MULTITHREADED PARALLEL IMPLEMENTATION OF HMMPFAM ON EARTH

by

Weirong Zhu

A thesis submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Master of Electrical Engineering

Spring 2004

MULTITHREADED PARALLEL IMPLEMENTATION OF HMMPFAM ON EARTH

by

Weirong Zhu

Approved: ______________________________

Guang R. Gao, Ph.D.
Professor in charge of thesis on behalf of the Advisory Committee

Approved: ______________________________

Gonzalo R. Arce, Ph.D.
Chair of the Department of Electrical and Computer Engineering

Approved: ______________________________

Eric W. Kaler, Ph.D.
Dean of the College of Engineering

Approved: ______________________________

Conrado M. Gempesaw II, Ph.D.
Vice Provost for Academic and International Programs

ACKNOWLEDGMENTS

Firstly, I would like to thank Dr. Guang R. Gao for his advisement and support during the last two years. His insight and methodology on the research problems always teach me a lot. I really appreciate his comments and valuable guidance provided in the development of my thesis.

I would also like to thank Dr. Kevin Theobold, who guided me to the area of parallel system and computer architecture and taught me lots of computing skills during past three years.

Also, I like to thank Yanwei Niu, Dr. Jizhu Lu, Chuan Shen, who worked with me for developing parallel system and parallel applications. One of important outcomes of our work – parallel HMMPFAM will be shown in this thesis.

In addition, I would like to acknowledge Dr. Ziang Hu, Dr. Clement Leung, Yuan Zhang for helping to proof read and comment on my thesis at various stages of development.

This research is funded in part by NSF, under the NGS grant 0103723, DOE, grant number DE-FC02-01ER25503. I also thank other US Federal agencies and the state of Delaware (via Delaware Biotechnology Institute) for their supports.

I would like to thank the Bioinformatics and Microbial Genomics Team at the Biochemical Sciences and Engineering Division within DuPont CR&D for the useful discussions during the course of this work.

I thank fully acknowledge the experiment platform support from the HPC Systems Group and the Laboratory Computing Resource Center at Argonne National Laboratory.

I would also like to acknowledge all of the members of the the CAPSL group, for sharing serious work and research for almost three years.

Finally, I wish to express my greatest sense of gratitude to my parents and sister in China, for their love and support all those years.

TABLE OF CONTENTS

LIST OF FIGURES . . . . **viii**
LIST OF TABLES . . . . **x**
ABSTRACT . . . . **xi**

Chapter

1 INTRODUCTION **1**

1.1 Goals and Synopsis 3

2 BACKGROUND **5**

2.1 profile HMMs, HMMER, PFAM and HMMPFAM 6

2.1.1 Profile Hidden Markov Models 6
2.1.2 HMMER 7
2.1.3 PFAM 9
2.1.4 HMMPFAM 9

2.2 EARTH Architecture and Execution Model 11

2.2.1 The EARTH Program Execution Model 12
2.2.2 The EARTH Architecture Model 13
2.2.3 The Threaded-C Language 14

2.3 Review of EARTH Runtime System 2.5 16

2.3.1 The Layout Design of RTS 2.0 17
2.3.2 Avoid Mutual Exclusions for Queues in RTS 18
2.3.3 New Layout Design of RTS 2.5 20
2.3.4 Locks in RTS 2.0 and RTS 2.5 22

2.3.5 EARTH Operations and Messages in RTS 2.5 23

2.3.5.1 EARTH Operations supported in RTS 2.5 23
2.3.5.2 New Message Formats 23
2.3.5.3 Receive Module 25
2.3.5.4 Execution Module 26
2.3.5.5 Send Module 27

2.3.6 Dynamic Load Balancers in RTS 2.5 28

3 CLUSTER-BASED SOLUTION ON EARTH 29

3.1 The Problem of PVM HMMPFAM Implementation in HMMER . . . 29
3.2 Embarrassingly Parallel Computing Strategy 31
3.3 A Two Level Master-Slave Parallel Scheme 32
3.4 Cluster-Based Solution 33

3.4.1 Static Load Balancing Approach 33
3.4.2 Robust Dynamic Load Balancing Approach 34

4 PORTABILITY OF EARTH PROGRAM 37

4.1 Code Portability of EARTH Program 38
4.2 Performance Portability of EARTH Program 41
4.3 Related Works 44

5 PARALLEL SMP SOLUTIONS ON SMP 49

5.1 Pthreads HMMPFAM Implementation 49
5.2 OpenMP HMMPFAM Implementation 52
5.3 EARTH HMMPFAM Implementation on SMP 55

6 EXPERIMENTS AND RESULTS 58

6.1 Synopsis 58
6.2 Summary of Main Observations 58
6.3 Experimental Strategy, Platforms and Data Sets 59

6.3.1 Experimental Strategy and Framework 59
6.3.2 Computational Platforms 60

6.3.3 The Data Sets Used in Experiments 61

6.4 Comparison With the PVM Implementation 61
6.5 Scalability on Supercomputing Clusters 63
6.6 Performance Analysis for The Cluster Based Solution 65
6.7 Robustness Experimental Results 69
6.8 Experiment Results on SMP Platform 70
6.9 Performance Portability of Hmmpfam Based on EARTH PXM 73
6.10 Conclusions Drawn from Experiments 75

7 CONCLUSIONS 77
BIBLIOGRAPHY 79

LIST OF FIGURES

2.1 HMMPFAM Algorithm 11

2.2 EARTH Architecture Model 14

2.3 Multi Hello Word 15

2.4 Layout of EARTH RTS 2.0 18

2.5 Layout of EARTH RTS 2.5 21

3.1 Parallel Scheme of PVM Version 30

3.2 Two Level Parallel Scheme 32

3.3 Static Load Balancing Scheme 34

3.4 Dynamic Load Balancing Scheme 35

5.1 Hmmpfam Parallel Scheme of Pthreads Version 50

5.2 OpenMP Hmmpfam Work-Sharing Construct: Parallel *for* 53

5.3 OpenMP Static Scheduler 53

5.4 OpenMP Dynamic Scheduler 54

5.5 Load Balancing for SMP Node in RTS 2.5 56

6.1 The Comparison of PVM Version and EARTH Version on COMET 62

6.2 The Comparison of PVM Version and EARTH Version on COMET 62

6.3 Static Load Balancing on Chiba City 63

6.4 Static Load Balancing on Chiba City 64

6.5 Dynamic Load Balancing on Chiba City 65

6.6 Dynamic Load Balancing on Chiba City 66

6.7 Dynamic Load Balancing on JAZZ 66

6.8 Performance Degradation Ratio under Disturbance to 1 CPU . . . 70

6.9 Performance Degradation Ratio under Disturbance to 2 CPUs on 1 node 71

6.10 Performance Degradation Ratio under Disturbance to 2 CPUs on 2 node 71

6.11 Performance Degradation Ratio under Disturbance to 4 CPUs on 2 node 72

6.12 Absolute Speedup of OpenMP HMMPFAM Implementations with Different Scheduler 72

6.13 Absolute Speedup of SMP based HMMPFAM Implementations with Different APIs 73

6.14 Performance Portability of HMMPFAM on EARTH for Different Platforms 74

6.15 Performance Portability of HMMPFAM on EARTH from Uni-processor Cluster to SMP Cluster 75

LIST OF TABLES

2.1 Resource Locking Summary for Running on Uni-Processor (One EM) 22

2.2 Resource Locking Summary for Running on SMP (Multiple EMs) . 23

2.3 EARTH Operations Supported by RTS 2.5 24

2.4 Message Formats Defined in RTS 2.5 25

ABSTRACT

Hmmpfam [1] is a widely used bioinformatics software for sequence classification provided by Sean Eddy's Lab at the Washington University in St.Louis. In real situations, this program may need a few months to finish processing large amounts of sequence data. Thus parallelization of the Hmmpfam is an urgent demand from bioinformatics researchers. HMMer 2.2g provides a parallel hmmpfam program based on PVM (Parallel Virtual Machine) [2]. However, this PVM version does not have good scalability and can not fully take advantage of the current advanced supercomputing clusters.

Using state-of-the-art multi-threading computing concept, we implement a new parallel version of hmmpfam on EARTH (Efficient Architecture for Running Threads), an event-driven fine-grain multithreaded programming execution model, which supports fine-grain, non-preemptive fibers. In its current implementations, the EARTH multithreaded execution model is built with off-the-shelf microprocessors in a distributed memory environment. The EARTH runtime system (version 2.5) performs fiber scheduling, inter-node communication, inter-fiber synchronization, global memory management, dynamic load balancing and SMP node support.

Compared with the original PVM implementation, our implementation based on EARTH shows notable improvements on absolute speed-up, better scalability and robust performance. Experiments on the Chiba City supercomputing clusters at Argonne National Laboratory (ANL) achieve an absolute speedup of 222.8 on 128 dual-CPU nodes for a representative data set, which means that the total execution time is reduced from 15.9 hours (sequential program) to only 4.3 minutes. Our implementation also demonstrates performance portability on different platforms, including uni-processor clusters, single SMP machine, and SMP clusters.

Chapter 1

INTRODUCTION

Over the past few years, building clustering servers for high performance computing is gaining more and more acceptance. Assembling large Beowulf clusters [3] is easier than ever and the performance is increasing dramatically. According to November 2003's statistics, there are 208 clusters in the top500 supercomputing machine list [4], while there were 93 clusters among the top500 one year ago. Cluster-based solutions for new driving applications need to be developed to take full advantage of the current computing resources. Highly demanded by academic research and drug discovery market, bioinformatics application will be an important driving force for the development and deployment of clustering servers and even the design of next generation computer architecture and programming models.

On the other hand, a large amount of DNA, RNA and protein sequences are discovered by different sequencing projects at an accelerating rate in the post genome era. It is very important to find out the structure and function of each sequence to discover useful information. Sequence database searching and family classification are common ways to analyze the function of the sequences. The family classification of sequences is of particular interest to drug discovery research. For example, if an unknown sequence is identified to belong to a certain protein family, then its structure and function can be inferred from the information of that family. Furthermore if this sequence is sampled from certain diseases X and belongs to a family F, then X can be treated using the combination of existing drugs for F [5].

Sequence classification is an important computation intensive applications which can exploit the computing resources of supercomputing clusters. A typical way to do sequence classification is to use consensus HMM model (Hidden Markov Model) [6] built from multiple alignment of protein families. HMM is a probabilistic model used very widely in speech recognition and other areas [7]. In recent years, HMM has been an important research topic in bioinformatics area. It is applied systematically to model, align, and analyze entire protein families and secondary structure of sequences. Unlike conventional pairwise comparisons, a consensus model can in principle exploit additional information such as the position and identity of residues that are more or less conserved throughout the family, as well as variable insertion and deletion probabilities [8].

