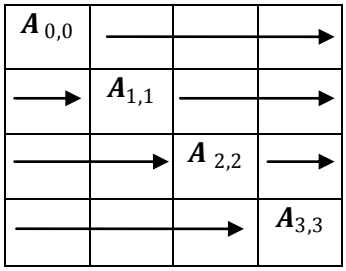

Figure 5.2 (a)

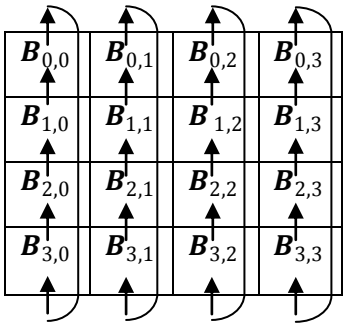

Figure 5.2 (b)

Figure 5.2 Broadcast and shift operations

First, starting from the diagonal sub blocks (Figure 5.2 (a)) of matrix A, $\boldsymbol{A_{0,0}}$ is broadcasted to all processes in the first row. In the second row $\boldsymbol{A_{1,1}}$ is broadcasted to all processors. For the other rows this process occurs similarly.

Then broadcasted **A** sub blocks are multiplied with sub blocks of **B** matrix in each processor and stored in the sub blocks of **C** matrix.

After each multiplication sub blocks of B matrix are shifted to the upper sub block and replaced with the lower sub block (Figure 5.2 (b)). This replacement is continuously done and become a circular movement. And the results of the product of sub blocks are added to the partial results of **C** sub blocks. Matrix multiplication operation continues until sub blocks of **B** matrix are retuned to their original places. Matrix multiplication steps are showed below in Figure 5.3 :

$A_{0,0}$ $B_{0,0}$	$A_{0,0}$ $B_{0,1}$	$A_{0,0}$ $B_{0,2}$	$A_{0,0}$ $B_{0,3}$
$A_{1,1}$ $B_{1,0}$	$A_{1,1}$ $B_{1,1}$	$A_{1,1}$ $B_{1,2}$	$A_{1,1}$ $B_{1,3}$
$A_{2,2}$ $B_{2,0}$	$A_{2,2}$ $B_{2,1}$	$A_{2,2}$ $B_{2,2}$	$A_{2,2}$ $B_{2,3}$
$A_{3,3}$ $B_{3,0}$	$A_{3,3}$ $B_{3,1}$	$A_{3,3}$ $B_{3,2}$	$A_{3,3}$ $B_{3,3}$

C

=

$A_{0,0}$	$A_{0,0}$	$A_{0,0}$	$A_{0,0}$
$A_{1,1}$	$A_{1,1}$	$A_{1,1}$	$A_{1,1}$
$A_{2,2}$	$A_{2,2}$	$A_{2,2}$	$A_{2,2}$
$A_{3,3}$	$A_{3,3}$	$A_{3,3}$	$A_{3,3}$

T

x

$B_{0,0}$	$B_{0,1}$	$B_{0,2}$	$B_{0,3}$
$B_{1,0}$	$B_{1,1}$	$B_{1,2}$	$B_{1,3}$
$B_{2,0}$	$B_{2,1}$	$B_{2,2}$	$B_{2,3}$
$B_{3,0}$	$B_{3,1}$	$B_{3,2}$	$B_{3,3}$

