CTUNES: A FRAMEWORK FOR SELF-ADAPTIVE, HIGH PERFORMANCE PARALLEL PROGRAMMING IN DISTRIBUTED SYSTEMS

By

TAI THE NGUYEN

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

MASTER OF SCIENCE IN COMPUTER SCIENCE

WASHINGTON STATE UNIVERSITY
School of Engineering and Computer Science, Vancouver

MAY 2016

To the Faculty of Washington State University:

The members of the Committee appointed to examine the thesis of TAI THE NGUYEN find it satisfactory and recommend that it be accepted.

Xinghui Zhao, Ph.D., Chair

Scott Wallace, Ph.D.

Sarah Mocas, Ph.D.

ACKNOWLEDGMENTS

First, I would like to express my sincerest gratitude to my supervisor, Professor Xinghui Zhao, who supported me throughout my research and study with her knowledge, guidance and encouragement. Without her consistent help, this thesis would not have been completed. I would also like to thank Dr. Sarah Mocas, whose courses and comments were invaluable.

Besides, I am grateful to Washington State University Vancouver for generously providing me with the assistantship. It was indispensable to help me become financially stable during my study at WSU Vancouver. I am also grateful to all WSUV's faculties to assist me in growing accustomed to studying at WSU Vancouver.

Last but not least, I thank my family who endured this long process with me, always offering love, supporting and understanding. Thanks are also due to numerous friends at WSU Vancouver.

CTUNES: A FRAMEWORK FOR SELF-ADAPTIVE, HIGH PERFORMANCE PARALLEL PROGRAMMING IN DISTRIBUTED SYSTEMS

Abstract

by Tai The Nguyen, M.S.
Washington State University
May 2016

Chair: Xinghui Zhao

The growing ubiquity of distributed system requires programmers to write communication programs that can be executed in parallel on multiple computers connected by a network. In addition, multicore architectures require programmers to write concurrent programs that can be executed in parallel on multiple cores in one computer. The combination of communication programs and concurrent programs is an emerging programming architecture, which is known as hybrid programming, or two-level parallel programming.

The traditional way of writing hybrid programs mixes network code, concurrency code, and functionality code. As a result, in order to exploit the parallelism of the underlying hardware, programmers must address critical challenges in their codes, such as transferring data, managing resources, and collecting the results. To simplify the task of writing efficient hybrid programs, we propose CTunes, a new software architecture for hybrid programming,

which addresses these challenges by separating programs' concurrency and networking code from their functionality code, and providing APIs for both communication and concurrency control.

Specifically, there are two main components of the proposed new software architecture of hybrid programming: a communication model and a programming paradigm. The new communication model handles distribution of computational tasks to nodes (or computers), and the new programming paradigm optimizes the level of concurrency of a computation at run-time based on predefined tuning policies in each node. Experimental results show that CTunes is effective in achieving high performance on high-end computing architectures, yet it does not introduce extra overhead on the hardware which does not provide high level of parallelism.

TABLE OF CONTENTS

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Challenges in High Performance Parallel Programming

Two widely used parallel programming models are Message Passing Interface [1, 2] (MPI) and OpenMP [3]. MPI is a standardized and portable message-passing system on distributed memory systems. It is popularly utilized for writing message passing programs across multiple computers connected by a network. In contrast, OpenMP is a portable approach for parallel programming on shared memory systems.

There are three main challenges in programming with the MPI model . First, the discrete memory view in the MPI model makes writing code difficult. Specifically, a MPI programmer must cautiously scrutinize good strategies for partitioning the data to each process and collecting the results from processes through message passing. Second, due to the distribution of the model, processes do not share data with each other. In other words, if the same input data is needed by each process, the data must be replicated. As a result, the overall memory demand is significantly increases within each node. Third, running a very large job using

MPI on a large scale parallel system is very challenging due to the insufficient support fault tolerance in the model [4]. Because of these challenges, the MPI model always exhibits high demand for resources, such as memory and network.

As an improvement to the basic MPI model, a shared memory system can be added to serve as the dominant parallel architecture within each node. One example is a combination of MPI and OpenMP. The hybrid MPI-OpenMP [5] approach supports multiple levels of parallel computing on a symmetric multiprocessing cluster or network of nodes, where MPI is used to handle parallelism across nodes, and OpenMP is employed to exploit parallelism within a node. The advantage of this hybrid model is that it can maintain cross-node performance with MPI and reduce the number of MPI processes required within a node. As a result, the hybrid MPI-OpenMP approach can help reduce the demand for network bandwidth by using one MPI process per node. Hence, this model can be used to run large jobs. However, the hybrid MPI-OpenMP model can not achieve high performance due to the inefficiency of OpenMP [6, 7, 8, 9]. This is discussed in greater detail in Section 2.3.1.

In general, the development of high performance and parallel software is extremely difficult because of the complexities involved in design, implementation, debugging, and deployment [10], especially when using a low-level language, such as C or Fortran.

1.2 Challenges in Exploiting Concurrency in Multicores

The recent advances in hardware technology have shown the potentials of multicore processors in achieving high performance while avoiding the power demands caused by increasing clock speeds . However, in order to fully exploit these potentials, programmers must write concurrent programs to effectively utilize the underlying hardware resources of multicore

processors [11]. Writing parallel or concurrent programs is hard, because it requires a different way of thinking that human brains find difficult [12], not to mention writing good concurrent programs that can efficiently utilize underlying multicores of hardware resources. Recent studies show that no matter how many cores there are in the processor, for most of the desktop applications, only 2 or 3 cores are more than adequate, and others are underutilized [13]. The mismatch between user programs and the underlying hardware presents challenges in leveraging the full power of the multicore technology.

Worst of all, even if an application is well written in a way that it can efficiently utilize all the cores, when it is executed on another multicore processor with a different number of cores, this mismatch may appear again, resulting in performance degradation. The reason for that is because the traditional way in which concurrent programs are written limits their flexibility to adapt to different hardware for better performance. Specifically, the concurrency in user programs is usually achieved by using threads of execution of programmed instructions.

A thread or process can be simply considered as a task that needs to be carried out by the CPU. A concurrent program consists of multiple threads which can execute in parallel. This type of concurrency is also called thread-level parallelism. Because thread level parallelism is traditionally always hard-coded in the program, its level of concurrency (i.e., number of threads) never changes at runtime, despite the difference of the underlying hardware on which it executes. The fundamental reason is that, when the concurrency code is mixed with the function code, it is hard to efficiently utilize the underlying parallelism provided by hardware resources, in a flexible way.

1.3 Thesis Statement

Concurrency management and control in parallel programming can be separated from its functionality, and this separation, if combined with careful distribution of computational tasks,

will lead to opportunities of achieving high performance on different hardware without changing users' programs. This effort can greatly simplify programmers task of writing efficient parallel software which runs on a distributed system.

1.4 Contributions

The contributions of this thesis are three-fold. First, we developed Dynamic Tuning, a new paradigm to obtain high performance automatically at runtime [14]. Second, we designed a new communication interface for distributed systems, named Function Communication Model which provides APIs for distributing tasks to and collecting results from a network of computational nodes. Finally, we combined these two components to build a new framework, CTunes, for supporting programming concurrency in distributed systems with minimum effort from the programmers. These contributions are explained in greater detail as follows.

- **Dynamic Tuning (DT)** is a new programming paradigm to obtain high performance by separating concurrency from the functionality code of a program. We developed a tool which serves as a tuning knob in between user programs and the underlying hardware by dynamically adjusting the thread-level concurrency at runtime based on different tuning policies. These tuning polices can be implemented separately from user programs as plug-in modules at runtime. Dynamic Tuning for controlling level-concurrency threads for executing in parallel is employed to exploit parallelism within a node.

- **Function Communication Model (FCM)** is a new communication model of parallel programming by distributing tasks. It supports sending data to each node, sending a message, and calling a function with the resultant data to a computation node through the network. A session ID, which is created in initialization, is communicated to all

computation nodes through the network.

- **CTunes** is a new framework which combines DT and FCM to achieve high performance on a network of computers. It greatly simplifies the task of writing high performance, scalable parallel programs on a distributed system. It can be employed in any network of computers such as a workstation, a cluster, or a network of desktop computers. FCM is used to handle communication across computation nodes and Dynamic Tuning is used for controlling thread-level concurrency to exploit parallelism within a node. Three case studies with different types of computations are used to show CTunes can obtain a high performance in a distributed system when compared to traditional MPI and OpenMP approaches. It also outperforms hybrid MPI-OpenMP, which combines both parallel programming models.

1.5 Thesis Organization

The rest of this thesis is organized as follows. In Chapter 2, we review related work, including the MPI model, the hybrid MPI-OpenMP model, and the hybrid MPI-Pthreads model. In Chapter 3, we introduce CTunes, a new framework for high performance parallel programming. Chapter 4 compares the API and the programming styles of CTunes, pure MPI, the hybrid MPI-OpenMP, and the hybrid MPI-Pthreads. To evaluate our framework, experiments have been carried out on three case studies. The results of these experiments are discussed in Chapter 5. In Chapter 6, we conclude our work, and present future directions of this research.

Chapter 2

Related Work

2.1 Shared Memory Parallel Programming

In shared-memory multiprocessor architectures, multiple threads can be used to implement parallelism. Each thread has a set of private variables, for example, local stack variables. Threads can also share a set of variables, such as static variables, shared common blocks, and a global heap. In shared-memory parallel programming, threads communicate implicitly by writing into and reading from the shared variables. Threads coordinate by synchronizing their read and write operations on shared variables. There are five popular programming models for shared-memory parallelism:

- POSIX threads (Pthreads) [15, 16] is a very popular API for multiple threading programming. It is a standardized C language threads programming interface. It also is an interface to operating system utilities. It is portable. However, it is relatively heavy-weight, which results in relatively low performance. Pthreads is supported uniformly across UNIX-like OS platforms. Pthreads contains support for creating parallelism and synchronization, as well as implicit support for communication in shared memory via a pointer to shared data which is passed among threads. Since Pthreads is lower

abstraction than OpenMP, Pthreads can achieve higher performance than OpenMP [17].

- OpenMP is a standard for application level programming. It supports scientific programming on shared memory. Parallelism with OpenMP is much easier than that with Pthreads because when Pthreads is used, the programmer must deal with low-level details of thread creation, management and synchronization. The advantages of OpenMP make it generally more suitable for parallelizing and debugging than Pthreads. On POSIX systems, the OpenMP runtime is built on top of Pthreads.

- Thread Building Blocks [18] is developed by Intel. It supports parallelism threading for C++ language. It takes full advantage of multicore performance, is portable and composable, and has future-proof scalability.

- CILK [19] is an extension to C that offers a quick and easy way to harness the power of multicore. It is a lightweight thread manager.

- Java threads [20] is built on top of POSIX threads. It is an object in Java language.

2.2 Distributed-Memory Parallel Programming

The parallelization in distributed memory computing is executed by several processes. Each process has a private space of memory that the other processes cannot access. Data must be shared using the communication network. All these processes, distributed across several computers, processors, and/or multiple cores, are the small parts that together build up a parallel program in the distributed-memory architecture. In distributed memory parallel programming, a parallel program is a collection of processing elements, or processes, that cooperate to solve large problems.

2.2.1 Message Passing Interface

MPI supports a programming model for sending and receiving point-to-point and collective communication messages between processes for programming in distributed-memory architectures. MPI implementations exist for virtually all popular parallel computing platforms. It is a commonly used, effective, and powerful programming model for an interconnected network of computers and clusters, but its low-level abstractions are challenging to program [21]. Parallel programming libraries and platforms for MPI are currently the most popular standard for parallel programming because the most important advantages of this model are achievable performance and portability. Good performance is achieved by utilizing the hardware support as a direct result of the optimized MPI libraries available to users' programs [22, 23]. Portability arises from the standard API and the existence of MPI libraries on a wide range of machines.

The MPI model considers that the underlying hardware is a collection of processors, each with its own local memory, and these processors can communicate with each other through an interconnected network. A process can only access the instructions and data stored in its own local memory. Processes pass messages to communicate and/or synchronize with each other. Figure 2.1 shows the architecture of the MPI model. A master process has its own local data, and it can send a message containing some of its local data values to a slave process through the network, giving the slave process indirect access to these values.

In order to harness the power of multicore architectures, MPI programs are often initialized with the number of processes being set equal to the number of cores in each node. Recently, the number of cores per chip has been continuously increasing; however, memory capacity and network performance are not able to keep up the same pace. Because the number of cores is increasing, we often increase the number of processes, so a memory capacity per process is decreasing. Hence, the MPI programming model requires the programmer to

carefully assign each process to the problem by the MPI rank. In addition, by using memory-to-memory copies to share data of MPI calls between MPI ranks, the memory bandwidth is a serious problem for message passing. Moreover, the current network bandwidth is not able to support sending and receiving a message containing large data between all cores on all nodes [24]. As a result, a hybrid model mixing distributed-memory MPI model with shared-memory OpenMP model is proposed to overcome these limitations [6, 25]. These approaches are reviewed in section 2.3.