HMM is used to solve three types of questions: 1) How to build an HMM to represent a family? 2) Does a sequence belong to a family? For a given sequence, what is the probability that this sequence has been produced by an HMM model? 3) Assuming a sequence comes from a family represented by an HMM, what can we say about the structure of the sequence? The problem solved in this thesis falls into the second category. Usually, for a given unknown sequence, it is necessary to do a database search against an HMM profile database which contains several thousands of families.

HMMER [1] is an implementation of profile Hidden Markov Models (profile HMMs) for sensitive database searches. A wide collection of protein domain models have been generated by using the HMMER package. These models have largely comprised the Pfam protein family database [9][10][11], which is widely used for protein domain detection and function prediction. Hmmpfam, one program in the HMMER 2.2g package, is a tool for searching a single sequence against an HMM database. In real situations, this program may take a few months to finish processing large amounts of sequence data. Thus efficient parallelization of the Hmmpfam is essential to bioinformatics research.

HMMER 2.2g provides a parallel hmmpfam program based on PVM (Parallel Virtual Machine) [2]. However, the PVM version does not have good scalability and can not fully take advantage of the current advanced supercomputing clusters. So a cluster-based solution for hmmpfam is an urgent demand, and it should be highly scalable and robust on hundreds to thousands of computing nodes. To meet such challenges, we exploit the power of a multithreaded architecture and program execution model, such as EARTH (Efficient Architecture for Running THreads) model [12][13], where parallelism can be efficiently exploited on top of a supercomputing cluster based on off-the-shelf microprocessors.

1.1 Goals and Synopsis

The purpose of this thesis is to present:

i. the first EARTH execution model based parallel implementation of bioinformatics sequence classification application;

ii. the first largely scalable robust parallel Hmmpfam implementation targeted to advanced supercomputing clusters, which also achieves good performance on SMP machines;

iii. a new efficient master-slave dynamic load balancer is implemented in EARTH Runtime System. This load balancer is targeted to parallel applications adopting master-slave model and shows more robust performance than static load balancer. In our parallel Hmmpfam implementation, tasks are coarse-grain, which guarantees that the overhead of master-slave load balancer does not affect the performance.

iv. the first documented analysis of the code portability and performance portability of EARTH program.

The rest of this thesis is organized as follows. Chapter 2 provides the background information necessary to understand the works in later chapters. The cluster-based multithreaded parallel implementation is described in Chapter 3. Chapter 4 presents the analysis of the code portability and performance portability of the EARTH programs. Chapter 5 presents and compares several SMP based parallel implementations of HMMPFAM. The results of the implementation are presented in Chapter 6, and conclusions in Chapter 7.

Chapter 2

BACKGROUND

"Bioinformatics is often defined as the application of computational techniques to understand and organize the information associated with biological macromolecules" [14]. Biological data are currently being produced world widely at an accelerating rate, bioinformatics techniques try to manage the huge amount of data, and extract useful information from those data. In order to aid the analysis of data, methods and tools are developed; at the same time resources, like large public databases (e.g., GenBank [15], Protein DataBank [16]), are also built. The methods, tools and resources are used to analyze the data and interpret the results in a biologically meaningful way. Certain methods for analyzing biological data (e.g., genetic, protein data) have been found to be extremely computationally intensive, providing motivation for the use of more powerful computers.

On the other hand, high-performance computing focuses on making use of both hardware and software techniques developed or being developed to build computer systems for performing computation intensive applications. The simultaneous use of large numbers of processors (e.g. parallel computing) is a general way to achieve high-performance computing. There are two main streams of building parallel computer: one is tightly-coupled shared-memory multiprocessors, the other is loosely-coupled clusters of PCs. Experience has shown a great deal of software support is necessary to support the development and tuning of applications on parallel architectures. The work presented by this thesis mainly demonstrates how to use one efficient parallel programming execution model to gain high-scalable-performance for one widely used bioinformatics tools.

The purpose of this thesis is to present how the high-performance computing technology is applied to an important bioinformatics application – HMMPFAM and gains impressive empirical results. The following sections will present the background knowledges about HMM, HMMPFAM, EARTH, and EARTH Runtime System necessary to understand the work presented in later chapters.

2.1 profile HMMs, HMMER, PFAM and HMMPFAM

Basic concepts of profile HMMs, HMMER, PFAM and HMMPFAM are introduced in this section.

2.1.1 Profile Hidden Markov Models

Hidden Markov Model (HMM) is a probabilistic and statistical model. For system which is identified as a Markov Process with unknown parameters, the HMM is used to extract information from the observable state to determine the hidden parameters or states. The states or parameters extracted by HMM can be further used to perform analysis on the system observed.

"A HMM makes two primary assumptions. The first assumption is that the observed data arise from a mixture of K probability distributions. The second assumption is that there is a discrete time Markov chain with K states, which generates the observed data by visiting the K distributions in Markov fashion." [17] The "hidden" of HMM means the hidden state of the system (i.e., sequence for this thesis) can not be directly observed, they can only be extracted from the observable states by using the probability model. HMM has been found to be very useful in a variety of areas, including sequential modeling problems. In 1994, Anders Krogh, David Haussler etc. at UC Santa Cruz introduced profile HMMs to computational biology [18], and it's soon found that HMM technology is well-suited to the “profile” methods for searching a protein sequence database using multiple sequence alignments, which improve both the sensitivity and selectivity of database searches.

The sequence profile method [19] [20] is a powerful tool to detect distant relationships between amino acid sequences. The profile methods use position-specific scores for either nucleic acids or proteins and position specific gap penalties for providing a generalized description of a protein motif, which then can be used for sequence alignments and database searches. A sequence profile is derived from a multiple sequence alignment.

Profile HMMs are statistical models of multiple sequence alignments. “Profile HMMs turn a multiple sequence alignment into a position-specific scoring system suitable for searching databases for remotely homologous sequences. Profile HMM analyses complement standard pairwise comparison methods for large scale sequence analysis.”[6].

Basicly, profile HMMs are used to solve three types of questions:

1) How to build a HMM to represent a family?

2) Does a sequence belong to a family? For a given sequence, what is the probability that this sequence has been produced by a HMM model?

3) Assuming a sequence comes from a family represented by a HMM, what can we say about the structure of the sequence?

The work presented by this thesis falls into the second category.

2.1.2 HMMER

HMMER[1] is a freely distributable implementation of profile HMM software for protein sequence analysis, developed by Prof. Eddy’s group in Washington University School of Medicine. The HMMER has several important applications:

- Once a multiple sequence alignment has been constructed on a protein sequence family, which has a number of key residues, the HMMER helps to find more members of that family in the sequence databases.

- Automated annotation of the domain structure of proteins. After a new sequence is found, and the analysis of BLAST[21] shows that it may be homologues to a known domain, HMMER can help to confirm that the new sequence is really an homologue of that known domain.

- Automated construction and maintenance of large multiple alignment databases like Pfam [9]. HMMER is able to organize sequences into evolutionarily related families. After building "seed" alignment of a relatively small number of representative sequences from a large sequence family, then HMMER can be used to search the database for homology, and automatically produce the high-quality alignment for the rest sequences of the family.

There are currently nine tools supported in HMMER 2 package:[1]

hmmbuild Build a model from multiple sequences alignment.

hmmalign Align any numbers of sequences to an existing model – an HMM profile.

hmmcalibrate Takes an HMM and empirically determines parameters that are used to make searches more sensitive, by calculating more accurate expectation value scores (E-values).

hmmconvert Convert a model file into different formats.

hmmemit Emit sequences probabilistically from a profile HMM.

hmmfetch Get a single model from an HMM database.

hmmindex Index an HMM database.

hmmsearch Search a sequence database for matches to an HMM.

[1] The descriptions of programs in HMMER package are got from The HMMER User's Guide

hmmpfam Search an HMM database for matches to a query sequence. The parallelization of hmmpfam is the main topic of this thesis.

2.1.3 PFAM

Pfam is a large collection of multiple sequence alignments and hidden Markov models covering many common protein families. The Pfam database [9] is jointly developed by several groups in UK, USA, Sweden and France and is available at the web site of the Washington University in St. Louis (http://pfam.wustl.edu/), Sanger Institute in UK (http://www.sanger.ac.uk/Software/Pfam/), Karolinska Institutet in Sweden (http://pfam.cgb.ki.se/Pfam/), and Institute National de la Recherche Agronomique in France (http://pfam.jouy.inra.fr/).

"Pfam is a database of multiple alignments of protein domains or conserved protein regions. The alignments represent some evolutionary conserved structure which has implications for the protein's function. Profile hidden Markov models (profile HMMs) built from the Pfam alignments can be very useful for automatically recognizing that a new protein belongs to an existing protein family, even if the homology is weak. Unlike standard pairwise alignment methods (e.g. BLAST, FASTA), Pfam HMMs deal sensibly with multi-domain proteins. Pfam is formed in two separate ways. Pfam-A are accurate human crafted multiple alignments, whereas Pfam-B is an automatic clustering of the rest of a non-redundant protein database derived from the PRODOM database." [22]

2.1.4 HMMPFAM

Hmmpfam reads a sequence file (we will refer a sequence file as a seqfile in this thesis) and compares each sequence within it, one at a time, against all the HMMs in hmm database, looking for significantly similar matches.

There is a output report for each sequence in seqfile. This report consists of three sections:

- a ranked list of the best scoring HMMs,

- a list of the best scoring domains in order of their occurrence in the sequence,

- alignments for all the best scoring domains.

A sequence score may be higher than a domain score for the same sequence if there is more than one domain in the sequence; the sequence score takes into account all the domains. All sequences scoring above the -E and -T cutoffs are shown in the first list, then every domain found in this list is shown in the second list of domain hits. If desired, E-value and bit score thresholds may also be applied to the domain list using the –domE and –domT options.[2]

Figure 2.1 shows the basic algorithm of HMMPFAM.

Suppose there are n HMMs in the profile database, and there are k sequences in the sequence file. The hmmpfam program reads one sequence each time, and matches that query sequence with the HMMs in the database one by one, and scores the domain according to its occurrence in the sequence, finally outputs a report for the query sequences as described above. Then hmmpfam program begin another iteration to work on the next sequence in the sequence file.

In real situations, this program may take a few months to finish processing large amounts of database and sequence data. Thus efficient parallelization of the Hmmpfam is an urgent request from bioinformatics research. In this thesis, an efficient parallel implementation based on EARTH Execution Model is discussed and presented.