B

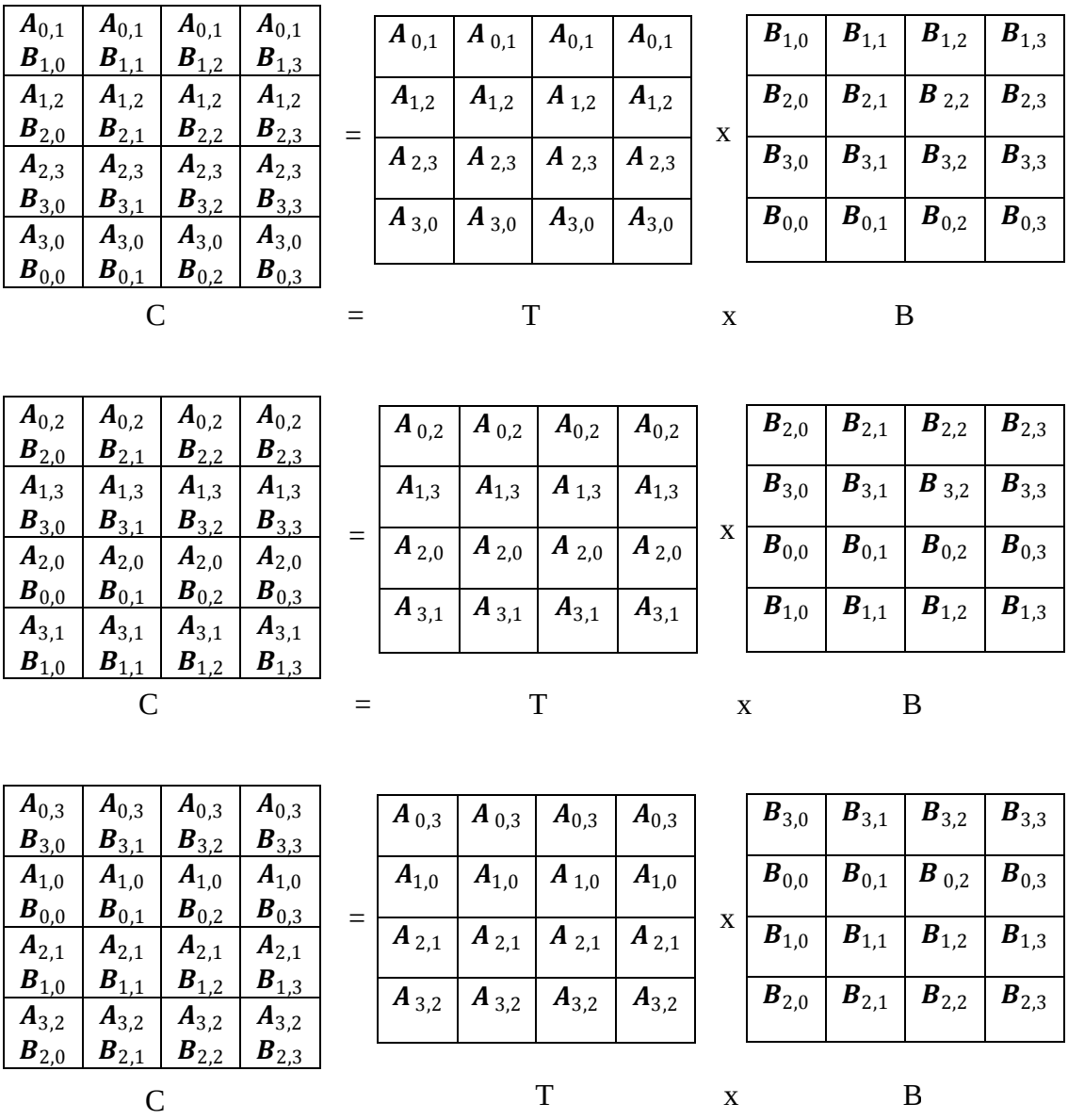

Figure 5.3 Steps of Cannon's Matrix Multiplication Algorithm

T matrix is the A matrix that is utilized at the current operation.

For the element of $C_{1,2}$ is equal to $A_{1,1} * B_{1,2} + A_{1,2} * B_{2,2} + A_{1,3} * B_{3,2} + A_{1,0} * B_{0,2}$.

The Cannon's algorithm is showed below:

```
The Cannon matrix multiplication algorithm.

for i = 0 to (√P − 1)  do  // P is the total number of processors

        broadcast_A( T ← appropriate A_sub along rows )

        C_sub ← T x B_sub

        cshift_B ( B_sub upward along colums )

end for
```

Figure 5.4 Cannon's Algorithm

As shown in Figure 5.4, algorithm has three fundamental steps. The first step broadcasts the diagonal $\boldsymbol{A_{sub}}$ sub-blocks along each row of tiles. The broadcast source will be shifted to the right of the rows for the next iteration. The second step performs sub matrix multiplication. The final step performs an upward circular shift along to each column of B matrix.

Chapter 6
Codes Overview

In this chapter we developed four different implementations of the same parallel program. This parallel program is matrix multiplication with Cannon's algorithm and MPI, UPC, hybrid MPI+UPC and optimized hybrid MPI+UPC version respectively.

6.1 MPI Implementation

Our MPI implementation of Cannon's algorithm is based upon a two-dimensional block decomposition, in which there are two collective communication operations involving a subset of the processes such as rows of processes and columns of processes. In order to involve only a subset of the original process group in a collective operation, we need to create a Cartesian topology, a two-dimensional virtual grid of processes as shown in Figure 6.1

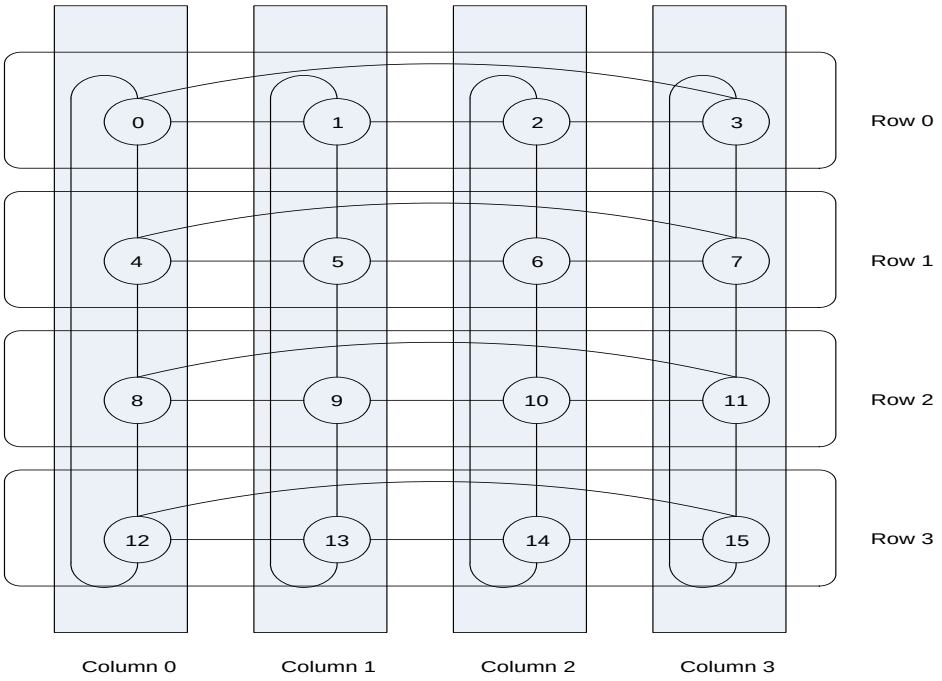

Figure 6.1: An MPI Cartesian Topology: a two-dimensional virtual grid of MPI processes with wraparound connections.

In order to involve only a subset of the original process group in a collective operation we need to utilize Cartesian virtual topology. Virtual topologies in MPI, define a mapping of processes into a geometric shape [15]. This is not a physical implementation, a logical implementation that should be programmed by software developer. Generally virtual topologies are used for simplifying writing code and optimizing communications.

Each process is connected to its neighbors virtually and shifting operation can be cyclic [19]. As we used in our MPI implementation, this topology might used for decomposing a matrix into two dimensional logical grid of processes and providing communication that user defined row wise or column wise communication as shown in Figure 6.1 above.