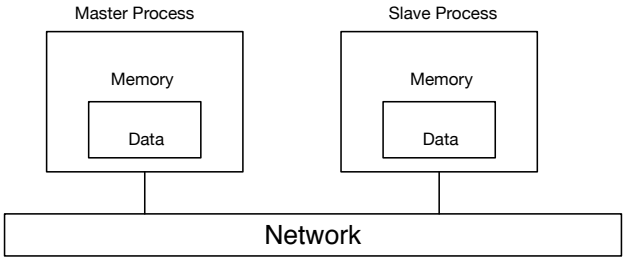


Figure 2.1: Architecture of MPI model

2.2.2 Actors

Actor is a model of concurrency. The Actor model provides a convenient way to develop concurrency, and distributed computing. In the Actor model, autonomous concurrently executing objects, called actors, communicate with each other using buffered, asynchronous, point-to-point messages. An actor encapsulates a state, a number of methods (which can change the state of the actor), and a thread of control. Actors are distributed over time and space. Each actor has a globally unique mail address, and it maintains a queue of unprocessed messages it has received. Figure 2.2 shows the structure of an actor.

The Actor model is one of the message passing models in programming concurrency. The Actor model encapsulates objects along with threads of execution. Therefore, earlier actor frameworks usually use one-thread-per-actor implementation of actors, such as Scala

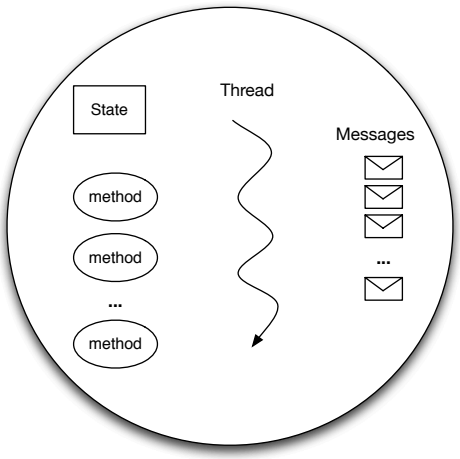


Figure 2.2: Structure of an Actor

[26] and Actor Architecture [27]. However, it turns out that in practice, one-thread-per-actor implementation of actors is not particularly efficient, because of the overhead caused by context-switching among actor threads. It is more efficient to have a pool of threads, where each thread processes messages for multiple actors in some order. Karmani et al. implemented this optimization strategy in the latest version of ActorFoundry [28]. So, it provides a convenient and less error-prone way to write concurrent programs. However, a faithful implementation of the Actor model results in high overhead.

By encapsulating each object along with a thread of execution, the Actor model helps eliminate some common hard-to-debug problems in concurrent programming, such as race conditions and deadlocks. This is simply because actors do not share state, and the actor messages are asynchronous. The fact that no state/data is shared by two actors prevents race conditions, and asynchronous message passing avoids deadlocks resulting from circular waiting. Since actors do not share state, the only way that actors communicate with each other is through message passing. Processing messages for an actor is the computation carried out by the encapsulated thread, which represents a sequential computation. In the Actor system, a number of actors can execute concurrently, representing concurrent

computations.

Actors languages and frameworks are developed and utilized underlying a number of languages, such as Erlang (from Ericsson) [29], SALSA (UIUC and RPI) [30], E language [31], Axum (Microsoft) [32], and Ptolemy (UC Berkeley) [33]. In addition, the Actor model is developed using existing programming languages, such as C/C++ (Act++ [34] , Broadway [35], Thal [36]), Java (Scala Actors Library (EPFL) [26], Kilim [37], ActorFoundry [38], and Actor Architecture [27]).

2.3 Hybrid Approaches

2.3.1 Hybrid MPI-OpenMP

The hybrid MPI-OpenMP programming model can be considered as a hierarchical two-level parallel architecture since it combines both features of shared and distributed memory of interconnected networks of computers. Hence, a combination of both shared memory and message passing parallelization paradigms within the same application, hybrid MPI-OpenMP programming may provide a strategy for better exploiting distributed shared-memory architecture than the basic MPI.

In the hybrid MPI-OpenMP model, communication between nodes is handled by MPI processes, and each MPI process has some OpenMP threads running inside to occupy the CPUs in a single node. In order to effectively utilize underlying hardware support of multiple cores in OpenMP, the number of OpenMP threads is set to be the number of cores in a single node, and there is only one MPI process in each node. This programming style involves a hierarchical model with MPI parallelism appearing at the top level, and OpenMP parallelism appearing below. Figure 2.3 shows the architecture of the hybrid MPI-OpenMP model. A master MPI process sends data to the MPI process at each node. Each node has one single

MPI process which further creates multiple OpenMP threads (for example, four threads at a four-core node).

In the hybrid MPI-OpenMP model, MPI processes can send data to each other. This is called process-to-process communication. In this communication, MPI routines are invoked outside OpenMP parallel regions, thus there is only MPI communication across nodes. OpenMP threads can share data with each other within a node, called thread-to-thread communication. In this communication, some MPI routines are inside OpenMP parallel regions, leading to OpenMP threads involvement in inter-node communication by direct access to the shared memory.

The important advantages of this hybrid MPI-OpenMP approach are a reduction of memory usage within a node, and avoiding an overhead of memory bandwidth and network bandwidth from sending and receiving data for processes created by MPI calls. However, there are two limitations of this approach: no interaction between MPI processes and OpenMP threads, and a limited performance of OpenMP, especially in a large scale system. Since OpenMP libraries can not effectively utilize the underlying hardware resources in a large-scale system, this approach is not able to achieve good scalability.

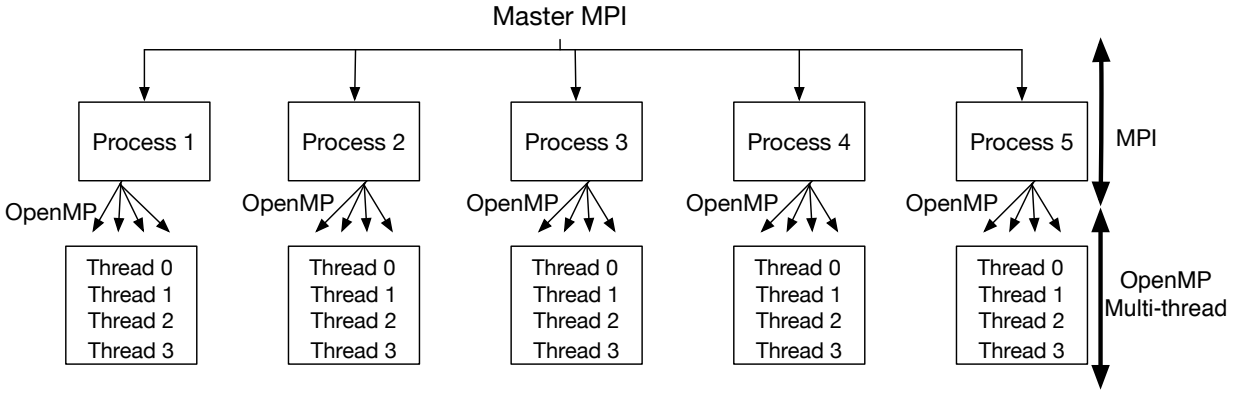


Figure 2.3: Hybrid MPI-OpenMP Programming Model

2.3.2 Hybrid MPI-Pthreads

The hybrid MPI-Pthreads programming model is the same as the hybrid MPI-OpenMP programming model. With Pthreads being used to occupy the CPUs of a node, instead of OpenMP threads. Same as in hybrid MPI-OpenMP, the number of threads is mapped to the architecture of a node in network.

2.4 Programming Languages

C and Fortran are system programming languages. Compared with C or Fortran, the advantages of the Java [39] programming language include built-in support for multi-threading, high-level programming concepts, improvement of compile-time, and runtime checking. As a result, the code in Java is good for problem detection, debugging, and overall coding productivity. In addition, the performance gap between Java and native system programming languages has been narrowing for the past few years [40].

In this thesis, we propose a new platform based on a high-level language which is easy to design, develop, debug, and maintain in the development of high computing and scalable parallel software. Across nodes, the platform supports sending tasks to each computation node. Also, in each node, it provides tuning knob in between user programs and the underlying hardware which dynamically adjusts the thread-level concurrency at runtime based on different tuning policies. As a result, this platform can harness the power of multicore processors within a computation node to achieve high performance in each node, resulting in the overall performance of the platform achieves high performance in distributed systems. The overall performance of this platform is better than the performance of MPI and hybrid MPI-OpenMP, though the communication of this platform may introduce extra overhead comparing to MPI and hybrid MPI-OpenMP.

Chapter 3

Approach

3.1 System Design and Architecture

CTunes is a new platform for high performance parallel programming in distributed systems. Figure 3.1 depicts the architecture of the CTunes platform, which is a combination of Function Communication Model (FCM) and Dynamic Tuning of thread-level concurrency (DTTLC). FCM is a new communication model for parallel programming which utilizes a master-slave communication pattern. FCM supports sending data to each slave (or computation) node, sending messages, and calling a function with the resultant data sent to a computation node through the network. DTTLC is a concurrency tuning policy which dynamically adjusts the level of concurrency based on the feedback from applications at runtime. CTunes supports two levels of parallel computing on a network of computers. Specifically, FCM is utilized to handle communication across computation nodes through the network, and DTTLC is employed to exploit suitable levels of parallelism within a node.

The CTunes programming model can be considered as a hierarchical two-level parallel architecture because it combines features of both shared memory and task distribution over a network. In CTunes, sending a task from a master node to slave nodes is handled by FCM,

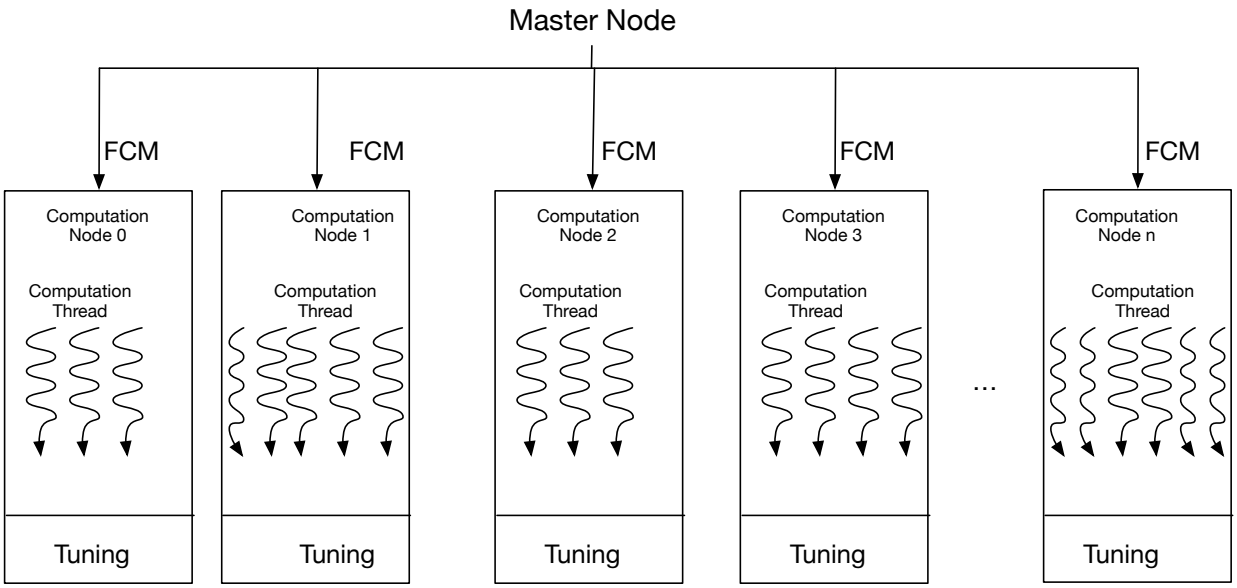


Figure 3.1: CTunes Architecture

and an individual task at each slave node is executed in parallel by several DTTLC threads. DTTLC threads utilize the CPUs of a computation node through self-adaptive, thread-level concurrency tuning.

The number of DTTLC threads is dynamically adjusted by a runtime tuning policy deployed on each node, based on the feedback from the applications. This programming style involves a hierarchical model: FCM parallelization occurring at the network level, and DTTLC parallelization occurring at the individual node level. In this model, the initial level of concurrency is set based on the hardware configuration of each computation node. The dynamic tuning policy enables programs to adapt to different hardware for achieving better performance in each computation node.

3.2 Function Communication Model

FCM is built on top of the socket programming API. Figure 3.2 depicts the structure of FCM. The computation node can serve function calls.

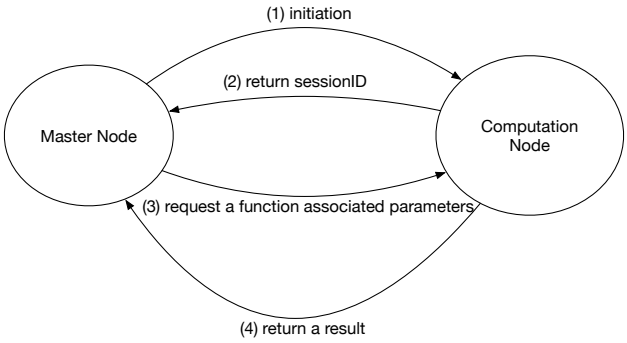


Figure 3.2: The FCM Model

During initialization, each computation node opens a listener socket at a specific port, as specified in a configuration file, and the master node searches for a desired computation node in the network. Next, the master node establishes communication by sending an initiation message to the computation node that has been chosen. After the communication is established, it is assigned to a session ID, which is an identifier representing a particular connection between the master node and the computation node. Through the established FCM communication, the master node can send data, call a function, or send a message to the computation node.