In HMMer 2.2 package, a parallel hmmpfam program based on PVM [2] is provided. However, this PVM version does not achieve good scalability and can

[2] The description of output report of HMMPFAM is got from The HMMER User's Guide

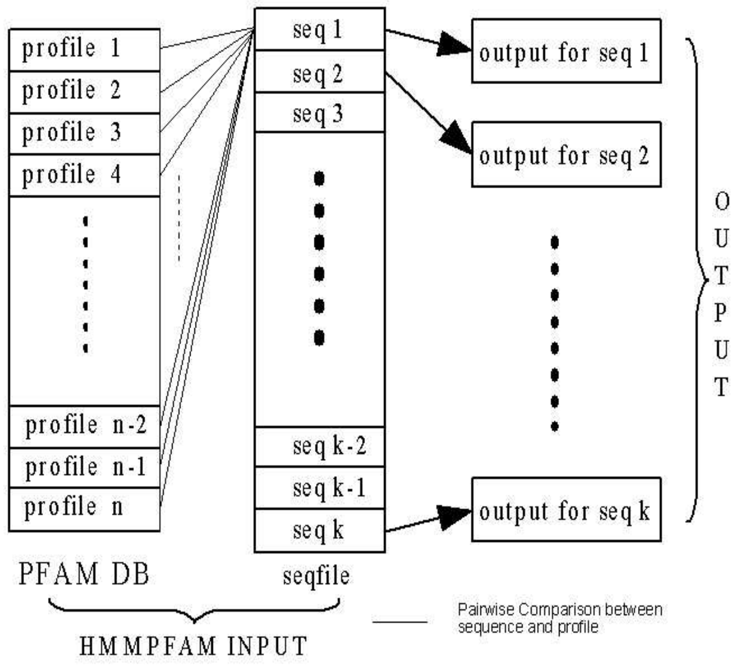

Figure 2.1: HMMPFAM Algorithm

not fully take advantage of the current advanced supercomputing clusters. Detail analysis will be given in next chapter.

2.2 EARTH Architecture and Execution Model

EARTH (*Efficient Architecture for Running THreads*)[12][13] is a multi-threaded architecture and execution model that supports fine-grain, non-preemptive threads. EARTH was firstly developed in ACAPS (Advanced Compilers, Architectures, and Parallel Systems) at McGill University, then was under development of CAPSL (Computer Architecture and Parallel Systems Laboratory) at University of Delaware. The EARTH model can be summarized and described from two levels: the *Program eXecution Model (PXM)* and the *Architecture Model*. In this section, the background knowledge of EARTH program execution model, the EARTH architecture Model and Threaded-C language will be given.

2.2.1 The EARTH Program Execution Model

The term *Program eXecution Model (PXM)* refers to the abstractions of "objects visible to the programmer, the operations which can be performed on these objects, and the general method for representing and executing computations on the machine."[12]

The PXM for EARTH has following important properties which differs EARTH from a conventional sequential computer:

- There can be multiple program counters, allowing concurrent execution of instructions from different parts of the program. While, there is only one single program counter in sequential program.
- Programs are divided into small sequences of instructions in a two-level hierarchy of threads.
- The execution ordering among threads is determined by data and control dependences explicitly identified in the program by a data-flow like firing rule.
- Frames holding local context for functions are allocated from a heap instead of a linear stack.

One of the most important features of EARTH PXM is its thread model. The EARTH PXM defines a two-level thread hierarchy: threaded procedures and fibers. In the EARTH model, a fiber is a *sequentially executed, non-preemptive, atomically-scheduled* set of instructions. And a fiber is always part of an enclosing threaded procedure, that means Threaded procedures are collections of fibers sharing the context of that procedure, including the local variables, input parameters of that procedure and sync slots. The context is stored in a frame, which is dynamically allocated from a heap. When a procedure is invoked, an appropriately-sized block of memory is allocated from the heap.

Programs structured using this two-level hierarchy can take advantage of both local synchronization and communication between fibers within the same thread, exploiting data locality. In addition, an effective overlapping of communication and computation is made possible by providing a pool of ready-to-run fibers from which the processor can fetch new work as soon as the current fiber ends and the necessary communication is initiated.

The EARTH PXM also defines a set of EARTH operations, which performs:

- Invocation and termination of procedures and fibers.
- Creation and manipulations of sync slots.
- Sending of sync signals to sync slots, either alone or atomically bound with data.

2.2.2 The EARTH Architecture Model

The EARTH Architecture Model is shown in Figure 2.2. An EARTH node is composed of an execution unit (EU), which runs the fiber, a synchronization unit (SU), which schedules the fibers when they are ready and handles the communication between nodes. There is also a ready queue (RQ) for storing ready fibers and an event queue (EQ) of EARTH operations generated by fibers running on EU. The local memory of an EARTH node is shared by the EU and SU. And there is an interface to the interconnection network.

The EU fetches individual fibers from the ready queue (RQ) and executes them as if they were normal sequential code. When the EU encounters EARTH operators, it forwards the operator to the SU by writing the request to event queue (EQ), where it stays until being read by SU. The SU processes each event it reads from the EQ. Some events will enable fibers. Enabled fibers are placed in the ready queue (RQ). Some events may request network communications, SU will interact

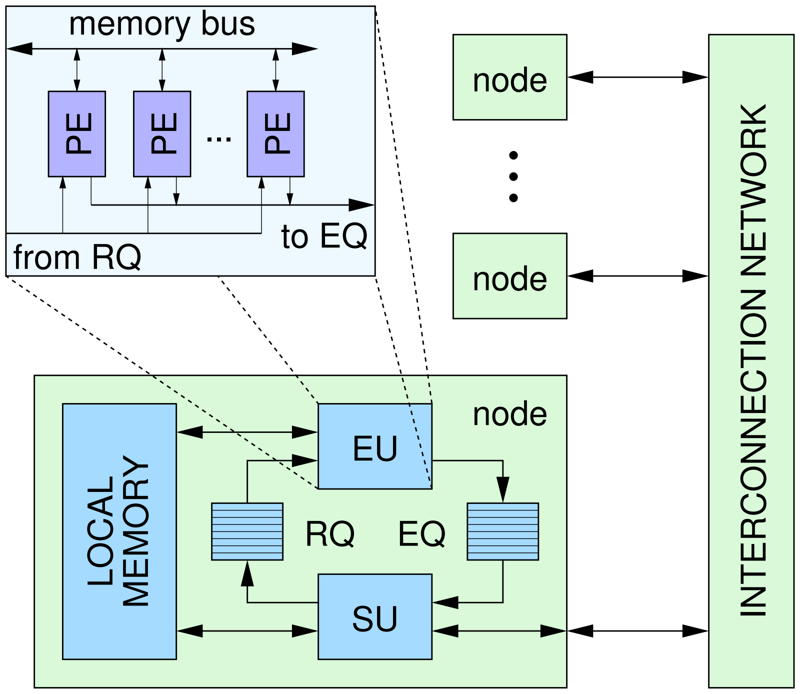

Figure 2.2: EARTH Architecture Model

with the interconnection networks. The SU is the unique and crucial component in EARTH architecture, it performs following functions:

- EU and network interfacing,
- Event Decoding,
- Sync Slot Management,
- Data Transfers,
- Fiber Scheduling,
- Procedure Invocation and Load Balancing.

2.2.3 The Threaded-C Language

Threaded-C [23][24] is a high level multi-threaded language, which supports the EARTH virtual machine. It's a ANSI standard C with extensions corresponding to the EARTH Operations defined in EARTH PXM. Threaded-C allow the programmer to indicate the parallelism explicitly. Till now, most applications for EARTH is written by Threaded-C. The latest version of Threaded-C is release 2.0 [24].

```
 1  #include <stdio.h>
 2
 3  #define REPEATS 1000
 4
 5  THREADED print_hello(SPTR done);
 6
 7  THREADED MAIN(int argc, char** argv)
 8  {
 9    int   i;
10    for (i = 0; i < NUM_NODES; ++i)
11      INVOKE(i, print_hello, TO_SPTR(DONE));
12    END_FIBER;
13
14    FIBER DONE <* NUM_NODES*REPEATS *> {
15      printf("Terminate normally.\n");
16      TERMINATE;
17    }
18  }
19
20  THREADED print_hello(SPTR done)
21  {
22    int i;
23    printf("node %d: Hello World!\n", NODE_ID);
24    for (i=0; i<REPEATS; i++)
25      SYNC(done);
26    TERMINATE;
27  }
```

Figure 2.3: Multi Hello Word

Figure 2.3 gives an example of Threaded-C code and the use of EARTH operations in a parallel version of famous “Hello World”. The MAIN function invoke an instance of the **print_hello** function, which prints a message identifying the location (node number) of each instance. Here is how this code works:

Lines 10-11: This loop is a simple example of a *forall* loop. In the loop, the MAIN function invokes an instance of the **print_hello** threaded procedure on each node.

Line 20: The **print_hello** procedure takes a reference to a sync slot as argument. SPTR is a pre-defined Threaded-C data type for a global pointer to a sync slot.

Lines 24-25: When **print_hello** finishes printing, it sends sync signals to the sync slot by REPEATS times. An SPTR is needed to refer to a sync slot.

Line 14: Fiber **DONE** plays the role of a *barrier*. It won't run until all nodes have finished their printing and sent out the REPEATS number of sync signals.

If line 11 in Figure 2.3 is replaced by

TOKEN(print_hello, SLOT_ADR(DONE));

the system will make its own decisions about where (which node) to run each instances, instead let programmer make the decision.

2.3 Review of EARTH Runtime System 2.5

In the implementations described in this thesis, the EARTH multithreaded architecture model is emulated with off-the-shelf microprocessors in a distributed memory environment. The EARTH runtime system (RTS) assumes the responsibility to provide an interface between an explicitly multithreaded program and a distributed memory hardware platform. The runtime system performs thread scheduling, context switching between threads, inter-node communication, inter-thread synchronization, global memory management, and dynamic load balancing.

The first portable EARTH RTS (release 1.0) was implemented and ported onto IBM SP-2, network of SUN workstations and Beowulf in 1998 [25]. Base on the RTS 1.0, the portable EARTH Runtime System 2.0 was developed [26]. The RTS 2.0 was written in standard C. Standard POSIX Threads was used for enabling multiple threads within a physical node, execution modules and network I/O modules were implemented by different pthreads. Standard Unix TCP/IP sockets was used for inter-node communication. The major features of RTS 2.0 was the support of SMP nodes, better overlapping of computation and communication, and its portability. A simple review of RTS 2.0 will be given in Section 2.3.1.