MPI supports and provides Cartesian topology and has Cartesian topology functions. Basics are:

- MPI_Cart_Coords
- MPI_Cart_Create
- MPI_Cart_Rank

MPI_Cart_Coords function is used to handle and process a Cartesian communicator. This function returns the coordinates of the process in the Cartesian topology as a result.

MPI_Cart_Create function is used to create a new communicator which includes topology information.

MPI_Cart_Rank function is used to return process rank in given Cartesian communicator.

Figure 6.2 shows the Cannon's algorithm implementation written for the MPI model. The code has two major parts, the first of which is to construct the Cartesian topology. The second part is to implement the algorithm in three steps. In the first step, the processes in the rows of the virtual process grid participate in the broadcast communication. The second step performs the sub matrix multiplication. In the last step, each column of processes in the grid performs the send/receive operations for executing a circular shift operation across a chain of processes in the grid column. The MPI program has explicit control of data locality. Regional data locality is provided among rows and columns by using the advance feature of MPI's Cartesian virtual topology. MPI can enhance UPC by providing explicit control over data locality in the hybrid programming model, and Cannon's algorithm is an ideal selection to demonstrate the importance of regional data locality.

```
// PART 1: Construct a Cartesian topology
MPI_Init (&argc, &argv);
MPI_Comm_rank (MPI_COMM_WORLD, &id);
MPI_Comm_size (MPI_COMM_WORLD, &p);
MPI_Dims_create (p, 2, grid_size);
MPI_Cart_create (MPI_COMM_WORLD, 2, grid_size, periodic, 1, &grid_comm);
MPI_Comm_rank (grid_comm, &grid_id);
MPI_Cart_coords (grid_comm, grid_id, 2, grid_coords);
MPI_Comm_split (grid_comm, grid_coords[0], grid_coords[1], &row_comm);
MPI_Comm_split (grid_comm, grid_coords[1], grid_coords[0], &col_comm);
// PART 2: Cannon Algorithm
int S = (int) sqrt( p );
for( k = 0; k < S; k++) {
  MPI_Bcast(Atmp, N*N, mpitype, src, row_comm); // STEP 1: Broadcast A
  matmul(Atmp,B,C);                             // STEP 2: C = A x B
  MPI_Sendrecv(B, N*N, mpitype, left, tag,      // STEP 3: CSHIFT B
               btmp, N*N, mpitype, right, tag, col_comm, &status);
}
```

Figure 6.2: MPI implementation of Cannon's Algorithm.

6.2 UPC Implementation

Figure 6.3 shows the UPC implementation of matrix multiplication with a block distribution. The UPC code for the matrix multiplication is almost the same size as the sequential code. This makes UPC easy to program and it allows for incremental parallelization of sequential codes. The global (shared) array declaration has the keyword *shared [block-size]* to distribute shared arrays in a block per thread in a round-robin fashion.

```
shared [N] double Aupc[N][N], Bupc[N][N], Cupc[N][N];

void matmul_upc( ) {

  int i,j,k;

  double sum;

  upc_forall(i=0; i<N; i++; &Aupc[i][0])

    for (j=0; j<N; j++) {

      sum = 0;

      for(k=0; k< N; k++)

        sum +=Aupc[i][k]*Bupc[k][j];

      Cupc[i][j] = sum;

    }

}
```

Figure 6.3: UPC matrix multiplication with block distribution.

The UPC does not provide a two-dimensional virtual topology to make a group of threads for regional data locality such as the row-wise or column-wise grouping as presented in Section 6.1. The UPC only differentiates between two different kinds of data, shared (global) and private (local) for threads. UPC partitions parallel works by using the *upc_forall* construct, which distributes iterations of the loop according to the affinity expression, &Aupc[i][0]. The UPC will assign each iteration to the thread that has affinity to the corresponding element of Aupc. Berkeley UPC distribution version 2.10.2 provides a matrix multiplication for the Cannon algorithm and that is the code we utilized for UPC matrix multiplication benchmarking. However, our MPI+UPC hybrid uses the block matrix multiplication (Figure 6.3) for the sub matrix multiplication.

6.3. Hybrid MPI+UPC Implementation

In Figure 6.4, the funneled hybrid MPI+UPC is formed with the careful combination of the MPI program of Figure 6.2 and the UPC program of Figure 6.3. Here again, we presented the simplified hybrid code however, the main algorithm of the code should be clear. The hybrid program consists of one MPI group and each MPI process has one UPC group as shown in Figure 3.1. There is only one master thread in each UPC group such as the last thread (MYTHREAD == THREADS-1) which can participate in MPI operations as an MPI process. The master thread initializes MPI and is able to construct an MPI Cartesian topology at Part 1 of the code. Part 2 performs Cannon's algorithm with two explicit levels of parallelism. MPI manages the outer parallelism by bringing the appropriate sub-blocks to the master threads of each UPC group. Master threads copy their private sub-blocks to the shared sub-blocks by a *copy_from_master_to_upc()* routine which performs simple copy operations and synchronizes each UPC thread group with *upc_barrier()*. The shared sub-blocks are in the global address space of each UPC group. Each UPC participates in the inner level parallelization of sub-block matrix multiplication with a block distribution which is given by *matmul_upc()* in Figure 6.3.

```
shared [N] double Aupc[N][N], Bupc[N][N], Cupc[N][N];

Boolean MASTER = (MYTHREAD == THREADS - 1);

int main (int argc, char *argv[]) {

  if(MASTER) { // PART 1: Construct a Cartesian topology

    MPI_Init (&argc, &argv);

    MPI_Comm_rank (MPI_COMM_WORLD, &id);

    MPI_Comm_size (MPI_COMM_WORLD, &p);

    . . .

    MPI_Comm_split (grid_comm, grid_coords[0], grid_coords[1], &row_comm);

    MPI_Comm_split (grid_comm, grid_coords[1], grid_coords[0], &col_comm);

  }

  for( k = 0; k < S; k++) { // PART 2: Cannon Algorithm

    if(MASTER)MPI_Bcast(Atmp,N*N,type,src,row_comm); // STEP 1:Broadcast A

    copy_from_master_to_upc(Atmp,B);                 // Copy to shared

    matmul_upc();                                    // STEP 2: C = A x B

    if(MASTER) MPI_Sendrecv(B,N*N,type,left,tag,     // STEP 3: CSHIFT B

                   btmp, N*N, mpitype, right, tag, col_comm, &status);

  }

}
```

Figure 6.4: Hybrid MPI+UPC Cannon's algorithm. The sub-block multiplication is with UPC block distribution.