Two fundamental components in the FCM are function calls and session IDs. A function call includes a function name and its parameters on a particular computation node. The initialization of a function call includes a server class of computation, an observation class of dynamic tuning, a return class of computation, and a subset tasks class of computation. After initialization, the master node can use the session ID to communicate with computation nodes. Specifically, the session ID is attached as a parameter in each message/call to allow a master node to communicate with a computation node in synchronous or asynchronous mode.

3.2.1 The Function Communication Model API

The functions that comprise the FCM API fall into four main categories: initialization and termination of the connection, FCM call function with a result, FCM sending messages, and sending data. The *initialize* and *close* connection functions are called before and after a computation. All other functions are called in between the *initialize*, and before the *close* function. Table 3.1 summarizes the FCM APIs.

Table 3.1: Function Message Interface APIs

FCM initialize and close connection functions	
initialize	reads the configuration file, initializes the required modules, and sends parameters to accept the connection. After this function, client and server has the same session ID.
process	executes initialization, computation, and close functions.
startup	executes tasks after the master node are connected to the computation nodes
close	releases any resources being used, and closes a connection between client and server.
FCM call functions	
sync/call	calls a synchronized function with parameters. It can have a result without compression.
syncWithReturnCompress	calls a synchronized function with parameters. It has a result with compression.
FCM send messages	
async/send	sends an asynchronized message with parameters.
FCM send data	
broadcastArrayData	broadcasts an array of objects to all computation nodes.
broadcastDataInChannel	broadcasts an object to all computation nodes by high performance socket.
asyncsendArrayDataInCache	broadcasts an array of objects to all computation nodes by distributed cache memory.
sendDataInCache	broadcasts an object to all computation nodes by distributed cache memory.
broadcastData	broadcasts an object to all computation nodes by socket.
waitToSendAllData	blocks until all of the specified requests in a given set have completed.

The semantics of FCM can be described as follows. Consider a FCM program that consists of a master node and a set of computation nodes. After initialization, the master node and each computation node have the same session ID, the master node can communicate with any computation node by sending asynchronous (non-blocking) messages using

the send function; send (message) has the effect of eventually appending the contents of message to the buffer of the desired computation node. However, the send function returns immediately without waiting for the message to arrive at its destination since FCM operates asynchronously. Because of the network delays, the arrival order of messages is non-deterministic. However, we assume the messages will eventually be delivered.

At the beginning of the execution, the buffer of each computation node is empty. All computation nodes are initialized and ready to receive messages at the socket port specified in the initialization. Each computation node can be viewed as executing a loop in the following steps: remove a message from its buffer (implemented as a queue), decode the message, and execute the corresponding method. If a computation node's buffer is empty, the computation node is blocked waiting for the next message to arrive in the buffer.

3.2.2 Synchronization

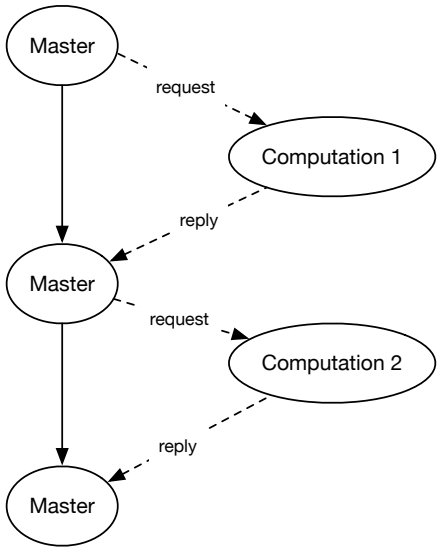


Figure 3.3: A master node requesting quotes from multiple competing services using RPC-like communication. The dashed slanted arrows denote messages and the vertical arrows denote that the master server is waiting or is blocked during that period in its life.

Synchronization in FCM is achieved through blocking communication using Remote Pro-

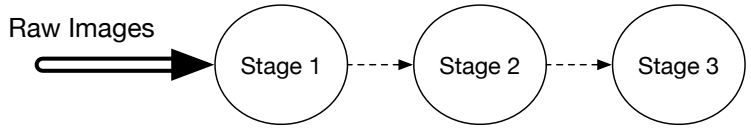
cedure Call (RPC)-like message-passing. RPC [41] is a protocol that one program can use to request a service from a program located in another computer over the network without having to know network details. A procedure call is also known as a function call or a subroutine call.

In RPC-like communication, the sender and the receivers have the same session IDs after initialization. After sending a message, the sender waits for the result to arrive before it proceeds with processing other messages. For example, we consider the pattern of Figure 3.3 in FCM call functions. A master node calls (requests) a function from computation node 1. The master node waits for a result. After it receives the result, it calls another function at computation node 2.

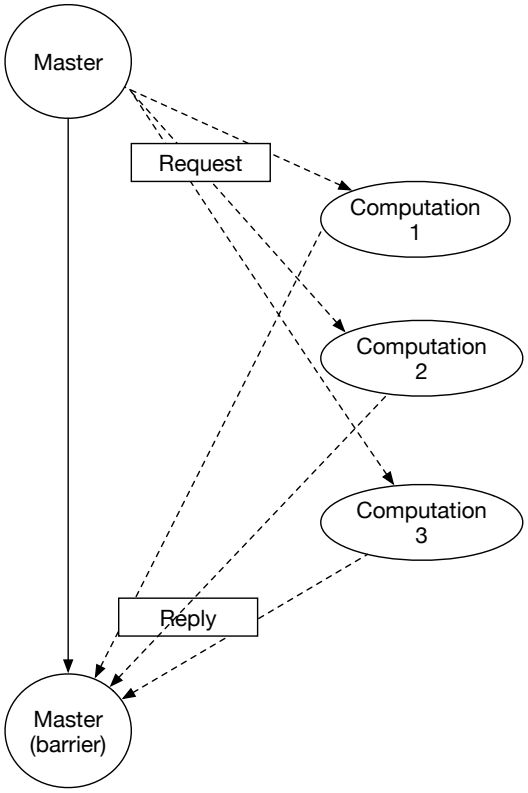
RPC-like message-passing is universally supported in FCM libraries. FCM call functions are particularly useful in two scenarios. One scenario occurs when a master node wants to call a sequence of functions in a specific order to a particular computation node. In this case, it wants to ensure that a result's the function is received before it calls another. A variant of this scenario is where the master node wants to ensure that the computation node has received a function before it communicates this information to another computation node. The second scenario is when the variable of the master node is dependent on the reply function it expects from a computation node. For example, when a master node calls function to obtain the resultant data from all computation nodes, the master node needs to wait for collecting the resultant data from all computation nodes, after which it can continue to return the resultant data to a user.

3.2.3 Patterns of FCM Programming

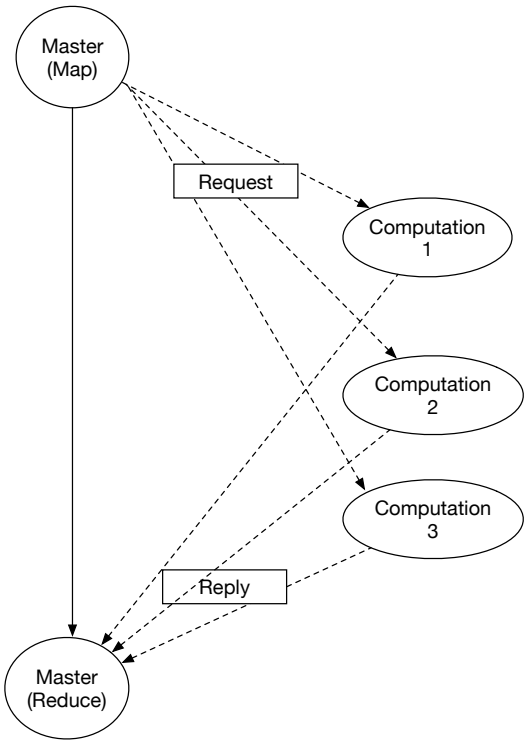
Two common patterns of parallel programming are pipeline and divide-and-conquer[42]. These patterns are shown in Figure 3.4.



(a) An example of divide-and-conquer pattern of Pipeline pattern



(b) An example of divide-and-conquer pattern of Parallel-and-barrier



(c) Map-reduce

Figure 3.4: Patterns of FCM programming

An example of the pipeline pattern is an image-processing network (see Figure 3.4 (a)). This is a stream of images which are passed through a series of filtering and transforming stages. The output of the last stage is a stream of processed images. This pattern has been demonstrated by an image-processing example, written using the Asynchronous Agents Library[43].

A parallel-and-barrier graph is an example of the divide-and-conquer pattern (see Figure 3.4 (b)). A master node divides tasks onto a set of computation nodes and waits for the output from each these computation nodes in the "join barrier" behavior of the master node.

A map-reduce graph is another example of the divide-and-conquer pattern (see Figure 3.4 (c)). A master node maps the computation onto a set of workers and the output from each of these workers is reduced in the "join continuation" behavior of the master node.

3.2.4 FCM Example

In order to show how FCM programs work, we use a **HelloWorld** example as shown in Listing 3.1. A class defining a master node's behavior extends **MasterNode** class, and a class defining a computation nodes behavior extends **ComputationNode** class. Messages are handled by methods which are accessible from other class, i.e, public methods. A session ID is automatically created to facilitate the communicate between a master node and computation nodes after the initialize() function. The process() function in HelloMasterNode class executes initialization, computation, and close functions.

Listing 3.1: Hello World program in FCM

```
public class HelloMasterNode extends MasterNode {
        protected void startup() throws Exception {
                System.out.println("Hello in client node");
                send("audience");
```

```
            String p =(String)call(nodes_str[0], "audiencewithparameter", "world");
            System.out.println("Computation Node return "+ p);
        }
        public static void main(String[] args) throws Exception {
            HelloMasterNode masternode = new HelloMasterNode();
            masternode.setNodeTask(WorldNodeTask.class);
            masternode.initialize();
            masternode.process();
            masternode.close();
        }
}
public class WorldComputationNode extends ComputationNode {
        public void audience() {
            System.out.println("Hello in Computation node");
        }
        public String audiencewithparameter(String name) {
            System.out.println("Hello " + name + ", in Computation node");
            return "FCM message";
        }
}
```

Listing 3.1 shows a HelloWorld program. The program comprises two classes, the **HelloMasterNode** and **WorldComputationNode**. **HelloMasterNode** is the definition of a master node; it executes in the master node. **WorldComputationNode** is the definition of any computation node in the network; it can execute in computation node upon receiving a request from either a master node or any computation node. An instance of the **HelloMasterNode** can send one type of message, the audience message, which triggers the execution

of the audience method in all computation nodes. On receiving an audience message, the **WorldComputationNode** executes the method audience with no parameters.

Next, the **audiencewithparameter** method is called; it is a synchronous message with a parameter and it returns a string value to the first computation node. The output string is "Computation Node return FCM message".

3.3 Currency Tuning

For decades, advances in hardware technology have long been the main reason for which software evolves. Computer architects are shifting processor design to multicore architectures - multiple CPUs on a single chip - for improved power efficiency. This is motivated by the relationship between a processors speed and its power requirement: the power consumed by a core is typically proportional to the cube of its frequency. In particular, multiple cores running at a lower frequency can deliver the same performance as can be achieved from a single core running at a higher frequency, while consuming less power. As with all other major changes in hardware technology, this shift is also calling for corresponding changes in software: programmers now must write concurrent programs which can be executed in parallel on multiple cores [44]. We have developed two different approaches to use hardware efficiently: static tuning which makes the decision at compile-time, and dynamic tuning which dynamically adjusts the level of concurrency at runtime.

3.3.1 Static Tuning

Static Tuning is a tuning policy which maps to the architecture of a computation node, and then makes decisions on the suitable level of concurrency for user programs. For example, if there are 12 cores in a cluster, and then the number of worker threads in the thread pool is 12. Figure 3.5 shows the static tuning policy.

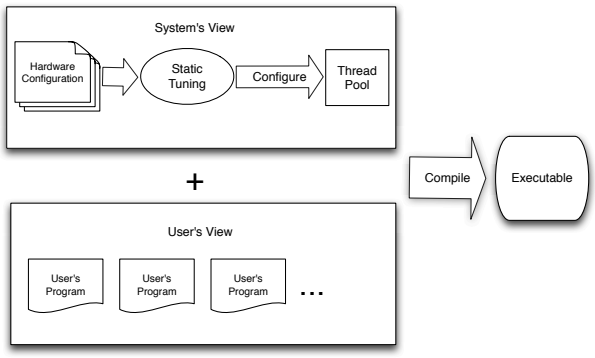


Figure 3.5: Static Tuning

3.3.2 Dynamic Tuning

When the applications have complex synchronization requirements, or the hardware resources are not continuously available, e.g., many different applications share the hardware resources, it may not be possible to analyze the near optimal level of concurrency for the applications at compile-time. In this case, we can use a runtime dynamic tuning policy to dynamically adjust the level of concurrency based on the feedback from user applications.

Figure 3.6 shows the dynamic tuning policy which should be enforced at runtime. Specifically, the dynamic tuning policy initializes the thread pool by setting the number of worker threads to be a small number, i.e., 1. During runtime, it increases the number of worker threads, and observes the progress of the user applications. If the feedback is positive, which means increasing the number of worker threads results in better performance than before, the dynamic tuning increases the number of worker threads again, until the feedback from user application becomes negative, which indicates that a balance has been reached. At this point, increasing the number of worker threads again usually results in negative impact on the overall performance, because of unnecessary context switching.