The development of RTS 2.5 began from April 2002 by Chuan Shen and the author of this thesis. The RTS 2.5 differs from RTS 2.0 on the local resource management, the way that inner node modules interact, network communication message formats and also improvements on the dynamic load balancer. The details of RTS 2.5's technical issues are documented in Chuan Shen's master thesis [27]. In this thesis, only a simple review of RTS 2.5 will be given here.

The development of RTS 2.5 is based on RTS 2.0. The network topology and network I/O models are inherited from the RTS 2.0 without modification. The same as RTS 2.0, RTS 2.5 still invokes n execution modules (the number of n depends on how many processors available), two network modules (send module and receive module) in a physical computing node. But the way that those modules interact and function is changed, which will be presented in section 2.3.3. Those changes make the requirements of mutual exclusions are considerably reduced, thus more efficient. The network messages is reformatted in the RTS 2.5, the new format contains only necessary information without redundancy and easier to parsed. And bugs and pitfalls in RTS 2.0 have been fixed, the performance of dynamic load balancer is improved.

2.3.1 The Layout Design of RTS 2.0

The design of RTS 2.0 is shown in the Figure 2.4. The RTS 2.0 is divided into three main threads. The thread of the RTS which is responsible for running Threaded-C application code is *Execution Module* (EM). The other two threads which perform all of the networking operations, are named as the *Sender Module* (SM) and the *Receive Module* (RM).

The three modules of the RTS are implemented as separate threads of execution by using POSIX threads (pthreads). The modules are created when the runtime system is launched, and they are terminated when the RTS has finished the execution of the Threaded-C application.

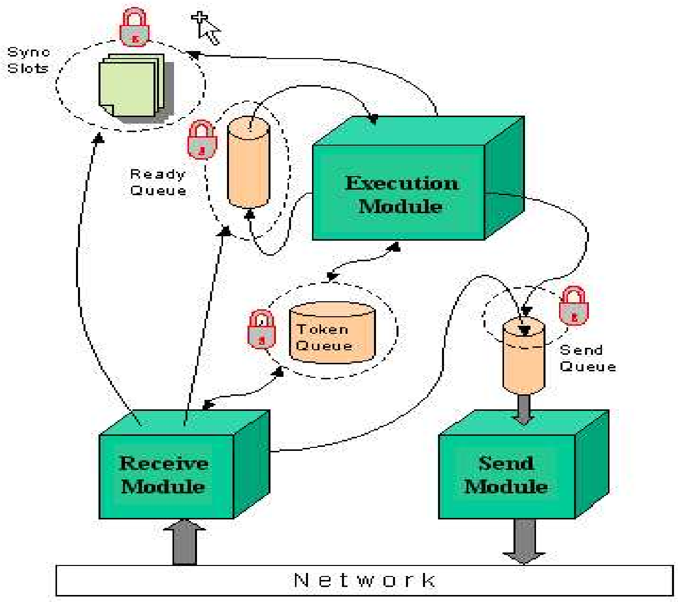

Figure 2.4: Layout of EARTH RTS 2.0

In the design of RTS 2.0, the EM is responsible for running the Threaded-C code; the SM is purely responsible for sending messages to the network; the RM receives incoming messages from the network, decodes the messages and perform the corresponding functionalities. The EM and RM communicates with the SM through the *Send Queue*; the *Ready Queue* stores ready fibers; and the *Token Queue* stores tokens.

The Figure 2.4 shows that there are four types of shared resources which can by accessed by different modules, thus locking mechanism is needed to guarantee the atomic and correct access. One the main target of RTS 2.5 is to avoid those locks to reduce management cost, thus to improvement the performance.

2.3.2 Avoid Mutual Exclusions for Queues in RTS

From the layout design of EARTH RTS 2.0, it is noticed that among the three modules there are four types of shared resources, three of which are queues:

Ready Queue, *Send Queue* and *Token Queue*. Queue is a first-in first-out data structure used to sequence multiple messages needed to be handled by one of the three modules. A queue can be regarded as an abstraction of producer-consumer problem [28]. Some modules work as producer to inject messages into the queue, while some other modules dequeue messages from the queue and handle those messages by specific handler. In case there are multiple writer and reader of a single queue, lock mechanism is necessary to guarantee the atomic access of the queue, thus to confirm the correctness of execution. In the implementation of EARTH RTS, the pthread mutex variable is used to protect access to shared queues. Mutex is an abbreviation for "mutual exclusion".

Experiments and analysis indicates that excessive instances of mutual exclusions is one major cause which prevents RTS 2.0 from achieving good performances [27]. One of main tasks of developing RTS 2.5 is to avoid those locks to reduce RTS management cost.

Locks are needed because there are multiple writers and readers for a shared queue. For example, in Figure 2.4, both the RM and EM talk to SM through the *Send Queue* in RTS 2.0.

If multiple modules communicates with one module through a single queue, to get rid of the locks, an intuitive method is to build multiple queues, which let each module has its own queue to send messages to the destination module. Consequently, each queue only has a unique writer and reader. Lamport proposed an algorithm for concurrent read/write of a fixed-size message buffer without incurring any mutual exclusions, when there are only one reader and one writer [29]. In EARTH RTS 2.5, the algorithm to perform concurrent read/write of a queue is derived from Lamport's.

From above analysis, in order to avoid the locks of *Send Queue*, separate send queues can be constructed as the message buffers for RM and EM to communicate with SM. When one module, for instance, the SM, behaves as reader for multiple queues, arbitration policies needed to serialize the reads.

However, when shared resources between two modules are through other shared data structures, for example the *Sync Slots* accessed by RM and EM, building multiple queues may not work in this case. The order of operations update the *Sync Slots* is nondeterminate. In RTS 2.5, mutual exclusions are used to serialize those updates. However, if there is a way to queue those update operations somewhere, and only allow one thread to perform operations in the queue, the mutual exclusions can be avoided, meanwhile serialized updates still performed.

In the design of EARTH RTS, all EARTH operations are performed by specific handlers, so only those specific handlers are possible to access the *Sync Slots*. In RTS 2.0, both the RM and EM have the right to call those handlers, thus mutual exclusions are needed. And when the access of one *Sync Slot* causes the count of *Sync Slot* reduced to zero, the handler will inject a new ready fiber into the *Ready Queue*. To avoid those mutual exclusions of *Ready Queue*, one solution is to only permit EM to call the handlers, which handles the *Sync Slots*, and RM's request for those handlers on *Sync Slots* are all queued in the new built *Event Queue*, which will only be polled by EM.

2.3.3 New Layout Design of RTS 2.5

The analysis in Section 2.3.2 leads the new design of EARTH Runtime System 2.5 demonstrated by Figure 2.5. The RTS 2.5 still inherits the three basic modules from RTS 2.0, the major differences are the way that the three Pthreads interact and function.

Firstly, in RTS 2.5, the *Receive Module* does not operate on *Sync Slots* and access the *Ready Queue* (RQ) and the *Token Queue* (TQ) any longer. The *Event Queue* (EQ) is built for RM to communicate with the EM, and the *Remote Send Queue* (RSQ) is used for RM to communicates with the SM. The RM is the only

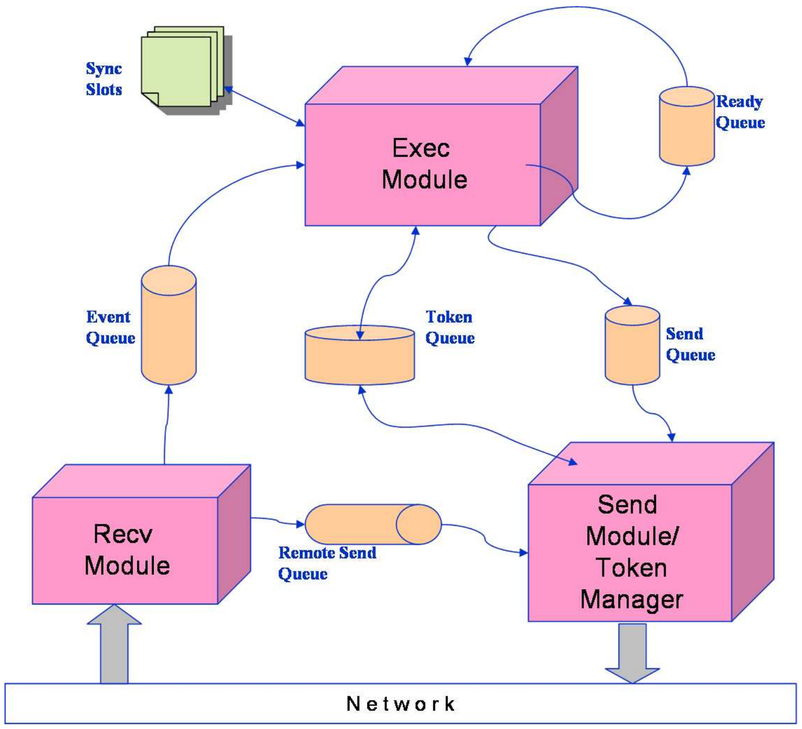

Figure 2.5: Layout of EARTH RTS 2.5

writer of those two queues, meanwhile, the EM is the only reader of EQ, and SM is the only reader of RSQ. The detail for how those modules work on those queue will be given in Section 2.3.5.

Secondly, A new module, *Token Manage* (TM) is introduced to the RTS 2.5 and dedicated to the dynamic load balancing. The TM resides in the same Pthread as the *Send Module* (SM), no separate Pthread is created for it. The TM handles TOKEN and TOKEN request messages from the *Send Queue* (SQ) and *Remote Send Queue* (RSQ), and reacts according to current load balance status, puts in or gets out TOKENs to or from *Token Queue* (TQ).

Thirdly, The *Token Queue* (TQ) is designed as a doubly-ended queue, accessible for both EM and TM. However, the EM only accesses TQ from one end, and the TM is only allowed to access TQ from the other end, so there is no lock for enqueue or dequeue operations except for handling the last element in the TQ. If

Resource	RTS 2.0			RTS 2.5		
	Reader	Writer	Lock	Reader	Writer	Lock
Sync Slots	EM, RM	EM, RM	Yes	EM	EM	No
Ready Queue	EM, RM	EM, RM	Yes	EM	EM	No
Event Queue	N/A	N/A	N/A	EM	RM	No
Send Queue	SM	EM, RM	Yes	SM	EM	No
Remote Send Queue	N/A	N/A	N/A	TM	RM	No
Token Queue	EM, RM	EM, RM	Yes	EM, TM	EM, TM	Last Item

Table 2.1: Resource Locking Summary for Running on Uni-Processor (One EM)

RTS is running on SMP computing nodes, more than one EM will be created in a node, each EM has its own TQ, and the *Token Manager* is capable of accessing all the TQs on the same physical node from the other end.