The hybrid program is compiled with the UPC compiler and linked with the MPI libraries. The *-fupc-threads-NUM* option generates code for a fixed number NUM of UPC threads. The MPI launcher is used to start the hybrid program.

Below is an example UPC compilation and an example MPI launcher in which 4 MPI processes are created and each process creates a UPC group with 4 threads. The total parallel thread number is 4MPI x 4UPC = 16 threads, as shown in Figure 3.1.

```
$ upcc –o matmul_hybrid matmul_hybrid.upc -fupc-threads-4 –O

$ mpiexec --mca btl self,openib –np 4 –hostfile hosts
matmul_hybrid
```

6.4. Optimized Hybrid MPI+UPC Implementation

Although overlapping communication with computation provides the opportunity to improve the execution time of a parallel program, this parallel programming style is not widely used due to its complexity. However, the hybrid Cannon's algorithm presents a good opportunity for overlapping communications with computation. In this algorithm, we only need the full synchronization of each UPC group before performing the sub-block matrix multiplication. The computation of sub-block UPC multiplication can be overlapped with the MPI's communication. However, the *upc_forall* construct distributes iterations of the loop according to the affinity expression at the fourth parameter of the construct. The UPC will assign each iteration to the thread that has affinity to the corresponding element of the shared array. This implies that assignment statements of *upc_forall* involving shared arrays are executed exclusively by those threads which are owners of the shared array elements. We structured our hybrid code such that the master thread will not spend excessive much time on *upc_forall* computations, but rather dedicate time to MPI communication, since the master thread is both the UPC thread and the MPI process.

If the hybrid code partitions the shared arrays such that the master thread does not have shared array elements or has much less than the other threads, the master will finish *upc_forall* earlier than the others and reach the MPI communication operations while the other threads are still executing their portion of *upc_forall* iterations. Figure 6.4 shows a distribution scheme in which fewer shared array elements reside on the master thread's memory space. The shared array declaration has the keyword *shared [block-size]* to distribute shared arrays in a block per thread in a round-robin fashion. The shared array size is NxN in the sub-block of Cannon's algorithm, and in this way the master thread should have no data. The block size must be (N*N)/(THREADS-1) so that the last thread will have no data in a round-robin fashion. However, the Berkeley UPC implementation has a limit for the block size of 64k. Even if the problem size reaches the limit, the solution in Figure 6.4 still provides less shared array elements to the last thread (master thread) because of the round-robin fashion distribution. The addition of the code in Figure 6.5 optimizes the hybrid MPI+UPC implementation of Figure 6.4 to overlap MPI communications with UPC computations on hybrid MPI+UPC implementations.

```
#define BLOCK (N*N)/(THREADS-1)

#if BLOCK < 65536

shared [BLOCK] double Aupc[N][N];

shared [BLOCK] double Bupc[N][N];

shared [BLOCK] double Cupc[N][N];

#else if

shared [ 65536 ] double Aupc[N][N];

shared [ 65536 ] double Bupc[N][N];

shared [ 65536 ] double Cupc[N][N];

#endif
```

Figure 6.5: Distribution scheme to optimize the hybrid MPI+UPC Cannon's algorithm.

Chapter 7
Performance Evaluation

This section will illustrate the impact of our proposed hybrid MPI+UPC approach through several experiments by running four different implementations of the Cannon algorithm on a 13-node HP BL460c cluster located at Kadir Has University. This SMP cluster consists of 2x2.66 GHz Intel Xeon Quad Core CPUs and 24 GB RAM with a total of 8 processing cores per node running Linux 2.6.18 connected with 20 GBps Infiniband.

The first experiment objective is to find an optimum UPC group for the hybrid MPI+UPC model. The results in Figure 7.1 show the time required to perform 5000^2 matrix multiplications with groups of 1, 4, 8, 16 and 64 UPC THREADS and MPI processes. The ideal performance is in the group of 8 UPC threads. Each node in our system holds 8 cores per slot, indicating that each UPC thread goes to different cores in the nodes. In fact, we configured UPC's GASNET_NODEFILE such that each consecutive thread goes to consecutive cores in the node for the hybrid runs. Similarly, we ensured that each MPI process goes to different nodes by putting slots=1 option in the MPI's hostfile for the best performance of the hybrid model with a round-robin fashion. However, during plain MPI runs, the hostfile is configured with slots=8 to advise consecutive processes to be in consecutive cores for the process affinity.