Dynamic tuning is different from static tuning in that it does not require prior knowledge about the available hardware resources. It is performed at runtime and adjusts the level of

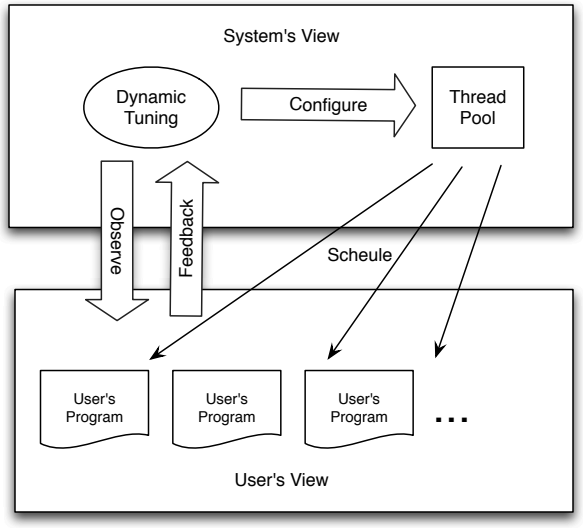


Figure 3.6: Dynamic Tuning

concurrency dynamically based on the progress of the user application. Therefore, it is a feedback based approach, and it is more flexible in adapting to changes of resource availability in the system. However, it does require that the progress of application can be evaluated at runtime. For a large class of iteration based problems, such as scientific computing problems, the progress of application can be simply represented by the number of iterations that have been completed.

There are three policies of Dynamic Tuning. They are CPU Usage Tuning, Performance Tuning, and Hybrid Tuning, described in the following sections.

3.3.3 Dynamic Tuning - CPU Usage Tuning

In CPU Usage Tuning, the system dynamically changes the number of threads based on the progress of the computation. For the CPU Usage, we start a system with the number of worker threads which is equal to the number of cores in the architecture of this node. Depending on the progress of the computation, we dynamically add threads to the system as needed, based on the observation of the feedback from the user application on the CPU

Usage. We check whether the current computation has CPU Usage which is less than δ. If so, adding more threads to the system may result in better performance. Otherwise, we stop adding more threads to the system.

In our implementation, we use $\delta = 93\%$.

3.3.4 Dynamic Tuning - Performance Tuning

In performance tuning, the system dynamically changes the number of threads based on the progress of the computation. For the computation, we start the system with the number of worker threads which is equal to the number of cores in this system. Depend on the progress of the computation, we dynamically add threads to the system as needed, based on the observation of the feedback from the performance of the user application. We check whether the execution time of the current iteration is less than that of the previous iteration by δ. If so, adding more threads to the system may result in better performance. On the other hand, if the execution time of the current iteration is greater than that of the previous iteration by δ, we decrease one thread to the system which may result in better performance. Otherwise, if the current execution time is not less by δ comparing to the previous iteration, we stop changing the number of threads to the system.

In our implementation, we use $\delta = 5\%$.

3.3.5 Dynamic Tuning - Hybrid Tuning

Hybrid Tuning is the combination of CPU Usage policy and Performance policy. There are two stages of this policy.

In the first stage, we use CPU Usage policy. For the computation, we start the system with the number of worker threads which is equal to the number of cores in this system. Depending on the progress of the computation, we dynamically add threads to the system

as needed, based on the observation of the feedback from the user's application based on the CPU Usage. We check whether the current computation has CPU Usage which is less than δ. If so, adding one more thread to the system may result in better performance. On the other hand, it stops this stage of CPU Usage policy. At this point the system moves to the next stage.

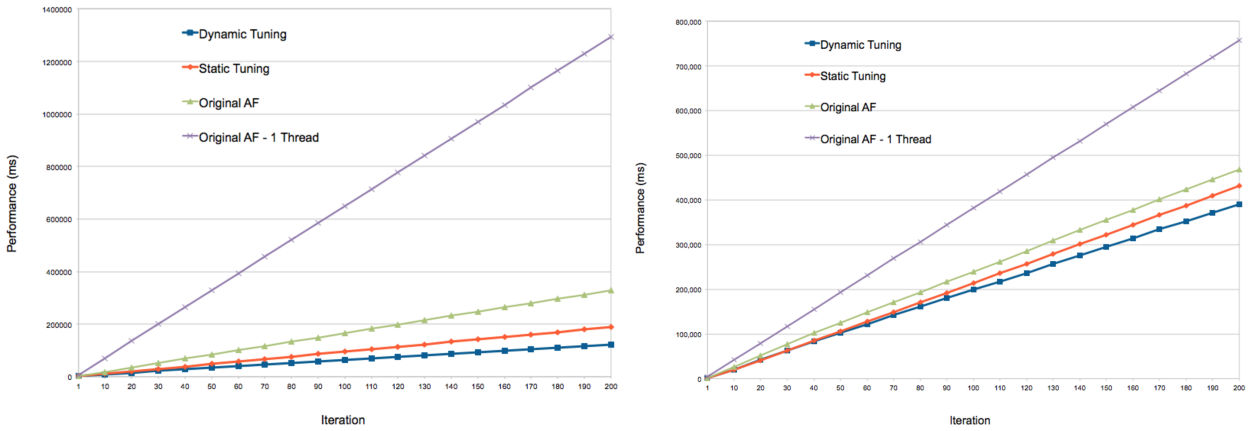
In the second stage, we use Performance Tuning to control the number of threads to improve the performance. We check whether the execution time of the current iteration is greater than that of the previous iteration by β. If so, decreasing one thread to the system may result in better performance. Otherwise, if the current execution time is not less by β comparing to the previous iteration, we stop changing the number of threads to the system.

In our implementation, we use $\delta = 93\%$, and $\beta = 5\%$.

3.3.6 Comparison of Tuning Policies

In a pilot study, we have investigated the effectiveness of our tuning policies in achieving high performance on different hardware. Two types of hardware are used in these experiments, an iMac desktop and a high performance cluster. The iMac desktop has an Intel i5 CPU (4 cores) @ 2.7GHz, and 8GB RAM. The cluster has dual 6-core Intel Xeon CPUs (12 cores in total), and 16 GB RAM.

We illustrate the effectiveness of dynamic tuning using the Gravitational N-Body Problem which will be discussed in greater detail in Chapter 5. We have implemented N-Body in ActorFoundry[28], an java implementation of the Actor model. We run a 20000-body computation on both hardware for 200 iterations. The results are shown in Figure 3.7. In both cases, dynamic tuning outperforms the original ActorFoundry, and achieves the best performance among the three. Note that AF-1 Thread is a sequential implementation of the problem, for comparison purpose.



(a) Desktop (b) Cluster

Figure 3.7: Performance Comparison of Tuning Policies

In addition, to investigate the scalability of dynamic tuning policies, we have carried out experiments on N-Body computations with various sizes from 10k to 80k using the cluster. The results are shown in Figure 3.8. When the computation size increases, Dynamic Tuning - Hybrid Tuning policy shows better scalability.

Based on these results, we chose Dynamic Tuning - Hybrid Tuning in our case studies.

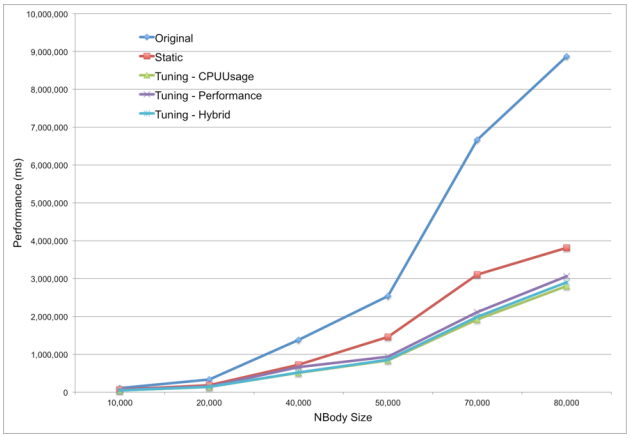


Figure 3.8: Scalability Comparison of Tuning Policies

3.4 CTunes Implementation

We combined FCM and Dynamic Tuning to build a new Distributed Computing System, CTunes. The operation of CTunes has four phases as follows:

1. In the first phase, the master node reads the configuration file to specify the computation node in the network.

2. The master node uses FCM to send a request for an initialization communication between the master node and a set of computation nodes in the connection phase. The request of computation includes a type of concurrency tuning, a class of computation node, a class of model result, and a class of sending task. The type of concurrency tuning can be Static Tuning, CPU Usage Tuning, Performance Tuning, or Hybrid Tuning. All computation nodes use the request of the initialization computation to create a new instance to execute tasks received from the master node. After this phase, the master node and each computation node have the same session ID to communicate.

3. In the execution phase, the master node assigns tasks to the computation nodes. The master node sends requests to the computation nodes to collect the data. The computation nodes execute tasks with a pre-defined type of currency tuning, and store the results for the master node to collect. The master uses a barrier for collecting the data with a constraint that the number of tasks to send to the computation nodes is equal to the number of the result data received from the computation nodes. Note that the result data can be empty if needed. Once passing the barrier, the computation nodes proceed to the next phase.

4. In the final phase, the master node collects and processes all the results from the computation node.

Chapter 4

Programmability Evaluation

4.1 Programming Models

MPI is the most well known and widely used programming model for data-parallel applications using message passing. The hybrid models built on top of MPI provide the benefit of exploiting multiple levels of parallelism. For example, the hybrid MPI-OpenMP has both node-level parallalism (provided by MPI), and in-node parallelism (provided by OpenMP).

Comparing to these models, CTunes has several advantages. It provides programmers an easier way to achieve suitable parallelism on different hardware without extra effort. It is developed in Java, an object-oriented language. In addition, it provides potentials for supporting fault tolerance mechanisms because the tasks are small and independent, which can be easily rolled back. As all other message passing models, network performance can be a bottleneck for CTunes and cause delays in communication among nodes. When this happens, a buffer of sub-taks can be used to improve the performance. Table 4.1 summarizes some important characteristics of MPI, hybrid variants and CTunes programming models.

Table 4.1: Programming Model Comparison

Characteristics	**MPI and Hybrid Variants**	**CTunes**
Language	System programming language	Object-Oriented programming language
Model	SPMD (SPMD is the technique that uses parallel programming to split up tasks and run different input simultaneously on multiple processors)	Client/Server
API	MPI, plus OpenMP or Pthreads for hybrid	FCM API
Parallelism	Data parallel	Task parallel
Template of Programming	None	Easy to integrate with sequential codes
Ease of programming	Difficult to implement parallelism	Easy to implement from existing sequential codes
Load balancing	Done by programmers	Integrated self-schedulers
Communication	Efficient	Depend on the environment
Deployment	Unnecessary	Necessary
Security	Not a problem in clusters	Can be implemented
Resources	Intended for networks	Intended for networks
Dynamic Tuning	No	Has three types of dynamic tuning
Performance	Not good at a large scale system	Good at all scales

4.2 Programming Styles

In the parallel model of hybrid MPI-OpenMP, the number of OpenMP threads is equal to the number of CPUs in a computer node and there is one MPI process in the computation node. The code style of MPI is single program multiple data (SPMD). In OpenMP, each thread worker takes one iteration in the **omp** for-loop (omp is an OpenMP compiler directive). Listing 4.1 is the pseudocode of Hybrid MPI-OpenMP.

Similarly, in the parallel model of hybrid MPI-Pthreads, the number of threads is equal to the number of CPUs in a computer node and there is one MPI process in the computation node. In the Pthreads structure, each thread worker takes a task region. Listing 4.2 is the parallel pseudocode of Hybrid MPI-Pthreads.

Listing 4.1: The standard code of hybrid MPI-OpenMP programming

```
number_of_tasks_execute_in_each_node = number_of_tasks dividend number_of_computation_processes
```

```
Send arrange_of_task (from, to) based on number_of_tasks_execute_in_each_node to each process
// In each process
#pragma omp parallel for private (...)
#pragma omp for
    for walker w with an arrange_of_task do
    // the code
    end for
```

Listing 4.2: The standard code of hybrid MPI-Pthreads programming

```
tasks_execute_in_each_node = number_of_tasks dividend number_of_computation_processes
Send arrange_of_task (from, to) based on number_of_tasks_execute_in_each_node to each process
// In each process
number_task_execute_in_each_thread = number_of_tasks dividend number_of_worker_threads
each p_thread_worker execute an arrange of tasks (to, from) based on tasks_execute_in_each_node.
```

In the parallel model of each node of CTunes, a whole computation is divided into subtasks. Each subtask is executed by one of the computation nodes. A subtask is divided into a list of tasks, and each computation node creates many thread workers to execute the list of tasks. The coding style of processing the list is an SPMD style. Each worker thread in each node executes a fraction of (the number of tasks divided by the number of threads in this node) the tasks. The listing 4.3 shows the pseudocode of CTunes.

Listing 4.3: The parallel pseudocode in CTunes

```
while i_subtask is available do
    i_execute_node = i_subtask modulo number_of_nodes
    send i_subtask to the the node with the node of i_execute_node id.
end for
// In each computation node
```

number_of_tasks = parse(i_subtask)

task_execute_in_each_thread = number_of_tasks div number_of_worker_threads

each thread_worker executes a range of tasks (to, from) based on task_execute_in_each_thread.