At the last, if there are multiple EMs, each EM has its own *Send Queue* (SQ), so that EM is the only writer of its SQ. And the SM will be the only reader of all SQs. Each EM also has its own *Ready Queue* (RQ), the EM is the only reader of its RQ, however, in some case, the EM may access other EM's RQ.

2.3.4 Locks in RTS 2.0 and RTS 2.5

One major goal of the new design of RTS 2.5 is to addressing the excessive locking problem in RTS 2.0. In the new design, the *Ready Queue* can be accessed only by the EM and is no longer a shared resource. If only one EM is initiated, then the *Sync Slots* is only accessed by the EM and is not a shared resource; if multiple EMs are initiated, then the *Sync Slots* are still needed to be locked by EM's access, but the RM and the SM will not lock them. The *Event Queue*, *Send Queue* and *Remote Send Queue* are all one-reader and one-writer queues, which is easily implemented as lock-free. The *Token Queue* is accessible to EM and TM from different ends, so locking is needed only for the last elements in the queue. Table 2.1 and Table 2.3 gave the summary of resource locking status of RTS 2.5 compared with RTS 2.0

Resource	RTS 2.0			RTS 2.5		
	Reader	Writer	Lock	Reader	Writer	Lock
Sync Slots	EM, RM	EM, RM	Yes	EMs	EMs	Yes
Ready Queue	EM, RM	EM, RM	Yes	EMs	EMs	Yes
Event Queue	N/A	N/A	N/A	EM	RM	No
Send Queue	SM	EM, RM	Yes	SM	EM	No
Remote Send Queue	N/A	N/A	N/A	TM	RM	No
Token Queue	EM, RM	EM, RM	Yes	EM, TM	EM, TM	Last Item

Table 2.2: Resource Locking Summary for Running on SMP (Multiple EMs)

2.3.5 EARTH Operations and Messages in RTS 2.5

In RTS 2.5, EARTH operations and messages are handled more intelligent than in RTS 2.0. Each Modules take its own responsibility to deal with corresponding EARTH operations.

2.3.5.1 EARTH Operations supported in RTS 2.5

The same as RTS 2.0, the RTS 2.5 mainly supports below EARTH Operations:

2.3.5.2 New Message Formats

In RTS 2.5, all network messages are redefined and reformatted. Message tags are used to identify the type of the message. The redundant fixed-size message header used in RTS 2.0 is discarded, only necessary information is packed in a message without any redundancy. The network message format defined in RTS 2.5 is summarized in Table 2.4. Those types of messages are not only used for network communication, but also used in the communication between different modules within one physical node.

Type	EARTH Operation	Function
Threaded Functions	INVOKE	Creates a new parallel invocations of threaded function on the specified node
	TOKEN	Creates a new parallel invocations of threaded function on a node selected by RTS
Fiber Synchronization	INIT_SLOT	Initializes sync slot
	SYNC	Decreases the sync count of sync slot by one, If the count reaches zero the associated fiber is scheduled for execution
	INCR_SLOT	Increases the sync count of sync slot by the value specified.
	SPAWN	Schedules the specified local fiber for execution
Data Transfer	PUT_SYNC	Sends a value to the destination address and update the specified sync slot
	GET_SYNC	Reads a value from the source address and copies it to the destination address. Then updates the specified sync slot
	BLKMOV_SYNC	Copies specified length of bytes of data from the source address to the destination address and then signals the specified sync slot when the data has reached its final destination

Table 2.3: EARTH Operations Supported by RTS 2.5

Message Name	Tag	Length (Bytes)
SYNC	1	12
SPAWN	2	16
INCR_SYNC	3	16
PUT_SYNC	4	20 + data_size
GET_SYNC	5	28
INVOKE	6	16 + arg_size
TOKEN	7	16 + arg_size
TOKEN_REQ	8	8
BLK_MOV_PUT	9	20 + data_size
BLK_MOV_GET	10	28

Table 2.4: Message Formats Defined in RTS 2.5

2.3.5.3 Receive Module

The *Receive Module* performs blocking call to *select()*, when kernel detects traffic for any of the incoming socket buffers watched by the *select()*, it wakes up the RM. Then the RM can get the incoming message with a *read()*, and handles this message according its type. After handling the message, the RM puts itself back to sleep by calling *select()* until the next incoming message arrives. Below is the summary of how RM handles different types of message:

1. *SYNC*, *SPAWN*, *INCR_SLOT* and *INVOKE* messages are directly put into the *Event Queue*, and will be handled by EM.

2. For *PUT_SYNC* and *BLO_MOV_PUT*, the RM copies the data to the local destination memory address, then check *SYNC* operation attached. If the address of sysn slots is local, the RM will put the *SYNC* message to the *Event Queue*, otherwise, the RM will put this *SYNC* message to the *Remote Send Queue*.

3. For *GET_SYNC*, the RM fetches the request data from local memory, then generates a *PUT_SYNC* message to the *Remote Send Queue*.

4. For *BLK_MOV_GET*, the RM generates a *BLK_MOV_PUT* message into the *Remote Send Queue*.

5. For *TOKEN*, *TOKEN_REQ*, the RM forwards them to the *Remote Send Queue* and lets the *Token Manager* handle them.

2.3.5.4 Execution Module

The *Execution Module* performs two different kinds of jobs: Firstly, it executes the Threaded-C code, and handles all EARTH Operations embedded in the code by directly executing them or generating corresponding messages to the *Send Queue* or *Token Queue*; Secondly it polls the *Event Queue* to get and handle the messages in EQ.

Summary of how the EM handles the EARTH operations embedded in the Threaded-C code is shown below:

INVOKE If the specified node is self, then put the first fiber of specified threaded procedure into RQ; otherwise, generating a INVOKE message to SQ.

TOKEN Generating a TOKEN message into TQ.

SYNC If the sync slot is a local slot, decreases the sync count; If the count reaches zero, the associated fiber is put into RQ. If the sync slot is a remote slot, generating a SYNC message into SQ.

INCR_SLOT If local, increase the specified sync slot, otherwise a INCR_SLOT message is put into SQ.

SPAWN Put the specified local fiber into RQ.

PUT_SYNC If the destination address is local, copies the value to the destination address, then handles the SYNC part. If it's remote, generating a PUT_SYNC message into SQ.

GET_SYNC If the both source address and destination address are local, does a memory copy, then handles the SYNC part. If the source is remote, generating a GET_SYNC message into SQ.

BLKMOV_SYNC If both the source address and destination address are local, does a memory copy, then handles the SYNC part. If the source is local, the destination is remote, generating a BLK_MOV_PUT message into SQ. If the source is remote, the destination is local, generating a BLK_MOV_GET message into SQ.

After EM executes all ready fibers in *Ready Queue* and all available TOKENs in *TOKEN QUEUE*, it goes to poll the *Event Queue*, and handles the messages in EQ as below:

SYNC Decreases the sync count. If the count reaches zero, the associated fiber is put into RQ.

INCR_SLOT Increases the specified sync slot

SPAWN Puts the specified local fiber into RQ.

INVOKE Puts the first fiber of specified threaded procedure into RQ.

After all messages in the EQ have been polled out and handled, the EM will try to find new ready fibers in RQ or new TOKENs in TQ and execute them.

2.3.5.5 Send Module

The *Send Module* and the *Token Manager* are implemented in the same Pthread. The *Send Module* polls the SQ of each EM and RSQ, and handles messages differently according to the message type:

1. For *SYNC*, *INCR_SLOT*, *PUT_SYNC*, *GET_SYNC*, *INVOKE* and *BLK_MOV_GET* messages, the SM simply copies them from the SQ or RSQ to the corresponding socket.

2. For *BLK_MOV_PUT*, the SM first sends the message head, and the retrieves the data as message body from local memory and sends it out.

3. For *TOKEN*, *TOKEN_REQ* messages, the SM passes them to the TM and let the TM handles those dynamic load balancing related messages.

2.3.6 Dynamic Load Balancers in RTS 2.5

Dynamic load balancing is always a crucial task since the first version of EARTH RTS. The RTS 2.5 continues to improve the dynamic load balancer and extend dynamic load balancing algorithm.

Normally, there are two types of load balancing: static load balancing is done by the programmer or compiler before runtime, it is applicable for regular applications; dynamic load balancing algorithm use system state information to make runtime decision on how to dispatch workloads, it can be used for applications, whose communication patterns and the granularity of work pieces are impossible to predicted before runtime.

Different kind of dynamic load balancing algorithms are discussed and studied by Haiying Cai and Kamala P. Kakulavarapu in their master theses and papers [30][31][32]. In the RTS 2.0, *dual* algorithm, which is an receive-initiated load balancer, is implemented and used by several representative benchmarks. In the RTS 2.5, a new load balancer, called *dual-startup* is implemented. It addresses the problem of increasing propagation latency of work loads with the number of computing nodes increasing. The implementation details and experimental results about this new load balancer can be found in Chuan Shen's master thesis [27].

The implementation of another new load balancer, called master-slave load balancer, is motivated by the real-world application – parallel hmmpfam, which is the main contribution of this thesis. It will be discussed later in Section 3.4.2.

Chapter 3

CLUSTER-BASED SOLUTION ON EARTH

3.1 The Problem of PVM HMMPFAM Implementation in HMMER

As we have introduced, Hmmpfam reads a sequence file and compares each sequence within it, one at a time, against all the HMMs in hmm database, looking for significantly similar matches. Figure 2.1 shows the basic algorithm of HMMpfam.

A parallel hmmpfam program based on PVM is included in the HMMER 2.2g. Figure 3.1 shows the task space decomposition of the parallel scheme in the current PVM implementation. In this scheme, the master-slave model is adopted, and within one stage, all slave nodes work on the computation for the same sequence. The master node dynamically assigns one profile from the database to a specific slave node, and the slave node is responsible for the alignment of the sequence to this HMM profile. Upon finishing its job, the slave node reports the results to the master, which will responds by assigning a new job, i.e. a new single profile, to that slave node. When all the computation of this sequence against the whole profile database is completed, the master node sorts and ranks the results it collects, and outputs the top hits. Then the computation on the next sequence begins.