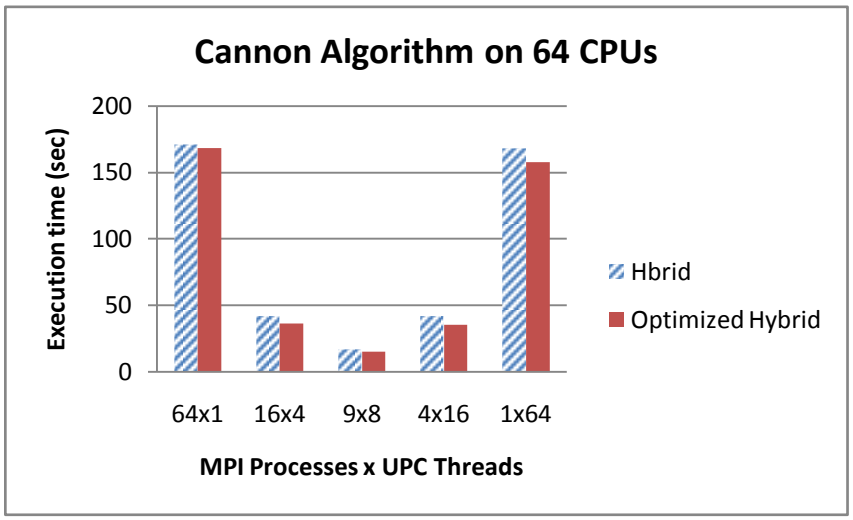

Figure 7.1: The effect of varying numbers of MPI processes and UPC threads on execution time for 5000^2.

The second experiment was designed to reveal how well the proposed hybrids perform compared to plain MPI and plain UPC versions. Figure 7.2 shows the total execution time on the vertical axis, and the horizontal axis denotes the problem size. The CPU times of hybrids and others for the small problem size of 2000^2 and 3000^2 are almost the same. For the other problem sizes, hybrids consistently obtain better performance than the UPC. However, the MPI outperforms hybrids for problem sizes of 7000^2 and 10000^2.

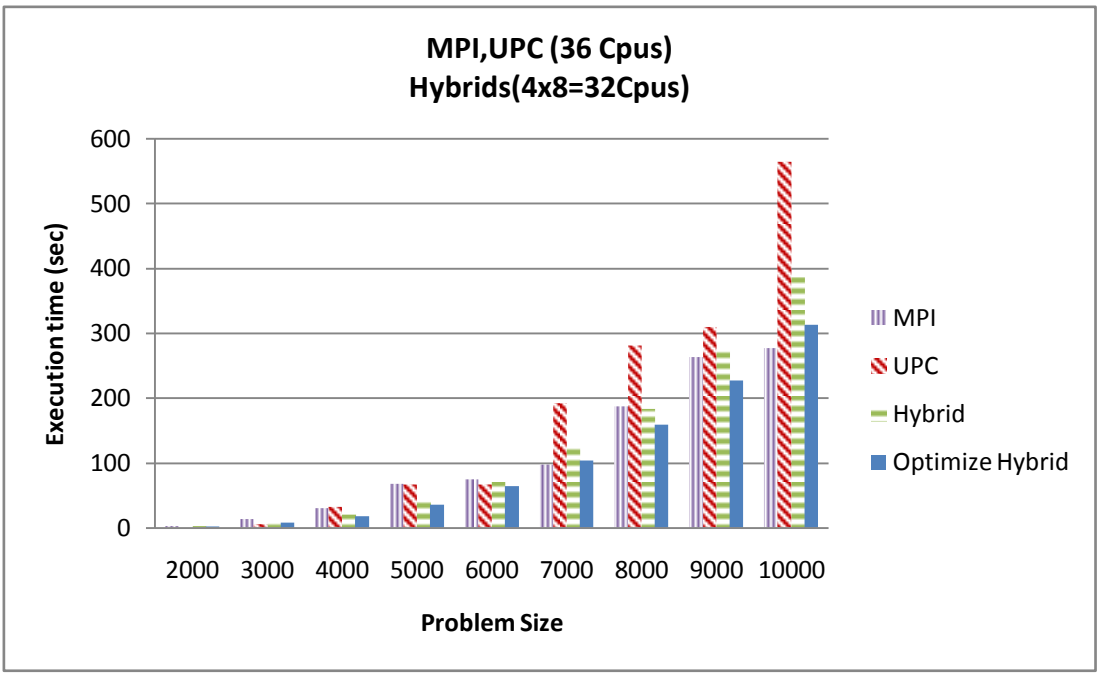

Figure 7.2: Execution time of Cannon's Algorithm on 36 nodes with varying problem sizes, Hybrids run 4 MPI x 8 UPC configurations.

Table 7.1 shows the percentage gains of the optimized hybrid compared to MPI, UPC and plain hybrid. For the 5000^2 problem size, the optimized hybrid shows an efficency of 47.72% over MPI and 46.83% over UPC. As for the plain hybrid, the optimized hybrid achieved approximately 15% efficency.

Table 7.1: Comparing percentage gains of the optimized hybrid MPI+UPC version to MPI, UPC and the plain hybrid MPI+UPC of Figure 6.4.

Data Size	**4000^2**	**5000^2**	**6000^2**	**7000^2**	**8000^2**	**9000^2**	**10000^2**
Gain Over MPI	40.95	47.72	13.4	-7.19	14.72	13.49	-12.92
Gain over UPC	44.76	46.83	3.13	45.72	43.06	26.49	44.56
Gain over Hybrid	10.36	13.11	13.10	15.35	12.94	17.45	19.04