4.3 Code Complexity

In CTunes, we define two aspects: model and control. In the model aspect, we define a task model, and a resultant model. In the control aspect, we define a master class which extends the **MasterNode** class, and a computation class which extends the **ComputationNode** class. Here we use a matrix multiplication application to demonstrate how to program in CTunes system.

4.3.1 Model

A matrix multiplication consists of a matrix range task class of RangeTask and matrix result class of Result.

Listing 4.4: The models

```
public class RangeTask implements Serializable {
        private int fromTask;
        private int toTask;
        public void setFromTask(int fromTask) {
                this.fromTask = fromTask;
        }

        public void setToTask(int toTask) {
                this.toTask = toTask;
```

```
        }

        public int getFromTask() {
                return fromTask;
        }

        public int getToTask() {
                return toTask;
        }
}
public class Result implements Serializable{
        private int row;
        private double[] cols;
        public void setRow(int row) {
                this.row = row;
        }
        public int getRow() {
                return row;
        }
        public double[] getCols() {
                return cols;
        }
        public void setCols(double[] cols) {
                this.cols = cols;
        }
}
```

4.3.2 MasterNode

This class is responsible for creating a list of subsets of tasks. This class extends the **MasterNode** and executes in a master node. There are 2 generic data types: a subset task model class, and a result model class.

Listing 4.5: The matrix multiplication of MasterNode

```
public class MatrixMasterNode extends MasterNode<RangeTask, Result>{

    private int numberOfMatrixEachTask = 40;
    private int matrixSize;
    private double[][] matrixA = null;
    private double[][] matrixB = null;
    private double[][] matrixC = null;
    private int taskId;

    public static final String INPUTFILE_MATRIXA = "inputFile1";
    public static final String INPUTFILE_MATRIXB = "inputFile2";

    public void setNumberOfMatrixEachTask(int numberOfMatrixEachTask) {
        this.numberOfMatrixEachTask = numberOfMatrixEachTask;
    }
    public void setMatrixSize(int matrixSize) {
        this.matrixSize = matrixSize;
    }
    // setup data and send data to each computation node
    protected void startup() throws Exception {
```

```
    taskId = 0;
    matrixA = MatrixMain.random(matrixSize);
    matrixB = MatrixMain.random(matrixSize);

    broadcastDataInChanel(INPUTFILE_MATRIXA, matrixA, false);
    broadcastDataInChanel(INPUTFILE_MATRIXB, matrixB, false);
    waitToSendAllData();
}
// create a subtask for each node
protected RangeTask setupSubsetTaskForEachNode() {

        if (taskId * numberOfMatrixEachTask < matrixSize) {
                RangeTask task = new RangeTask();
                task.setFromTask(taskId * numberOfMatrixEachTask);
                int taskTo = (taskId + 1) * numberOfMatrixEachTask-1;
                taskTo = taskTo < matrixSize - 1 ? taskTo:matrixSize - 1;
                task.setToTask(taskTo);
                taskId++;
                return task;
        } else {
                return null;
        }
}
//collect the resultant
protected void cleanup() {
        matrixC = new double[matrixSize][matrixSize];
        Result[] results = getResults();
```

```
            for (Result matrix: results){
                matrixC[matrix.row] = matrix.cols;
            }
        }
}
```

4.3.3 ComputationNode

This class is responsible for receiving a subset of tasks, splitting it into a list of tasks, and executing the list of tasks in parallel. This class extends the **ComputationNode** . This class executes on all computation nodes. There are 3 generic data types: the subset task model class, the task model class, and result model class.

Listing 4.6: The matrix multiplication of ComputationNode

```
public class MatrixComputationNode extends ComputationNode<RangeTask,Integer,Result>{

    private double[][] matrixA = null;
    private double[][] matrixB = null;
    protected void receiveArrayData(String nameCache) {

        if (nameCache.equals(MatrixMasterNode.INPUTFILE_MATRIXA)) {
            matrixA = arrayData(nameCache, double[].class);
        } else if (nameCache.equals(MatrixMasterNode.INPUTFILE_MATRIXB)){
            matrixB = arrayData(nameCache, double[].class);
        }
    }
    protected void receiveBroadcastData(String nameCache, Object object) {
        if (nameCache.equals(MatrixMasterNode.INPUTFILE_MATRIXA)) {
```

```
                matrixA = (double[][]) object;
        } else if (nameCache.equals(MatrixMasterNode.INPUTFILE_MATRIXB)){
                matrixB = (double[][]) object;
        }
}
protected Integer[] splitSubsetTaskToTaskList(RangeTask task) {
        Integer[] list = new Integer[task.getToTask()-task.getFromTask()+1];
        for(int i = task.getFromTask(); i <= task.getToTask(); i++){
                list[i-task.getFromTask()] = new Integer(i);
        }
        return list;
}
protected Result processTaskInParallel(Integer i) {
        Result mresult = new Result();
        try {
                double[][] A = matrixA;
                double[][] B = matrixB;
                int len = A[i].length;

                double[] cols = new double[len];
                for (int j = 0; j < len; j++) {
                        double result = 0;
                        for (int k = 0; k < len; k++) {
                    result = result + (A[i][k] * B[k][j]);
                }
                        cols[j] = result;
                }
```

```
                mresult.setRow(i);
                mresult.setCols(cols);
            } catch (Exception e) {
                throw new RuntimeException(e);
            }
            return mresult;
        }
}
```

The following is a main class to run MatrixMasterNode class.

Listing 4.7: The main class

```
public class MatrixMain {
    public static void main(String[] args) throws IOException {
        MatrixMasterNode task = new MatrixMasterNode();
        if (args.length>1){
            task.setMatrixSize(Integer.parseInt(args[0]));
            task.setNumberOfMatrixEachTask(Integer.parseInt(args[1]));
        }
        task.setObservationControlClass(HybridControl.class);
        task.setNodeTask(MatrixComputationNode.class);
        task.setModelClass(Result.class);
        task.setSubtaskClass(RangeTask.class);
        task.process();
        task.close();
    }
    public static double[][] random(int n) {
        double[][] rand = new double[n][n];
```

Table 4.2: MPI and Hybrid vs. CTunes

	MPI	**Hybrid MPI+OpenMP**	**Hybrid MPI+Pthread**	**CTunes**
NBody	240 lines + NBody structute: 14 lines	256 lines + NBody structute: 14 lines	303 lines + NBody structute: 14 lines	4 files including NBody: 30 lines, NBodyComputationNode: 63 lines, NBodyMain: 35 lines, NBodyMasterNode: 56 lines, NBodyRangeTask: 20 lines. 204 lines in total.
Matrix Multiplication	147 lines	167 lines	215 lines	4 files including MatrixMain: 25 lines, MatrixComputationNode: 44 lines, MatrixMasterNode: 50 lines, RangeTask: 15 lines, Result: 15 lines. 149 lines in total.

```
            for (int i = 0; i < n; i++)
                for (int j = 0; j < n; j++)
                    rand[i][j] = Math.random();
        return rand;
    }
}
```

Table 4.2 compares the the length of the code for each model in NBody and Matrix Multiplication. CTunes has 4 files, making it readable and maintainable. MPI, hybrid MPI-OpenMP, hybrid MPI-Pthreads are less readable, and hard to maintain.

Chapter 5

Performance Evaluation

We illustrate the effectiveness of our approach using three case studies, each of which represents a class of computations:

- PI calculation is to calculate the number of PI.
- Matrix multiplication is to calculate the multiplication of two n $\times$ n matrices.
- The Gravitational N-Body Problem (GNBP) [45] is a traditional physics problem which aims to predict the motion of a group of celestial objects which exert a gravitational pull on each other. The objective is to find the positions and movements of the bodies in space, i.e. their subsequent motions given the initial positions, masses, and velocities that are subject to gravitational forces from other bodies as identified by Newtonian laws of physics .

We chose PI calculation because it represents a small computation and a strong synchronization (a mutex lock). We chose matrix multiplication as our second case study because it represents computations with large amount of data (matrix A and matrix B). Also, the matrix multiplication computation shows low spatial locality. Finally, we chose the N-Body problem as our third case study because it represents computations which consist of a series

of parallel computations connected by barriers, and the size of the input is medium. Also, an N-Body problem shows good spatial locality.

We used two types of hardware in these experiments, a network of 5 Virtual Machines (VMs) representing a small scale distributed system and a cluster representing 5 nodes for a large scale system. In the small scale system, VM is set to be the master node, and the rest of VMs are computation nodes. Each VM has an Intel CPU (4 cores) @ 3.2GHz, and an 8GB RAM. In the cluster, one node is set be a master node, and the rest of nodes are computation nodes. Each node of the cluster has dual 6-core Intel Xeon CPUs (12 cores in total), and a 32 GB RAM. The cluster nodes are collected by an infinitive-band network.

5.1 Case Study I: PI Computation

5.1.1 Experiment Design

In the first case study, we use a PI Computation to evaluate the effectiveness of CTunes, MPI, and hybrid MPI-OpenMP in achieving high performance on both systems. For PI calculation, we use the well-know dartboard algorithm [46]. Specifically, we simulate the process of throwing a number of darts. The computation checks whether the x-value and y-value of the point is in the circle located at the center of square with a radius of 1. We also defined the number of rounds of calculation to complete in order to estimate PI more accurately.

We developed three approaches for programming in PI computation in CTunes: CTunes - Global Variable, CTunes - Local Variable, and CTunes - Sequential. CTunes - Global Variable uses a shared variable to count the number of random points in multiple threads in each node. CTunes - Local Variable uses a local variable to count in each thread, and at the end of computation, all local variables are added together. In CTunes - Sequential, the

master node sends the number of random points to each node. In each node, the computation runs sequentially to check how many random points are in the circle. Therefore, there is no in-node parallelism in this approach.

In the MPI model, each MPI process handles the number of darts divided by the number of processes, and determines whether each dart is in the circle. In hybrid MPI-OpenMP, there are two types of parallelism where each node of hybrid MPI-OpenMP handles the number of darts divided by the number of computation nodes, and OpenMP is employed to exploit parallelism within a computation node. There are two variants of parallelism in OpenMP for hybrid MPI-OpenMP are hybrid MPI-OpenMP Global Variable, and hybrid MPI-OpenMP Reduction. Hybrid MPI-OpenMP Global Variable uses a global variable to calculate the sum of the number of points across multiple threads, and uses a mutex lock synchronization to modify the global variable. Hybrid MPI-OpenMP Reduction uses a reduction clause, which is applicable to the whole parallel region in OpenMP, to calculate the sum the number of points across multiple threads.

Listing 5.1: The PI Computation

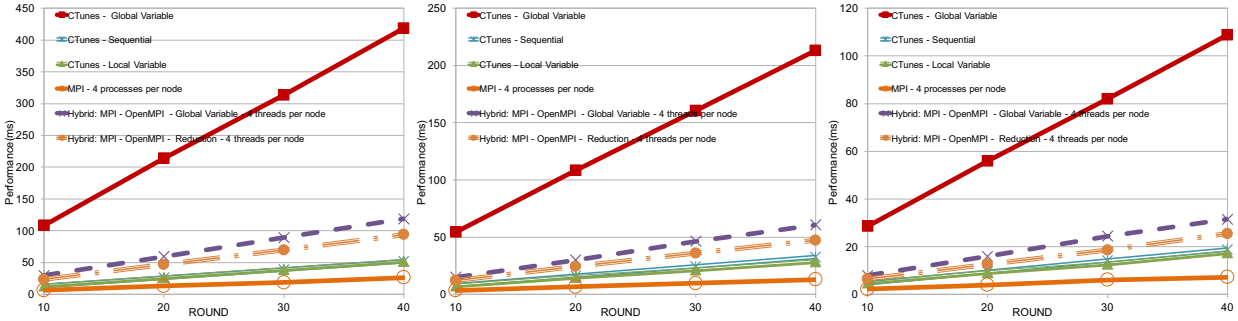
```
void main() {
    int DARTS = get from the data input // the number of random points
    int ROUND = get from the data input
    int numworkers; // number of process worker
    double avepi=0;

    for (i = 0; i < ROUNDS; i++) {
        if (process is a master process) {
            avereachprocess = DARTS/numworkers;
            extraprocess = DARTS%numworkers;
            for (dest=1; dest<=numworkers; dest++)
```

```
            {
                rows = (dest <= extra) ? averow+1 : averow;
                send rows to each process
            }
            totalscore = use MPI_Reduce to sum all homePI from each process
            calculate pi = 4.0 * totalscore/DARTS;
            avepi = ((avepi * i) + pi)/(i + 1);
        } else {
            rows = receive from a master process
            homePI = dboard(rows);
            send homePI to the master node
        }
    }
}
int dboard(int darts) {
    int score = 0;
    for (int i=0; i<darts ;i++) {
        r = random number from 0 to 1;
        x_coord = (2.0 * r) - 1.0;
        r = random number from 0 to 1;
        y_coord = (2.0 * r) - 1.0;
        if ((sqr(x_coord) + sqr(y_coord)) <= 1.0) { // check the random point
            score++;
        }
    }
    return score;
}
```

5.1.2 Experimental Results

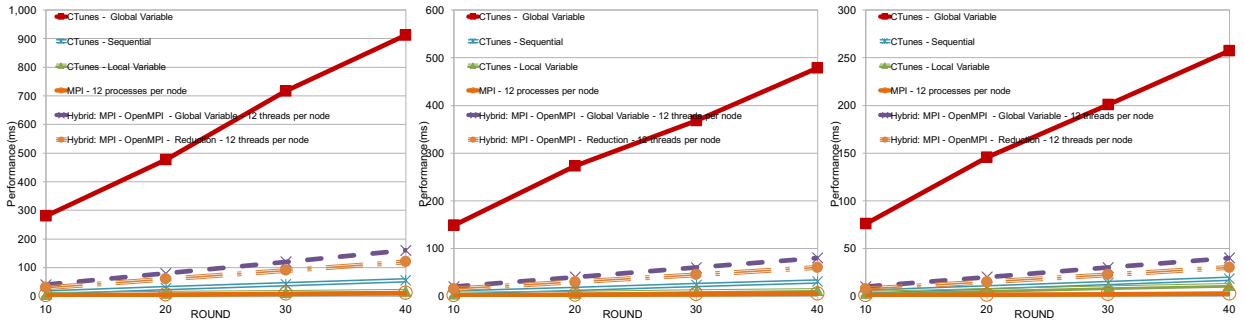
While CTunes - Global Variable and CTunes - Local Variable execute computations in parallel in each computation node, CTunes Sequential executes the computation sequentially in each computation node. Also, while MPI executes the computation in sequence in each process, hybrid MPI-OpenMP executes the computation in parallelism in each process.