However, the experimental results shows that this implementation does not achieve good scalability with the number of computing nodes increasing(Fig. 6.1, Fig. 6.2). The problem is that the granularity of computation is too small relative to the communication overhead. Moreover, the master node becomes a bottleneck when the number of the computing nodes increases, since it involves in both communications with slave nodes and computations such as sorting and ranking. The

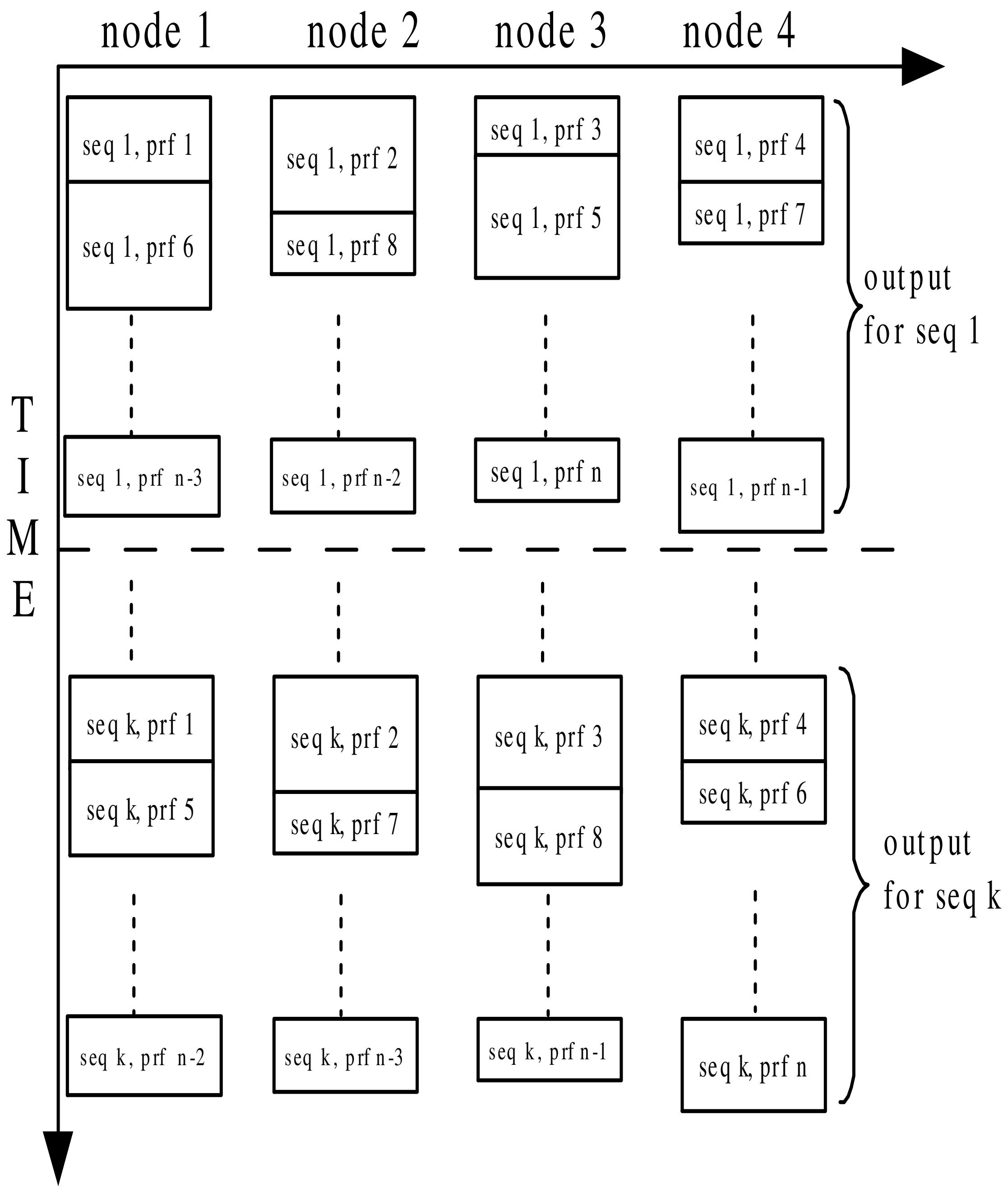

Figure 3.1: Parallel Scheme of PVM Version

implicit barrier at the end of the computation of one sequence also wastes the computing resources of the slave nodes.

3.2 Embarrassingly Parallel Computing Strategy

The basic idea of the hmmpfam algorithm is to read a single sequence from seqfile each time and to compare it against all the HMMs in the HMM database file looking for significantly similar sequence matches. To efficiently parallelize this application, it is important to determine the granularity of computation. In the original scheme, the alignment of one sequence with one profile is considered a single job. By profiling a typical benchmark on a Pentium III 500MHz machine, we find that the average computation time for such a single job is only 0.03 seconds. Thus the granularity of computation in original parallel scheme is too small relative to the latency and the overheads.

However, if we consider the computation of one sequence against the whole database as a single job, then the job itself is sufficiently big. Normally the number of sequences in a sequence data file is much larger than the number of computing nodes available in current Beowulf clusters. So the number of single jobs is still large to keep all nodes busy, thus to achieve ideal parallelism. Moreover, because all the computation of one single sequence will be performed by one process on one fixed node, there is no need to send back the result to the master node, and the sorting and ranking can be executed locally. This means after a job is assigned to one slave node, there is no more communication needed between the master and this slave, and there is also no implicit or explicit barriers needed.

Therefore, the hmmpfam algorithm can be immediately divided into completely independent parts that can be executed simultaneously. It is an ideal embarrassingly parallel computation problem, which could be efficiently implemented by classical master-slave parallel model. By using the embarrassingly parallel programming strategy, the new parallel hmmpfam version can take advantage of the current advanced supercomputing clusters to reduce the execution time dramatically with scalable speedup. The following subsections will illustrate how the cluster-based solution for parallel hmmpfam is implemented on EARTH.

3.3 A Two Level Master-Slave Parallel Scheme

To exploit the parallelism of hmmpfam algorithm, our multithreaded parallel scheme divides the work-load at two different levels. In Fig. 3.2, each circle represents a THREADED procedure, which is a C function containing local states (function parameters, local variables, and sync slots) and one or more fibers of the code. A THREADED procedure is the basic element to be scheduled. Either the programmer or EARTH RTS can determine where (on which node) the procedure get executed.

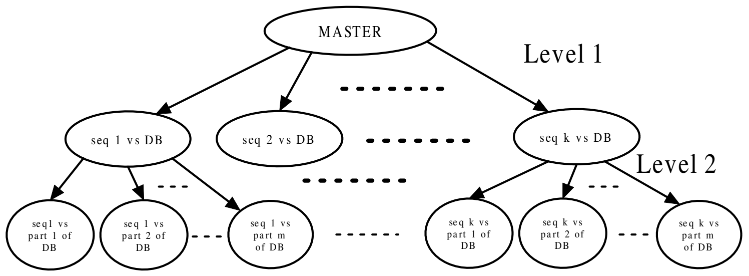

Figure 3.2: Two Level Parallel Scheme

At the first level, the master process assigns each sequence to one and only one procedure. Each procedure will complete the computation for its sequence against the whole HMM database, then collect and handle the alignment results, and output the top hits to a file on the local disk. The job at this level is large and independent to other jobs in the same level, so this level exploits the coarse-grain parallelism. The jobs are distributed either manually by programmer or automatically by RTS. Once a job is assigned to a slave procedure, there is no synchronization and barrier required, and no more communication needed between the master and slave processes.

At the second level, in order to achieve further parallelism, the HMM database file is divided into a number of partitions. Each procedure in the second level gets one partition of the database, and performs the corresponding computation, then returns the result to parent procedure. This level exploits the fine-grain parallelism. The jobs are distributed at runtime by the dynamic load balancer of RTS.

3.4 Cluster-Based Solution

To provide the cluster-based solutions, currently the first parallel level in Fig. 3.2 has been succesfully implemented by two different approaches: one predetermines job distribution on all computing nodes by a round-robin algorithm; the other takes advantage of the dynamic load balancing support of EARTH Runtime system, which is more robust and can also simplify the programmer's coding work by making the job distribution completely transparent. Details about those two implementations are described in following sections.

3.4.1 Static Load Balancing Approach

In the static load balancing implementation, as shown in Fig.3.3, the job distribution is pre-determined. Programmer takes responsibility to explicitly indicate the node, on which a specified job will run. To achieve an even work load on all computing nodes, round-robin algorithm is adopted.

At the initiation stage, master process reads a single sequence from seqfile each time and generates a new job by invoking a threaded procedure on the specified node according to the round-robin algorithm. On each computing node, the EARTH RTS maintains a ready queue, which contains the ready-to-run fibers. The RTS will automatically put the first fiber of this procedure to the ready-queue of the node appointed by programmer. As soon as a node gets a new job, it will begin the computation immediately. After the initiation stage, all jobs have been put into the destination computing nodes, so the execution unit on each node can successively

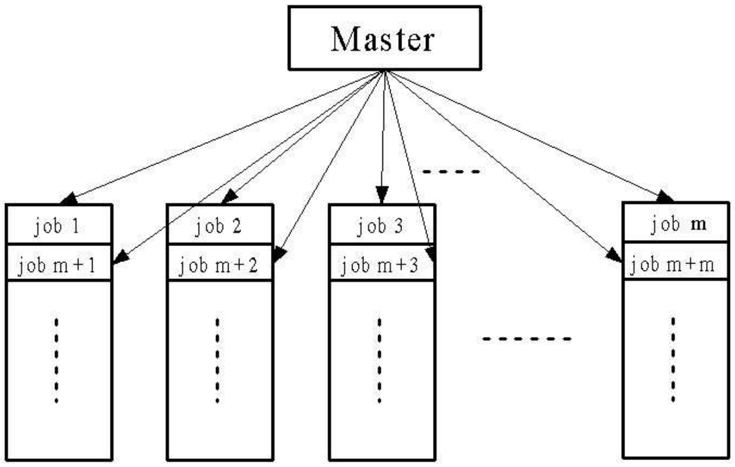

Figure 3.3: Static Load Balancing Scheme

fetch new jobs from the ready queue until all jobs are done. That means in the computation stage, all nodes become independent and execute jobs sequentially without frequent context-switch due to the non-preemptive fiber support of EARTH. If there are very large amount of sequences in the seqfile, which is a very common case, this approach can achieve evenly balanced work load and good scalability.

3.4.2 Robust Dynamic Load Balancing Approach

One of the most important features of EARTH is the dynamic load balancing support. Dynamic load balancer uses system state information to make runtime decision on how to dispatch workloads. The design of dynamic load balancer focuses on two objectives: 1) keeping all the nodes busy; 2) optimizing load balancing by minimizing balancer overheads and maximizing benefits due to load balancing.