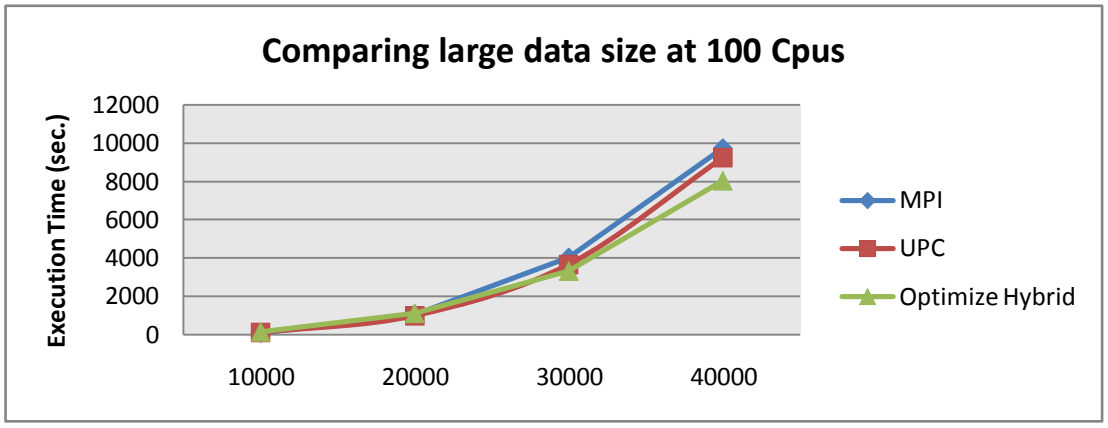

Figure 7.3: Execution time of Cannon Algorithm on 100 CPUs with large problem sizes.

Figure 7.3 shows the behavior of MPI, UPC and optimized hybrid MPI+UPC codes on the large data size with a fixed number of CPUs. The optimized hybrid achieved around 20% and 15% efficiency compared to MPI and UPC, respectively. However, UPC code outperforms MPI in this scenario. As MPI requires copying data on shared memory, when the data size increases and CPU size stays the same this increases the number of the direct access of shared memory of UPC, and therefore MPI could be considered less efficient unless the number of CPUs increases along with data sizes.

Chapter 8
Related Works

Hybrid parallel programming is a model of combining multiple models within a single application. The general objective of these studies is to merge strengths of different kinds of models for increasing performance, scalability and also cost. Hybrid programming with MPI and OpenMP studies are very popular. In these studies MPI takes the responsibility of outer level parallelism and OpenMP makes inner communication.

Hybrid programming is shown as a model that is utilized to provide more efficient use of memory, improving load balance and performance. In the literature, there are lots of papers about hybrid programming that shows hybrid model and we mentioned some of them below.

One of the studies on SMP nodes is Rabenseifner et al. [18] discussed benefits of MPI+ OpenMP hybrid programming model, comparing with the main strengths and weaknesses of pure MPI model and pure OpenMP model. This study also describes solutions of possible problems that can be occur in hybrid programming model.

The hybrid model which is discussed in this study is a masteronly model and combines distributed memory parallelization on the node interconnect with shared memory parallelization inside each node. This model uses one MPI process (main thread) per node with no MPI calls inside the node. OpenMP makes calls inside the node. MPI provides this model with MPI_THREAD_FUNNELED. And in addition to this MPI supports MPI_THREAD_SINGLE which has only one thread to execute, MPI_THREAD_SERIALIZED which provides multiple threads to make MPI calls,

only one at a time and MPI_THREAD_MULTIPLE that provides multiple threads can make MPI calls with no restrictions.

We mentioned that MPICH is the most known MPI implementation used worldwide. Prototov and Skjellum [3] explain layers, capabilities and deficiencies of MPICH implementation in the study. One of the main deficiencies is known that nonthread safety of this implementation this means all communication and computation activities run in a single thread. They generate a portable, thread safe and multithreaded MPICH design that allows multithreaded applications to take advantage of MPI.

Skjellum et al. [4] describe the requirements for thread safe MPI implementation. The study mentions that it should use thread safe run time libraries and the design of the model should call multiple threads and services of them for execution. The paper also defines features of non-thread-safe MPICH design and improved model of thread safe so multithreaded MPICH design, named MPICH_MT. (Polling, pushing and busy waiting operations are eliminated for gaining performance.) Designing new model, portability and efficiency remained, performance added to the current design. This new model is developed for the Windows NT operating system using Win32 API.

Ojima et al. designed and implemented a software distributed shared memory system named SCASH-MPI [20] by using MPI as the communication layer by utilizing features of OpenMP such as incremental parallelization and shared memory feature.

Jost et al. [21] compare different parallelization approaches on clusters of SMP (symmetric multi-processors) nodes for the selected benchmark application. These approaches are; pure MPI, pure OpenMP and two hybrid (MPI+OpenMP) strategies and selected benchmark Block Tridiagonal (BT) solution of the CFD (computational

fluid dynamics) benchmark. BT benchmark solves system in three dimensional equations.

For this comparative study three sun fire cluster environments running on Solaris are used. Four Sun Fire 6800 nodes are connected by a Sun Fire Link (SFL), Four Sun Fire 6800 nodes are connected by a Gigabit Ethernet (GE) and one Sun Fire 15K node are used for implementations. Fortran 95 compiler was used. They report the best timings for the hybrid implementations. The pure MPI model shows the best scalability. In spite of the pure OpenMP shows good scalability, it has disadvantages when compared to MPI implementation. These disadvantages are reported that; OpenMP requires shared address space and this limits scalability of the number of CPUs in the node, because of the usage of OpenMP directives on the outer loops, it restricts the scalability the number of inner grids in one dimension. As conclusion, this study shows hybrid parallelization is suitable for large applications.

Parks [22] studied on comparison of process networks and MPI. The process network model is a communication model that is used applications of embedded signal processing, image processing for geographical information systems and sonar beam forming. In this paper it's shown that, send/receive, broadcast and gather/scatter primitives of MPI can be emulated in process networks. Also benchmark performances are made using java library for MPI and process network model.

He et al. define the MPI+OpenMP model and implements the row-wise and column-wise block-striped decomposition based matrix multiplication with hybrid (MPI + OpenMP) programming model in the multi-core cluster system [23]. To compare the performance of hybrid model also pure MPI implementation is generated too. Programs are executed with the 1400x1400 and 2100x2100 size of matrixes and compilers are Windows XP Visual Studio 2005 and GCC (compiler system of Unix Compatible Operating System-GNU-),and the compiler on Linux is GCC.