Figure 5.1 and Figure 5.2 show that CTunes - Global Variable and hybrid MPI-OpenMP - Global Variable have the lowest performance because they use a mutex lock synchronization to calculate a shared variable across multiple threads. CTunes - Local Variable is better than hybrid MPI-OpenMP - Reduction and CTunes Sequential because it can effectively take advantage of the parallelism provided by the hardware since it doesn't use a strong mutex synchronization. MPI outperforms other approaches in this computation because it uses all cores in each node, and it does not have a shared variable. In addition, MPI only uses one level parallelism; hence, it has one synchronization and MPI_Reduce is an optimized implementation of synchronization. Among the approaches with two-level parallelism, CTunes - Local Variable is the best choice or the best performance because it can utilize a large number of cores by providing the dynamic tuning mechanisms.



(a) 1 node (b) 2 nodes (c) 4 nodes

Figure 5.1: Performance of PI computation on VMs. DARTS = 100,000,000



(a) 1 node (b) 2 nodes (c) 4 nodes

Figure 5.2: Performance of PI computation on cluster. DARTS = 100,000,000

5.2 Case Study II: Matrix Multiplication

5.2.1 Experiment Design

In the second case study, we investigate the performance of CTunes, MPI, and hybrid MPI-OpenMP using a matrix multiplication computation, in which we calculate the multiplication of two n × n matrices. This application is also computation intensive. To calculate an element in the result matrix C (C = A × B), a row from matrix A and a column from matrix B are required. To enable a fair comparison, the data of matrix A and matrix B are sent to all computation nodes in CTunes and hybrid MPI-OpenMP; in MPI, the data of matrix A and matrix B are also sent to all processes of computation nodes in the communication.

We have implemented 2 algorithms in the computation of matrix multiplication in the MPI programming model, MPI (Basic) and MPI (Optimized). Figure 5.3 shows the MPI (Basic). All computation processes create matrix A, matrix B, and matrix C. Then, all computation processes copy a part of matrix A and all of matrix B from the master process to compute. Figure 5.4 shows the MPI (Optimized). Matrix B is created by, and copied to, all computation processes. In addition, all the computation processes create a part of matrix A and a part of matrix C. This optimization helps reduce the overhead and improve the performance, with the cost of increased programming complexity.

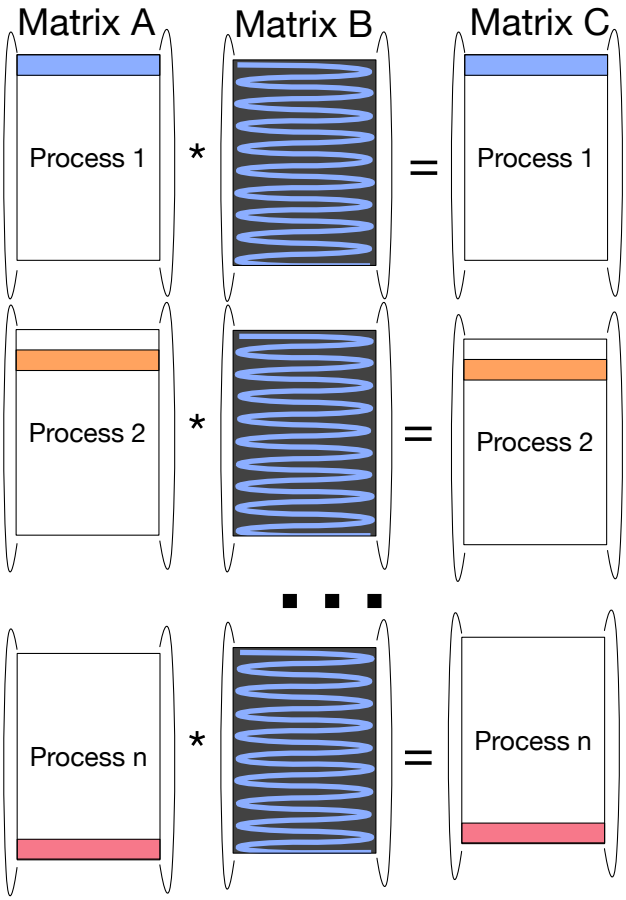


Figure 5.3: Matrix Multiplication with Parallel Algorithm in MPI (Basic)

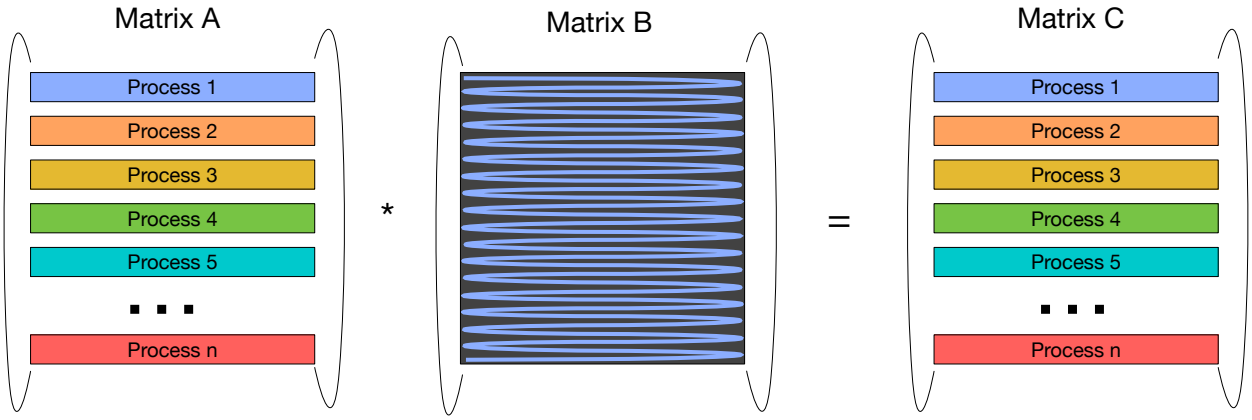


Figure 5.4: Matrix Multiplication with Parallel Algorithm in MPI (Optimized)

For each computation model, we measure the communication performance, overall performance (in both time and MFLOPS). Note that data type of the element of matrixes is double.

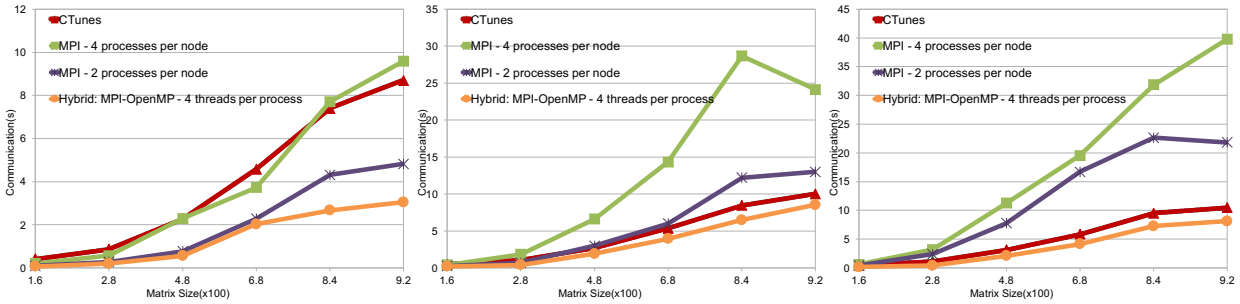
5.2.2 Experimental Results

Figure 5.5 shows the communication performance of two matrixes in a network of virtual machines. The hybrid MPI-OpenMP is the most efficient model in terms of communication. MPI shows the lowest broadcast performance, because there are 4 processes on each node, and the master process must send the matrix data to each of the 4 processes in each node. Recall that processes do not share data in MPI.

Figure 5.6 presents the overall performance of Matrix Multiplication in a network of virtual machines. CTunes outperforms all other models. The hybrid of MPI-OpenMP and MPI-Pthreads show similar performance. Both MPI programming models, MPI and MPI Optimize, have the lowest performance due to its heavy communication although matrix B, and a part of matrix A are sent to all computation processes.

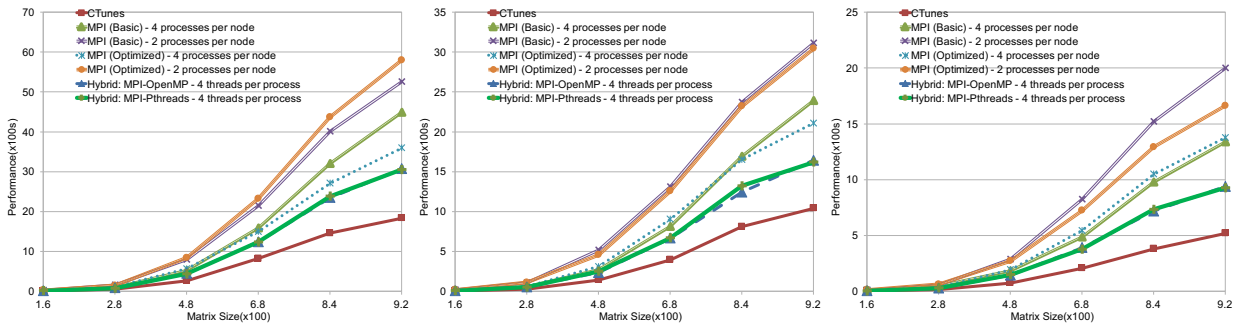
Figure 5.8 shows the communication performance of all models in the cluster. In this case, CTunes has the worst broadcast performance. The hybrid MPI-OpenMP has the best broadcast performance. In pure MPI, when the data is large, the bandwidth of the network is high, MPI program errors occur, giving no results. Let us consider an example of multiplying two 10000 x 10000 matrices. We use four cluster nodes with each running 12 processes. Since the data type of each matrix element is a double, the total size of the data passed to each node would be 2 (two matrixes) * 8 bytes (the size of double type) * 10,000 * 10,000 (the size of matrixes) = 1.6 GB. Since we need to send the same amount of data to 12 processes on four nodes, the total would become 76.8 GB. This exceeds the overhead capacity of the network, so MPI returns no results.

Figure 5.9 and Figure 5.10 present the overall performance of Matrix Multiplication in the cluster. the performance of MPI performance is reduced when the matrix size is large. CTunes has the best performance when the matrix is large, and it also shows better scalability then other approaches. Notably, CTunes shows a nearly linear speedup when the system scales up.