The target of parallelizing hmmpfam motivates us to design a special load balancer in EARTH RTS 2.5 – master-slave load balancer, which can be used for all master-slave parallel applications. This load-balancer is working like a server-client

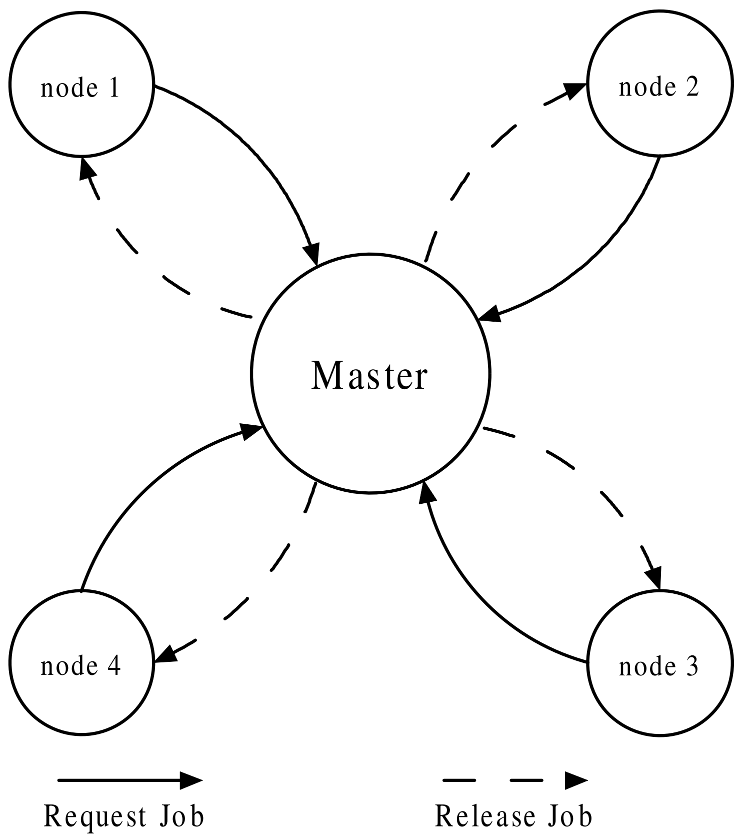

Figure 3.4: Dynamic Load Balancing Scheme

model as illustrated in figure 3.4. Once a slave node finishes a job, it sends a request to the master process. Master responds by sending back a new job to satisfy the slave's work requirement thus to keep it busy. The overhead of load balancing is relatively small since each single job is large enough. With this load balancer, the job-request and job-assignment are determined by EARTH RTS dynamically.

Compared with the pre-determined job assignments strategy, the dynamic load balancing approach is robust. For example, with static load balancing, all jobs have been put into the ready queue of computing nodes at initiation stage, and can not be moved away after that. If one node is busy with computing tasks from other users, the remaining jobs in this node can not be re-assigned to other idle nodes.

While with dynamic load balancing strategy, this situation, in which some nodes are busy forever and some others are free, is avoided. Since the hmmpfam tool may run for quite a long time, for instance, several months in real situation, the robust dynamic work distribution strategy is quite practical. Also in order to run the parallel hmmpfam on the advanced supercomputing clusters, which are consisted of hundreds of computing nodes, a robust approach is necessary, since it's not easy to guarantee all nodes working properly without any system problem or the intrusions from other user-level applications during the long running time.

With the dynamic load balancing support of EARTH RTS, the job distribution is completely transparent to the programmer. The programmer only concerns about the generation of new jobs, and does not need to determine where the jobs will be executed. The EARTH RTS takes over the responsibility to distribute jobs at runtime, which quite simplifies and eases programmers' coding work and makes it conceptually straightforward.

Chapter 4

PORTABILITY OF EARTH PROGRAM

The cluster based solution of parallel hmmpfam is presented in Chapter 3. When we begin to design the parallel scheme for hmmpfam, the clusters are the target platform. The experimental results (see Chapter 6)shows that near linear speedup achieved on the supercomputing clusters. Then two questions immediately raised:

> **1)** Are the EARTH programs (hmmpfam, in this thesis) portable from clusters to SMP machines and SMP clusters? And why?
>
> **2)** Are the EARTH programs "performance portable" from clusters to SMP machines and SMP clusters? And why?

The first question focus on the portability of the code itself, it wants to ask *"If an EARTH implementation of one application is already available on one platform*[1]*, is this code portable to other platforms without code modifications but still keep correctness?"* We call this problem the *"code portability"* problem of one EARTH program.

The second question consider another aspect of the portability of one EARTH implementation, it tries to ask *"If an EARTH implementation of one application already achieves good performance*[2] *on one platform, can this implementation also*

[1] *In this and next chapter, platforms refer to uni-processor clusters, single SMP machines and SMP clusters*

[2] *The performance here refers to absolute speedup of the parallel program.*

achieve the same good performance on other platforms?" We name this problem the *"performance portability"* problem of one EARTH program.

We claim that the EARTH program can achieve both *code portability* and *performance portability* on various platforms. Following subsections will try to illustrate how these two portability achieved for applications parallelized by EARTH PXM.

4.1 Code Portability of EARTH Program

The code portability cares whether it is possible for the same code to port to different platform without any modification. I will try to explain the code portability of the EARTH program from the point view of EARTH virtual machine. The current implementation of EARTH (RTS2.5) are based on the EARTH Virtual Machine (EVM) – EVM-A, defined in Kevin Theobald's doctoral thesis [12]. The EARTH Virtual Machines specifies the interface between the programmer/compiler and the hardware. The goal of EVM is "to defines a common set of operations in sufficient detail to permit both multiple hardware platforms and multiple high-level languages to be designed around this set"[12]. The EVM defines the memory model, thread model, data types and EARTH instructions for implementing EARTH on distribute-memory computers and SMP computers.

In Kevin Theobald's doctoral thesis, he defines the first EARTH virtual machine, which is called EVM-A (stands for an EVM based on *Addresses*). The design of the EVM-A is based on the global addresses. Because the most intuitive way to make it possible for referencing memory locations globally is to extend the concept of local memory address to the global address.

For the distributed memory environment, the memory is not physically shared by all the computing nodes. The EVM-A lets each node has its own memory space, which is divided to two parts: *replicated address space* and *non-replicated address space*. The replicated space must be identical on all nodes. A copy of the executable code used by the EARTH program is placed at the same address (base address) in the replicated space on each physical node. This make it possible that any addresses (branches, sequential function calls, threaded procedures, fibers and static data structures) in the executable code to refer to the identical code or data structure on each node. The stacks, frames, which hold the contexts of instances of threaded procedures, and the dynamically allocated objects are put in the non-replicated space. For a remote node being able to access the memory location in the non-replicated space, the global address is defined as the combination of node id and the local address on that node. EVM-A requires all global address accesses must be done through defined EARTH operators, instead of directly using load and store instructions.

For the SMP machine, the memory space is shared by all processors, and there is only one copy of executable code in this space. Theoretically, the processor is able to access any memory location in this space. However the EVM-A requires the memory accesses to other threaded procedures' frames and heaps must also be done through the defined EARTH operators.

So from the point view of memory addresses of EVM-A, the key points of how to make the code portable on different platforms are how to design the interface between programmer and RTS, and how to implement those EARTH operators responsible for inter-node memory accesses in RTS.

Firstly the interface between the programmer and the EARTH RTS, is the Threaded-C [23][24] language, which is briefly introduced in Section 2.2.3. The programmer of Threaded-C is not assumed to be aware of the underlying platform used. From the programmer's view, a number of nodes are available for running threaded procedures and fibers. When the code in one threaded procedure tries to access the memory locations or sync slots in other thread procedures' frames or dynamically allocated spaces, the corresponding EARTH operators are called by given the global addresses of that memory location. The EARTH operations, which take the global addresses as parameters, are responsible for these globally memory accesses.

In the EARTH RTS, a "node id" in Threaded-C, is actually a unique identifier for a Execution Module (EM). For example, for two nodes – A and B, if node B's corresponding EM is performed on the same machine as node A's EM, we say that node B is local to node A, otherwise, it is remote to node A.

All the EARTH operators are implemented in the EARTH RTS. When a global address is passed to RTS. The RTS extracts both the identity of the node and the local address on that node. The RTS will determine the node is local or remote. If the node is located on a remote machine, the memory access request containing the specified memory location will be forwarded to the correct machine, and handled by the RTS on that machine. If the node is just on the same machine, the RTS will satisfy the memory access request immediately by direct memory load or store.

Now, from the point of view of programming based on the EVM-A memory model, there is no difference for global address accesses on different platforms. The address space is smoothly extended from a single-CPU machine to a cluster, from an SMP machine to an SMP cluster, which plays the most important role to make the EARTH program portable on different platforms.

Based on EVM-A memory model, the EVM-A thread model is defined. The code for a threaded procedure is stored at the same addresses in the replicate space of each physical node. So that procedure can be referenced on all the nodes by the same address. This is also true for individual fibers within a procedure. This globally uniformed reference to procedures and fibers make it possible to invoke the procedures on a remote machine. For synchronization with another threaded procedure, specific EARTH operators are used to increase or decrease the count of the sync slots within that procedure. A reference to other procedure's sync slots is also global address. The RTS handles the sync slots related request in the same way for global memory address accesses. The operations on sync slots may cause new fiber enabled, and RTS will put the enabled fibers into the Ready Queue. The RQ pairs the address of a fiber's code with the address of the frame associate with the instance of that fiber. EM takes the fiber from RQ for performing execution.

Based on the EVM-A thread model, the threaded procedure invocation, inter procedure synchronization makes no difference on different platforms from the point of view of coding. A uniformly address of the thread procedure is used by programmer to invoke that thread procedure, no matter locally or remotely. The global addresses of the sync slots are used by the programmer to do the synchronization among procedures, no matter locally or remotely. The RTS is responsible for the correctness of all those EARTH operations on different platforms.

EARTH EVM-A is defined based on EARTH Program Execution Model (PXM). The EARTH PXM is focus on the high-level description of the structure of EARTH abstract machine, the EARTH thread model and the fundamental EARTH operations, while the EVM-A specify the details about how the EARTH operations are encoded and how the memory is shared among the nodes to provide a general model for multithreaded execution on different platforms. To develop an EARTH application, the Threaded-C programming is done by using a dataflow-like event-driven synchronization model defined by EARTH PXM [12][33], which works smoothly with the EVM-A memory model and thread model. Thus we can draw the conclusion that the code portability is mainly due to the successfully defined EARTH EVM-A and the corresponding implementation of EARTH RTS.

4.2 Performance Portability of EARTH Program

Once the code portability is guaranteed, the next issue is whether the performance is portable to different platforms. More clearly, if the EARTH program of one application is able to achieve good performance on one platform, can the same program also achieve the same good performance on different platforms? I try to answer this question from the implementation of the EARTH EVM – the EARTH RTS. The current implementation of EARTH RTS – version 2.5, is reviewed in Section 2.3.