Performance tests are made using visual studio compiler on windows, GCC compiler on Linux and GCC compiler on Windows.

Experimental results are shown in the paper that running time of hybrid implementation is shorter than pure MPI implementation and Windows is better than Linux for parallel environment. The study is resulted that MPI + OpenMP makes good use of CPU computing resources and optimizes the parallel algorithm which is based on pure MPI implementation.

Another study that evaluates MPI performance against UPC and OpenMP written by Mallon et al.[24]. UPC is defined as a PGAS language that allows shared memory like programming on distributed memory systems with utilizing the advantage of MPI's data locality. Additionally features of MPI, OpenMP and hybrid model are shown in the paper.

Smith and Bull [25], discuss implementation, development, performance and benefits of MPI/ OpenMP applications on single and clustered SMPs in their study. Before the case studies benefits and deficiencies of OpenMP, MPI and hybrid model are reviewed. For the case study, Game of Life code parallelized and MPI, OpenMP and MPI+OpenMP versions are developed and performance results are shown. Game of Life consists a collection of 2D cells that have only two states alive or dead, based on some mathematical rules. Result of this implementation is reported as; OpenMP code offers potential for MPI/ OpenMP codes to give a better performance than pure MPI implementations. Another case is a real application code named Quantum Monte Carlo code (QMC). This consists of large class of computer algorithms to simulate quantum systems.

In this study pure MPI version and OpenMP version of mixed mode QMC code are implemented and results of the implementations are shown. As conclusion it's

reported that, a hybrid model of programming may provide more effective performance for an SMP cluster because of hybrid model merges different characteristics of both paradigms to give the best performance. However, it doesn't mean that it will always be the most effective and ideal model for all clusters. It must be considered for the nature of the code.

Blagojevic et al.[1] designed and implemented Process Shared Memory (PSHM), an implementation of Berkeley UPC runtime, to provide shared memory among processes and allows hybrid execution models. UPC tasks are mapped to a mix of processes and pthreads (POSIX-thread programming library Portable Operating System Interface- shared memory segments). This is the first PGAS implementation that provides hybrid mapping of threads to both processes and pthreads using shared memory. Fine grained benchmarks, GUPS, MCOP and Sobel are discussed. GUPS is a benchmark which operates read/write and modifies accesses to random locations in large arrays. MCOP benchmark performs solution of matrix chain problem by distributing columns across UPC threads and communication occurs between these threads. Sobel performs edge detection by partitioning source image across threads. this study combines PHSM with pthreads and fine grained benchmarks, more than %60 increment of performance is reported.

A mixed mode MPI+OpenMP version of the CGWAVE [26] code have developed by The DoD High Performance Computing Modernization Program (HPCMP) Waterways Experiment Station (WES). The CGWAVE code is used for forecasting and analysis of harbor conditions. To distribute the wave components MPI is used and for parallelization of the code OpenMP is used. The mixed code won the "Most Effective Engineering Methodology Award" at SC98.

A new hybrid model that combines MPI and UPC was explored first by [16] Dinan et al. and this model is improved to allow incremental access to greater amount of

memory and utilizes MPI's data locality and UPC's partitioned global address space features. Hybrid MPI+ UPC programming model is defined in terms of sub models that vary the level of nesting and the number of instances of the model. They classify the hybrid model into three categories then compare and explain these models with calculating dot product of two vectors using hybrid model. The flat model permits all processes to participate in both UPC and MPI communication. In the nested-funneled model only one member of each UPC group can participate in MPI communication. Another hybrid model is the nested-multiple model, the most powerful, allowing MPI to span to all UPC processes in all groups. But, this added flexibility causes greater complexity. Each process participates in communication so all processes in each group initializes MPI. Authors have demonstrated its effectiveness and performance gains with Barnes-Hut n-body simulation which simulates gravitational interactions and motion of astronomical bodies over time. Our funneled model is similar to the nested-funneled model in that only one member of each UPC group is able to make MPI calls. However, in our funneled model, every UPC group has its own shared variables which are only distributed at that UPC group; they are not distributed across all groups to enlarge the MPI memory. This study calculates lower and upper bounds of *upc_forall* by using MPI id and UPC group id. In contrast, our approach simply uses the shared array index *upc_forall* for lower and upper bounds in each UPC group. In addition, we implemented the Cannon algorithm with advanced MPI features to show the importance of regional locality in comparison with UPC implementation of the same algorithm. We also exploited some of the advanced features of MPI and UPC programming to overlap MPI communications with UPC computations on the hybrid MPI+UPC implementations.

Chapter 9
Conclusion

In this thesis, we have researched advantages, disadvantages, strengths, weaknesses and various implementations with giving examples in real life of MPI model, UPC model. And hybrid MPI+UPC model is discussed to increase performance and scalability with combining two different models features. Also memory types that are being used in parallel programming are introduced with figures and a matrix multiplication algorithm called Cannon's algorithm is discussed and used with MPI virtual topology in our implementations.

We have developed a hybrid parallel programming model that is formed by combining strength of MPI's data locality control and coarse grain parallelism style with taking advantage of strengths of UPC's fine grain approach, global address space and easy programming features.

We provided four different implementation codes of Cannon's algorithm employing four different parallelization paradigms, pure MPI, pure UPC, the hybrid MPI+UPC and the optimized hybrid MPI+UPC model on a cluster of SMP nodes. Each implementation employed the advance features of the underlining programming model to have the best performance gains.

We evaluated the performance and scalability of the purposed hybrid MPI+UPC model on the Cannon matrix multiplication benchmark by comparing the baseline pure MPI and pure UPC implementations on up to 100 cores. We also utilized MPI Cartesian topology and recognize that hybrid MPI+UPC implementation with MPI Cartesian topology can provide the locality control in which UPC needs and hence, improved the performance 2X comparing to UPC only.