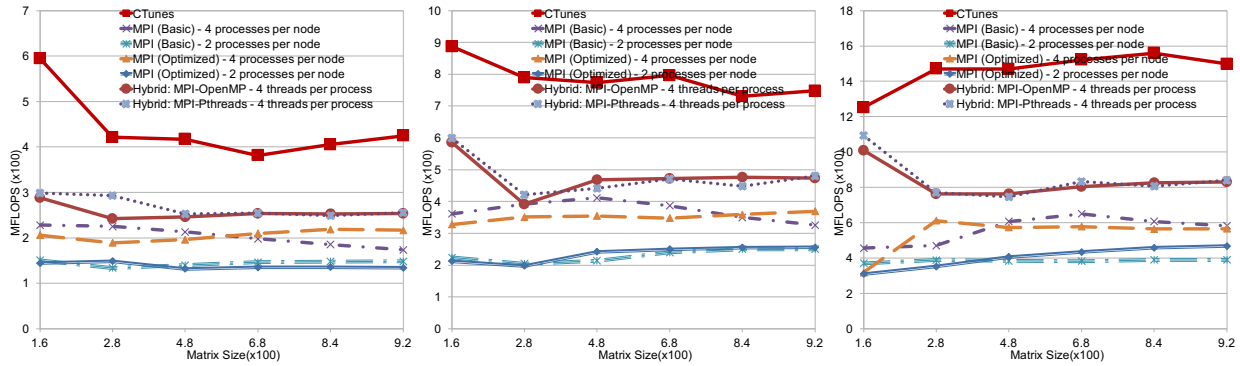
(a) 1 node (b) 2 nodes (c) 4 nodes

Figure 5.5: Broadcast Performance of Matrices on VMs



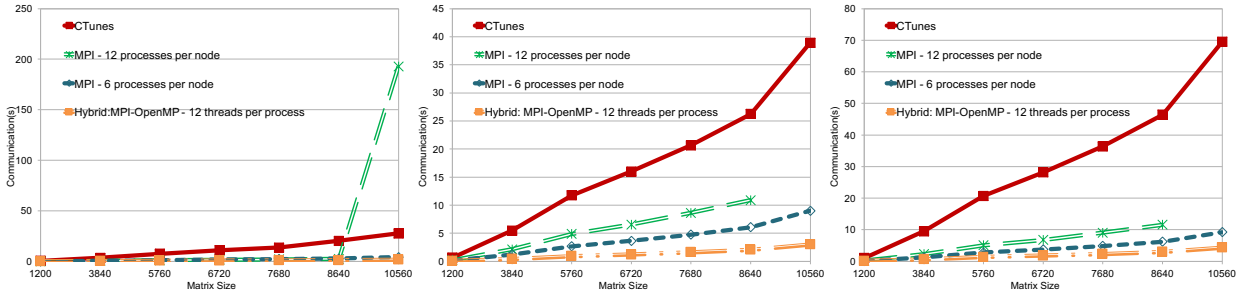
(a) 1 node (b) 2 nodes (c) 4 nodes

Figure 5.6: Performance of Matrix Multiplication on VMs



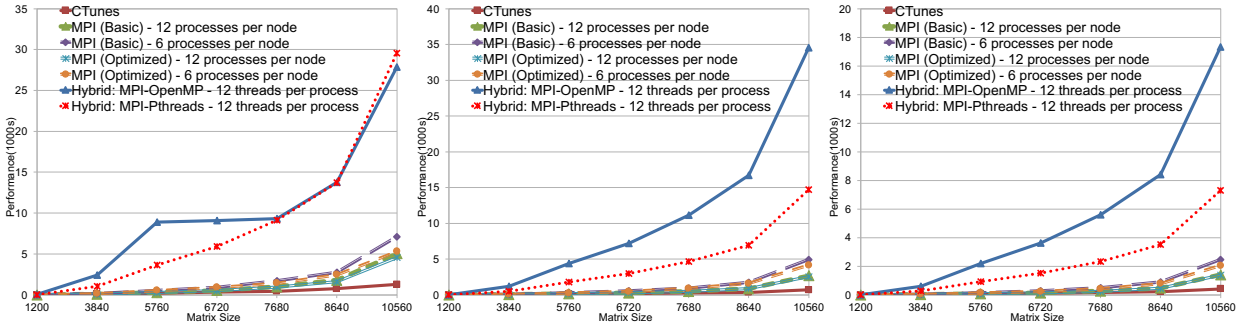
(a) 1 node (b) 2 nodes (c) 4 nodes

Figure 5.7: MFLOPS of Matrix Multiplication on VMs



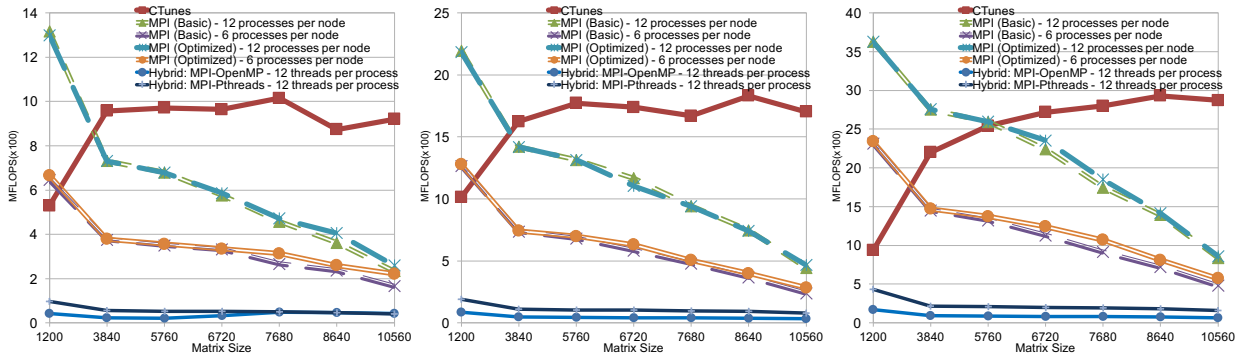
(a) 1 node (b) 2 nodes (c) 4 nodes

Figure 5.8: Broadcast Performance of Matrices on Cluster



(a) 1 node (b) 2 nodes (c) 4 nodes

Figure 5.9: Performance of Matrix Multiplication on Cluster



(a) 1 node (b) 2 nodes (c) 4 nodes

Figure 5.10: MFLOPS of Matrix Multiplication on Cluster

5.3 Case Study III: N-Body Problem

5.3.1 Experiment Design

For a fair comparison of different computing models, N-Body data is sent to all computation nodes in CTunes, hybrid MPI-OpenMP, and MPI.

For N-Body implementation in CTunes, the master creates tasks, and sends them to each computation. Each task contains a range of data, which includes data for multiple bodies. The computation node receives a task from the master node, and it processes the range of data for this task in parallel with the dynamic tuning mechanisms. The master node requests the result from each computation. The master node receives all the results of N-Body from a set of computation nodes continuously.

5.3.2 Experimental Results

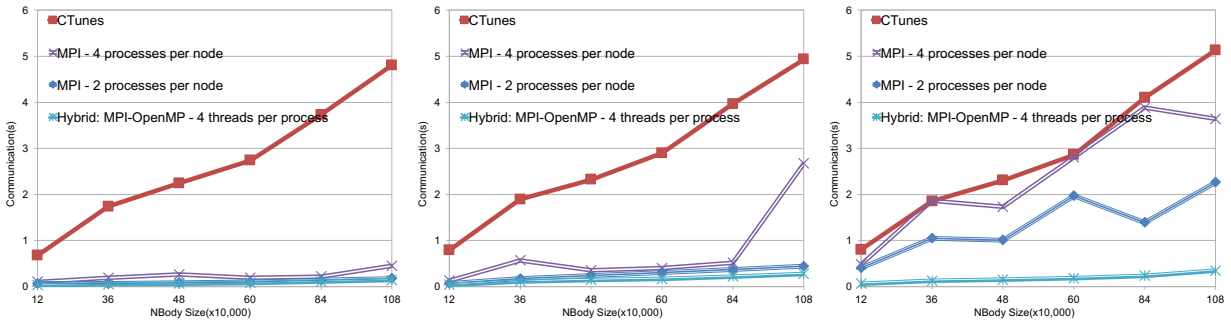
Figure 5.11 shows the communication performance of sending an array of N-Body data from the master node to a set of slave nodes in a network of VMs. The hybrid approach achieves the highest communication performance. CTunes is slower comparing to other models.

Figure 5.12 and Figure 5.13 shows the overall performance of the computation in a

network of VMs in terms of execution and MFLOPS respectively. CTunes achieves the best overall performance despite its low performance in communication. The hybrid of MPI-OpenMP and MPI-Pthreads show similar. The MPI with 2 processes per node has the lowest performance in this case.

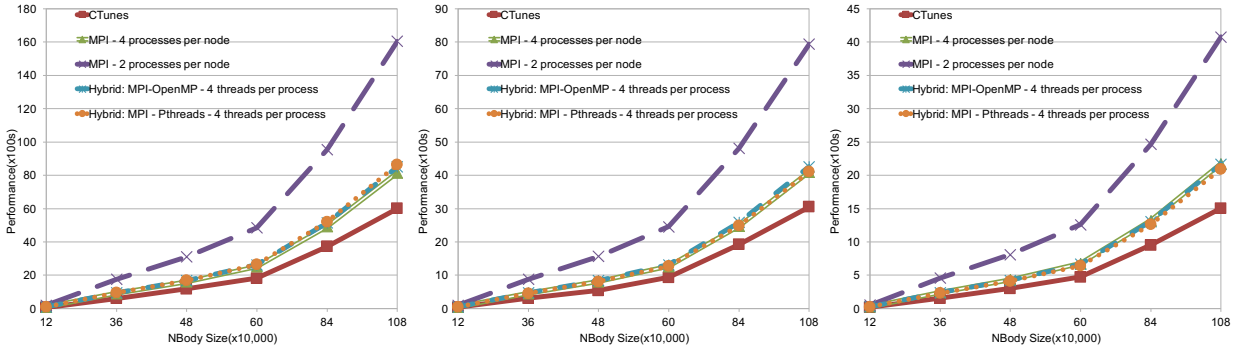
Figure 5.14 shows the communication performance of N-Body computation in a network of cluster. Similar to the case of VMs, CTunes has the lowest communication performance. The hybrid approach has the best communication performance.

Figure 5.15 and Figure 5.16 show the overall performance of the computation in a network of cluster. CTunes outperforms other approaches, and shows the best scalability.