Similarly the hybrid implementation demonstrated %20 performance gain on some configurations in comparison to MPI-only implementation by reducing intra-node communication overhead and also providing a larger message size since the UPC part of the hybrid handles the fine grain parallelism. Furthermore, overlapping UPC computations with MPI communications optimizes the hybrid code with an additional 20% performance enhancement over the benchmark.

References

[1] F. Blagojevic, *et al.*, "*Hybrid PGAS Runtime Support for Multicore Nodes,"* Fourth Conference on Partitioned Global Address Space Programming Model (PGAS10), Oct 2010.

[2] M. Forum, *"MPI: A Message-Passing Interface Standard,"* University of Tennessee Knoxville, TN, USA UT-CS-94-230, 1994.

[3] B. V. Protopopov and A. Skjellum, *"A multithreaded message passing interface (MPI) architecture: performance and program issues,"* J. Parallel Distrib. Comput., vol. 61, pp. 449-466, 2001.

[4] A. Skjellum, *et al.*, *"A Thread Taxonomy for MPI,"* presented at the Proceedings of the Second MPI Developers Conference, 1996.

[5] Y. Ojima, *et al.*, *"Design of a Software Distributed Shared Memory System using an MPI communication layer,"* presented at the Proceedings of the 8th International Symposium on Parallel Architectures,Algorithms and Networks, 2005.

[6] F. Trahay, *et al.*, *"An analysis of the impact of multi-threading on communication performance,"* in Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on, 2009, pp. 1-7.

[7] M. J. Quinn, *Parallel programming in C with MPI and openMP*. Dubuque, Iowa: McGraw-Hill, 2004.

[8] S. Chauvin, *et al.*, *"UPC Manual, v1.2"* High Performamnce Computing Laboratory, The George Washington University, May 2005.

[9] E. Lusk, "*MPI in 2002: has it been ten years already?,"* in *Cluster Computing, 2002. Proceedings. 2002 IEEE International Conference on*, 2002, p. 435.

[10] D. G. Elizabeth Wiebel, Steve Seidel, *"UPC Collective Operations Specifications, V1.0,"* UPC Consortium, December 12, 2003.

[11] G. Fox, Hey, A. J. G, Otto, S, *"Matrix Algorithms on the Hypercube I: Matrix Multiplication,"* Parallel Computing, vol. 4, 1987.

[12] https://computing.llnl.gov/tutorials/parallel_comp/#HybridMemory/ Accessed: 10.04.201

[13] K. A. Robbins, S. Robbins, "*UNIX systems programming: communication, concurrency, and threads (2 ed.),"* Prentice Hall PTR, 2003

[14] Multiprocessing Cots Technology Feature *" Distributed memory or shared memory? By Doug Clarke"* http://pdf.cloud.opensystemsmedia.com/vmecritical.com/Synergy. Jun04.pdf

[15] https://computing.llnl.gov/tutorials/mpi/#Virtual_Topologies/ Accessed: 10.05.2011

[16] J. Dinan, *et al., "Hybrid parallel programming with MPI and unified parallel C,"* presented at the Proceedings of the 7th ACM international conference on Computing frontiers, Bertinoro, Italy, 2010.

[17] W. Gropp and R. Thakur, *"Issues in developing a thread-safe MPI implementation,"* in Recent Advances in Parallel Virtual Machine and Message Passing Interface. vol. 4192, B. Mohr, et al., Eds., ed, 2006, pp. 12-21.

[18] R. Rabenseifner, *et al., "Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes,"* presented at the Proceedings of the 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing, 2009.

[19] Ohio Supercomputer Center, Parallel Programming with MPI http://www.osc.edu/supercomputing/training/mpi/Feb_05_2008/mpi_0802_mod_ topologies.pdf

[20] Y. Ojima, *et al., "Design of a Software Distributed Shared Memory System using an MPI communication layer,"* presented at the Proceedings of the 8th International Symposium on Parallel Architectures,Algorithms and Networks, 2005.

[21] Jost, G., Jin, H., Mey, D. a., & Hatay, F. F. *"Comparing the OpenMP, MPI, and Hybrid Programming Paradigms on an SMP Cluster"*. Germany: NAS Technical Report, 2003.

[22] T. M. Parks, *"A Comparison of MPI and Process Networks,"* presented at the Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 5 - Volume 06, 2005.

[23] L. He, *et al.*, *"MPI+OpenMP Implementation and Results Analysis of Matrix Multiplication Based on Rowwise and Columnwise Block-Striped Decomposition of the Matrices,"* presented at the Proceedings of the 2010 Third International Joint Conference on Computational Science and Optimization - Volume 02, 2010.

[24] D. A. Mallon, *et al.*, *"Performance Evaluation of MPI, UPC and OpenMP on Multicore Architectures,"* in Recent Advances in Parallel Virtual Machine and Message Passing Interface, Proceedings. vol. 5759, M. Ropo, et al., Eds., ed, 2009, pp. 174-184.

[25] L. Smith and M. Bull, *"Development of mixed mode MPI / OpenMP applications,"* Sci. Program., vol. 9, pp. 83-98, 2001.

[26] US DoD High Performance Computing Modernization Program (HPCMP) Waterways Experiment Station (WES), Dual-level parallel analysis of Harbor Wave response using MPI and OpenMP, http://www.wes.hpc.mil/news/SC98/HPCchallenge4a.htm and http://www.wes.hpc.mil/news/SC98/awardpres.pdf

Curriculum Vitae

Elif Öztürk was born on 21 September 1985, in Istanbul. She received her BS degree in Computer Engineering in 2007 from Kadir University. She worked as System Analyst in IngBank from 2008 to 2009 and then she worked as a research assistant at the department of Computer Engineering of Kadir Has University from 2009 to 2011. During her master education she has been affiliated with the Parallel Programming. Her research interests include data mining and computer networks.