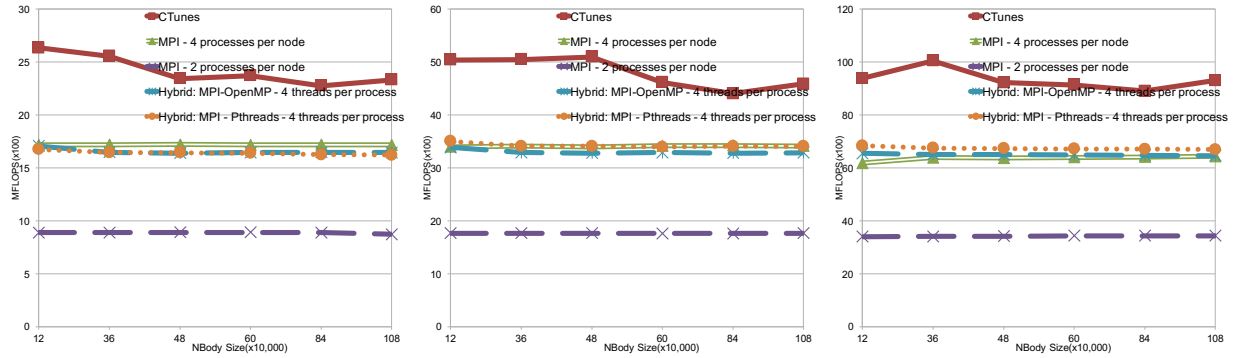
(a) 1 node (b) 2 nodes (c) 4 nodes

Figure 5.11: Broadcast Performance of N-Body on VMs



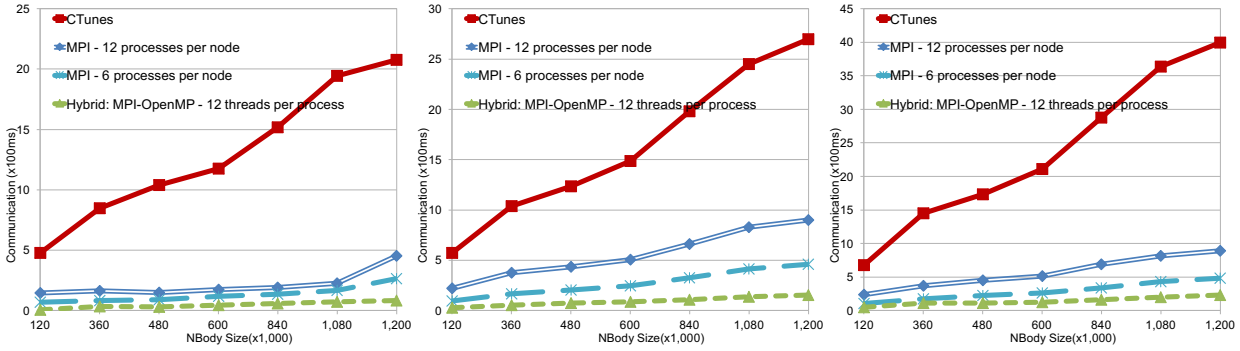
(a) 1 node (b) 2 nodes (c) 4 nodes

Figure 5.12: Performance of N-Body on VMs



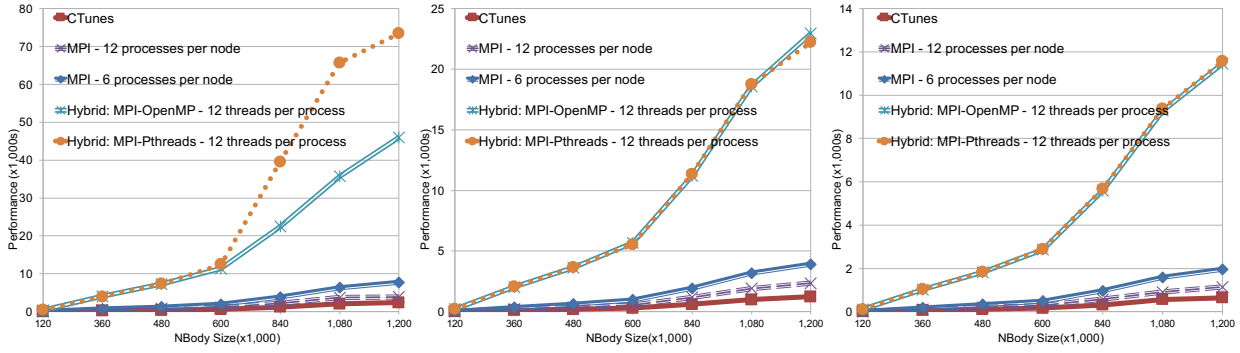
(a) 1 node (b) 2 nodes (c) 4 nodes

Figure 5.13: MFLOPS of N-Body on VMs



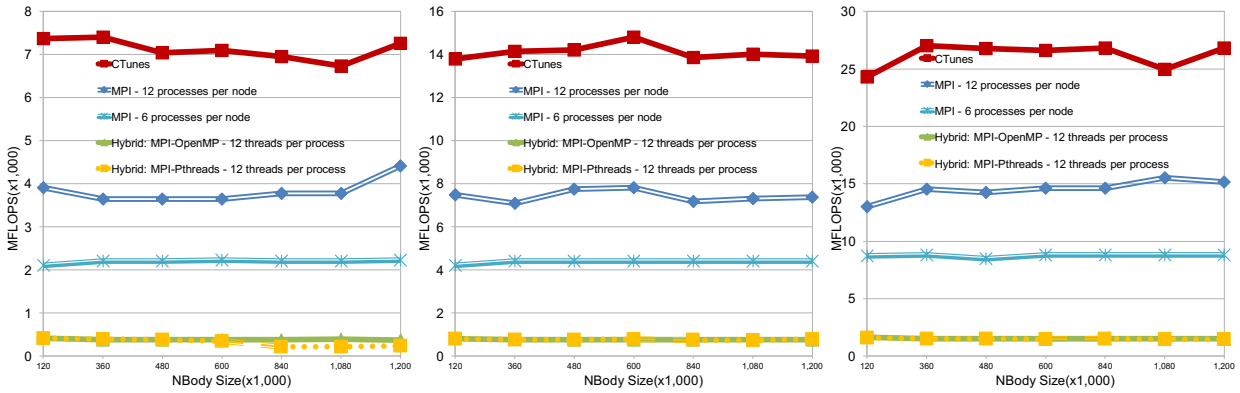
(a) 1 node (b) 2 nodes (c) 4 nodes

Figure 5.14: Broadcast Performance of N-Body on Cluster



(a) 1 node (b) 2 nodes (c) 4 nodes

Figure 5.15: Performance of N-Body on Cluster



(a) 1 node (b) 2 nodes (c) 4 nodes

Figure 5.16: MFLOPS of N-Body on Cluster

5.4 Discussions

In communication using CTunes, data is serialized to a binary data, and the binary data is sent to all the computation nodes through a socket channel. The communication performance of CTunes is worse than MPI and hybrid MPI-OpenMP. In addition, the hybrid MPI-OpenMP has only one process on each node resulting in less data being sent through the network.

The experimental results from the three case studies illustrate that with CTunes, we can bridge the gap between the hardware level and software level concurrency, and actively match these two for achieving better performance. Although CTunes communication performance is lower than MPI and hybrid MPI-OpenMP, its performance is better than MPI and hybrid MPI-OpenMP at both small and large scale systems. At a small scale, VMs are not powerful computers, and CTunes' performance is a little better than MPI, and hybrid MPI-OpenMP. At a large scale, where each node of the cluster is a powerful computer, CTunes' performance is twice as good as MPI, or hybrid MPI-OpenMP.

In the experiment of the communication of Multiplication Matrix in a cluster, MPI does not have a result if the matrix size is greater than 10,000. Since matrices A and B are sent

to all processes in nodes, the infinitive-band of bandwidth network reaches a limit. Our approach is not limited by network bandwidth because it is similar to hybrid MPI-OpenMP in communication, where data is sent once to each computation nodes.

In addition, CTunes implements a dynamic tuning of parallel tasks in each node. The dynamic tuning policy works at runtime. It dynamically adjusts the level of concurrency based on the progress of the users programs. The dynamic tuning is better suited for the situation where extensive synchronization is needed, which makes it difficult to estimate the optimal level of concurrency that the underlying hardware can support. Hence, our approach, the CTunes model, performs better than MPI and hybrid MPI-OpenMP at both scale systems because our approach can efficiently utilize the underlying hardware of each node, even though the performance of native language, C, is generally better the performance of Java language.

In conclusion, CTunes is not good in broadcasting data to each node. However, the mechanisms it provides are well suitable in exploiting the parallel hardware resources automatically, and demonstrate good scalability. In addition, this approach also simplifies the programmers' job of writing efficient parallel programs by providing APIs for communication and an easy-to-extend template for programming computations.

Chapter 6

Conclusion

We have presented our work to enhance the performance of parallel computing by effectively utilizing underlying hardware support of multiple cores and distributing tasks to each computation node through the network. Our work focuses on addressing two main challenges: 1) the separation of concurrency management and control in parallel programming from its functionality, network programming; and 2) the opportunities of achieving high performance on different hardware without needing to make changes to the users' programs.

In this thesis, we have described our approach to extending concurrency in parallel programming with dynamic tuning policies to improve the performance of each computation node. The dynamic tuning policies are separated from the user programming, and dynamically changes the thread-level concurrency to achieving high performance on different hardware without changing the program.

For inter-node communication, we have introduced a new communication model, FCM, which utilizes a server/client model. In FCM, we can send data, send a message, and call a function in each computation node. FCM is implemented by high performance socket programming in Java. Note that, the performance of sending data in FCM is worse than the performance of sending data in MPI and hybrid MPI-OpenMP.

Despite the slower communication, the new approach CTunes, a combination between FCM and Dynamic Tuning, outperforms MPI, hybrid MPI-OpenMP, and hybrid MPI-Pthreads in various case studies because the approach can effectively utilize the underlying hardware support of multiple cores in each computation node. The main disadvantage of the hybrid MPI-OpenMP approach is the relatively low performance, due to the limitations of OpenMP and Pthreads. The main disadvantage of the MPI approach is a problem in memory usage and overhead associated with MPI calls such as a limit of the memory and network bandwidth. Hence, MPI, hybrid MPI-OpenMP, and hybrid MPI-OpenMP models can not effectively utilize the parallelism supported by the underlying hardware of multiple cores in a large scale system. As our experiments demonstrate, CTunes effectively overcomes the disadvantages of MPI, hybrid MPI-OpenMP, and hybrid MPI-Pthreads approaches, but also to achieve higher performance than the these approaches.

Bibliography

[1] D. W. Walker, "The design of a standard message passing interface for distributed memory concurrent computers," *Parallel Computing*, vol. 20, no. 4, pp. 657–673, 1994.

[2] M. P. Forum, "Mpi: A message-passing interface standard," Knoxville, TN, USA, Tech. Rep., 1994.

[3] A. OpenMP, "Openmp application program interface, v. 3.0," *OpenMP Architecture Review Board*, 2008.

[4] H. Jin, D. Jespersen, P. Mehrotra, R. Biswas, L. Huang, and B. Chapman, "High performance computing using mpi and openmp on multi-core parallel systems," *Parallel Computing*, vol. 37, no. 9, pp. 562–575, 2011.

[5] R. Rabenseifner, G. Hager, G. Jost, and R. Keller, "Hybrid mpi and openmp parallel programming," in *PVM/MPI*, 2006, p. 11.

[6] L. Smith and M. Bull, "Development of mixed mode mpi/openmp applications," *Scientific Programming*, vol. 9, no. 2-3, pp. 83–98, 2001.

[7] F. Cappello and D. Etiemble, "Mpi versus mpi+ openmp on the ibm sp for the nas benchmarks," in *Supercomputing, ACM/IEEE 2000 Conference*. IEEE, 2000, pp. 12–12.

[8] A. Kneer, "Industrial mixed openmp/mpi cfd application for practical use in free-surface flow calculations," in *International Workshop on OpenMP Applications and Tools, WOMPAT 2000, http://www. cs. uh. edu/wompat2000/Program. html*, 2000.

[9] P. Kloos, F. Mathey, and P. Blaise, "Openmp and mpi programming with a cg algorithm," in *EWOMP 2000, European Workshop on OpenMP*, 2000.

[10] P. E. McKenney, "Is parallel programming hard, and, if so, what can you do about it?" *Linux Technology Center, IBM Beaverton*, 2011.

[11] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek *et al.*, "A view of the parallel computing landscape," *Communications of the ACM*, vol. 52, no. 10, pp. 56–67, 2009.

[12] H. Sutter and J. Larus, "Software and the concurrency revolution," *Queue*, vol. 3, no. 7, pp. 54–62, 2005.

[13] G. Blake, R. G. Dreslinski, T. Mudge, and K. Flautner, "Evolution of thread-level parallelism in desktop applications," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3. ACM, 2010, pp. 302–313.

[14] T. Nguyen and X. Zhao, "Self-adaptive parallel programming through tunable concurrency," in *Proceedings of the Companion Publication of the 2014 ACM SIGPLAN Conference on Systems, Programming, and Applications: Software for Humanity*, ser. SPLASH '14. New York, NY, USA: ACM, 2014, pp. 59–60. [Online]. Available: http://doi.acm.org/10.1145/2660252.2660394

[15] B. Nichols, D. Buttlar, and J. Farrell, *Pthreads programming: A POSIX standard for better multiprocessing.* " O'Reilly Media, Inc.", 1996.

[16] D. R. Butenhof, *Programming with POSIX threads.* Addison-Wesley Professional, 1997.

[17] W. Zhong, G. Altun, X. Tian, R. Harrison, P. C. Tai, and Y. Pan, "Parallel protein secondary structure prediction schemes using pthread and openmp over hyper-threading technology," *The Journal of Supercomputing*, vol. 41, no. 1, pp. 1–16, 2007.

[18] Threading Building Blocks. https://www.threadingbuildingblocks.org.

[19] K. H. Randall, "Cilk: E cient multithreaded computing," Ph.D. dissertation, Massachusetts Institute of Technology, 1998.

[20] S. Oaks and H. Wong, *Java threads.* " O'Reilly Media, Inc.", 2004.

[21] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: portable parallel programming with the message-passing interface.* MIT press, 1999, vol. 1.

[22] L. Hochstein and V. R. Basili, "The asc-alliance projects: A case study of large-scale parallel scientific code development," *Computer*, no. 3, pp. 50–58, 2008.

[23] L. Hochstein, F. Shull, and L. B. Reid, "The role of mpi in development time: a case study," in *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for.* IEEE, 2008, pp. 1–10.

[24] T. Hoefler, J. Dinan, D. Buntinas, P. Balaji, B. Barrett, R. Brightwell, W. Gropp, V. Kale, and R. Thakur, "Mpi+ mpi: a new hybrid approach to parallel programming with mpi plus shared memory," *Computing*, vol. 95, no. 12, pp. 1121–1136, 2013.

[25] R. Rabenseifner, G. Hager, and G. Jost, "Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes," in *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on.* IEEE, 2009, pp. 427–436.

[26] P. Haller and M. Odersky, "Actors that unify threads and events," in *Coordination Models and Languages*. Springer, 2007, pp. 171–190.

[27] M.-W. Jang, "The actor architecture manual," *Department of Computer Science, University of Illinois at Urbana-Champaign*, 2004.

[28] R. K. Karmani, A. Shali, and G. Agha, "Actor frameworks for the jvm platform: a comparative analysis," in *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*. ACM, 2009, pp. 11–20.

[29] J. Armstrong, *Programming Erlang*. Pragmatic Bookshelf, 2013.

[30] C. Varela and G. Agha, "Programming dynamically reconfigurable open systems with salsa," *ACM SIGPLAN Notices*, vol. 36, no. 12, pp. 20–34, 2001.

[31] The E language, 2000. http://www.erights.org/elang.

[32] Microsoft Corporation. Axum programming language, 2008.

[33] E. A. Lee and I. John, "Overview of the ptolemy project," 1999.

[34] D. G. Kafura and K. H. Lee, "Act++: building a concurrent c++ with actors," 1989.

[35] D. C. Sturman, G. Agha *et al.*, "A protocol description language for customizing failure semantics," in *Reliable Distributed Systems, 1994. Proceedings., 13th Symposium on*. IEEE, 1994, pp. 148–157.

[36] W. Kim, "Thal: An actor system for efficient and scalable concurrent computing," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1997.

[37] S. Srinivasan and A. Mycroft, "Kilim: Isolation-typed actors for java," in *ECOOP 2008–Object-Oriented Programming*. Springer, 2008, pp. 104–128.

[38] M. Astley, "The actor foundry: A java-based actor programming environment," *University of Illinois at Urbana-Champaign: Open Systems Laboratory*, 1998.

[39] J. Gosling, *The Java language specification*. Addison-Wesley Professional, 2000.

[40] G. L. Taboada, S. Ramos, R. R. Expósito, J. Touriño, and R. Doallo, "Java in the high performance computing arena: Research, practice and experience," *Science of Computer Programming*, vol. 78, no. 5, pp. 425–444, 2013.

[41] J. Waldo, "Remote procedure calls and java remote method invocation," *Concurrency, IEEE*, vol. 6, no. 3, pp. 5–7, 1998.

[42] G. Agha, "Concurrent object-oriented programming," *Communications of the ACM*, vol. 33, no. 9, pp. 125–141, 1990.

[43] Microsoft Corporation. Asynchronous agents library. https://msdn.microsoft.com/en-us/library/dd492627.

[44] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick, "A view of the parallel computing landscape," *Commun. ACM*, vol. 52, no. 10, pp. 56–67, Oct. 2009. [Online]. Available: http://doi.acm.org/10.1145/1562764.1562783

[45] J. Barnes and P. Hut, "A hierarchical o(n log n) force-calculation algorithm," *nature*, vol. 324, no. 6096, pp. 446–449, 1986.

[46] D. Patterson, "The trouble with multi-core," *Spectrum, IEEE*, vol. 47, no. 7, pp. 28–32, 2010.