The design of EARTH RTS 2.5 is targeted to achieve portable performance on different platforms – uni-processor cluster, single SMP machine, and SMP cluster. The implementation is instructed by the EARTH EVM-A as described in Section 4.1.

Overall, the EARTH RTS 2.5 running on one physical node itself is a multithreaded parallel program powered by POSIX Thread [34]. There are two types of modules in the EARTH RTS: Execution Modules (EM), Network I/O Modules (Receive Module (RM), Send Module (SM)). Each module is implemented by one POSIX thread. The modern multiprogramming operating system is responsible for scheduling those modules to utilize CPU resources by time-sharing based policies.

The EM is the module performing real computations. Each EM is corresponded to one conceptual node in Threaded-C program. RTS is responsible for creating EMs on different machines, which is transparent to the programmer. EMs are feed by EVM threaded procedures and fibers to keep busy. When running on SMP nodes, multiple EMs are allowed to be created. Normally the number of EMs created is equal to the number of processors available. The parallelism of the application is exploited by the Threaded-C program. Keeping EMs usefully busy is, indeed, the feature of event-driven multithreaded EARTH PXM, which has made the performance robust as long as there is plenty of parallelism. When enough parallelism is available, the RTS is responsible to keep the EMs as busy as possible. In RTS 2.5, the EM will never yield the CPU by itself, it always tries to find new enabled fibers from its ready queue, and executes them. EM may perform some

EARTH SU functions, however normally it responses to the new ready fibers fast enough.

On the other hand, all network operations are completed by RM and SM. All remote procedure invocations, remote memory accesses, and remote synchronizations requested by EMs are handled by RM and SM. When no such requests exists, the RM and EM yield CPU resources by themselves, thus EMs get great chance to occupy the CPU as much as possible. When network I/O requests happen, the RM and SM will be waked by the OS, handle them immediately and go to sleep again. So the computation and communication within one node is well overlapped. To draw the conclusion, the available parallelism represented by threaded procedures and fibers keep the EMs as busy as possible, while communication is fast responded and handled by the network modules.

For a computation intensive application, like Hmmpfam presented in this thesis, when good performance achieved on one platform, it is able to port to different platforms. Since satisfying performance is already shown on one platform, it can be concluded that the Threaded-C program is well coded, enough parallelism is always available. Because the amount of communication is relatively small compared to computation, there is no big difference from running the program on different platforms. In case of dynamic load balancing, where the work load balancing is determined dynamically by RTS, instead of the programmer, since the same load balancing algorithm is used for all platforms, the performance is still able to be portable when the network latency is small enough.

For a communication intensive application, achieving scalable good performance is a big challenge for any parallel program execution model. Normally, a lot of coding work is involved to get efficient implementation. For EARTH, if we know an application already achieves good performance on the cluster, we can imagine that the same performance is portable on the SMP machine. Since EARTH PXM is an event-driven fine-grain multithreaded dataflow-like programming model, the Threaded-C program can be described by a dataflow-like graph [33]. Thus the EARTH EVM has the same inter-procedure communication pattens for all platforms, the difference is the communication latency. For cluster the communication is done by network, which has long latencies, while for SMP the communication is done through directly memory transactions, which has small latencies. And the EARTH EVM does not use mutual exclusion like synchronizations, while excessive mutual exclusions always plays as the performance killer of parallel applications for other parallel APIs, which are targeted to SMP specially. Thus for EARTH the performance is portable from cluster to SMP machine for communication intensive application. In the future, by integrating the state-of-the-art high performance networking to EARTH RTS, we can expect that the performance is also portable from SMP machine to clusters.

Up to this point, we can see that one of the great properties of EARTH EVM is that the same code can be portable to different platforms without any modification but still keep the same performance.

4.3 Related Works

This section covers some of the work which is relevant to the code portability and performance portability of EARTH EVM from the perspective of the portability of other popular parallel APIs, e.g. MPI (Message-Passing Interface) [35][36], PVM (Parallel Virtual Machine) [2], OpenMP [37], and PThread (POSIX Threads) [34].

MPI has become an industrial standard for developing parallel applications, especially on clusters. A large amount of applications are parallelized by MPI. MPI is based on message passing, which is natural for the communication network of current clusters. Recently, with the prevalence of the SMP clusters, a lot of work has been conducted to improve MPI to support SMP clusters, thus to get the portability for current parallel applications, which are implemented by MPI. Since the version of

1.1, the MPICH [38] developed by Argonne National Laboratory, uses shared memory for communication within an SMP node to achieves better support for SMP nodes within the cluster. Hong Tang and Tao Yang's work – TMPI [39][40], shows that using threads to execute MPI program can achieve great performance on SMP clusters. They design and implement a threaded-based MPI system on SMP clusters, which shows the performance improved substantially over the process-based MPI. Steve Sistare [41] introduces new algorithms for MPI collective operations, which can take advantage of the computation capability of SMP node, thus improve the performance of MPI on SMP clusters. Tokahashi and his colleagues [42] designed and implemented an MPI library called MPICH-PM/CLUMP for SMP clusters. The MPICH-PM/CLUMP supports multiple processes on the SMP nodes of the cluster by realizing zero-copy transfer of messages between SMP nodes and one-copy transfer of messages within an SMP node. On the other direction, some research groups try to port the MPI program designed for clusters to a single SMP machine, Demaine [43] implemented a threads-only implementation of MPI, called TOMPI [44], which aims to efficient development of MPI program meanwhile achieve good performance on a single workstation, either a uni-processor machine or an SMP machine. By the use of threads and shared memory, instead of UNIX processes and sockets, TOMPI makes it possible to port existing MPI program to SMP workstations without modification. However, it is worth noticing that the same EARTH RTS running on different platforms to perform the same Threaded-C coded applications seamlessly, while the TOMPI system can only running on a single workstation.

Another widely used parallel API for clusters is the PVM, which is developed by Oak Ridge National Laboratory. Various groups make efforts to achieve efficient PVM implementation on SMP machines. PM-PVM (Portable Multithreaded PVM) [45] is an implementation of PVM targeted to SMP architectures. PM-PVM maps PVM tasks onto threads and performs PVM message passing functions through shared memory. Consequently, PVM programs get supported on SMP machines by running on PM-PVM. PM-PVM is portable on different SMP platforms. LPVM (lightweight-process PVM) [46] was implemented on SMPs by using of lightweight processes and threads as the basic units to exploit parallelism. The LPVM slightly changes the PVM API, which requires code modification when porting existing PVM applications. TPVM [47][48] is an extension to PVM system, but using threads as basic unit for computing and scheduling. TPVM is implemented by using TREX thread package, which has been only ported to several architectures, which limits the portability of TPVM. And TPVM also has its own API extended from PVM API, which also make it hard to port PVM program to TPVM. Experiments shown in those papers do not provide enough evidences that PM-PVM, LPVM, TPVM is portable to uni-processor clusters and SMP clusters.

OpenMP is the most widely acceptable API for parallelizing applications on multiprocessors. The compiler directives based OpenMP API is an very friendly interface for programmer, which makes OpenMP programming is easier than most of other APIs. Then the question comes that whether it is possible to port the OpenMP environment to distribute memory systems, thus make OpenMP programs portable to different platforms. Hu et al. [49][50] present the first OpenMP system implementation on a network of SMPs. A translator is implemented to convert OpenMP directives to the corresponding function calls of the underlying software distribute memory systems. ParADE [51] is an OpenMP programming environment for cluster of SMPs on top of the multithreaded software distribute shared memory system (SW-DSM). To enhance the performance on the SMP clusters, ParADE runtime system provides explicit message-passing operators to make it a hybrid-programming environment. To achieve code portability, this level of hybrid-programming is hided and transparent to OpenMP programmer, which is made possible by an OpenMP translator implemented in ParADE runtime system. Mitsuhisa Sato et al. [52][53][54] demonstrates an implementation of “cluster-enabled” OpenMP compiler, which translates OpenMP program into a parallel program on SCASH, which is a page-based software distributed shared memory system. By the coordination of the compiler and SCASH, the OpenMP program is able to run on the clusters transparently. It can be seen that all those work are based on some kinds of SW-DSM. Similar efforts also conducted to implement POSIX Thread or other thread library on top of both SW-DSM and HW-DSM, for example, THROOM[55] and CableS[56]. However, the DSM itself may meet difficulties when performed on a large number of computing nodes. Thus, the scalability of those OpenMP or POSIX Threads systems on large supercomputing SMP clusters has not been convincingly demonstrated, which requires further research and development.

Hybrid programming, like OpenMP plus MPI, is regarded as one solution to achieve code portability and performance portability on multiple platforms. The idea is that the shared address space within each node is suitable for shared memory APIs, like OpenMP, Pthread, and meanwhile message passing can be utilized across the nodes within a cluster. Thus multi-level parallelism is achieved by integrating shared memory programming and message passing programming. There are many published reports on performance of hybrid programming and the comparison of hybrid programming with other programming paradigm. Here we are only able to name a few of them. Lorna Smith et al. [57][58] presents their approaches and performance analysis on development of mixed MPI/OpenMP application. Cappello et al. [59][60] compares the performance of NAS benchmarks implemented by MPI versus hybrid MPI/OpenMP on different platforms. A comparison of OpenMP, MPI and hybrid programming on SMP cluster is presented in [61]. And Rabenseifner [62] discussed the performance problems and chances of hybrid programming. Hybrid programming can achieve good performance on SMP clusters, however, large amount of programming efforts needed to port OpenMP and MPI program to the hybrid paradigm. And hybrid programming requires programmer to parallelize applications by multiple level parallelism, which is difficult by itself.

Some newly proposed programming model, like UPC [63][64][65], Co-Array Fortran [66][67][68], may also be candidate APIs for developing parallel applications running on different platforms with code and performance portability.

Chapter 5

PARALLEL SMP SOLUTIONS ON SMP

As newer shared memory machine architectures started to become prevalent since 1997, it's also important to investigate how to parallelize the applications on SMP platform or distribute shared memory system. Two popular APIs – POSIX Threads [34] and OpenMP [37], for developing parallel application on shared memory machine, have been used widely.

It's also an important task to get the hmmpfam working on the SMP machines. In the HMMer package, the hmmpfam is parallelized for SMP by Pthreads, however, their implementation appears that can not make Pthreads and PVM work together at the same time. In case of an SMP cluster, they can only use the PVM version, and the experiment results show the performance is not good. For comparison, we also developed an OpenMP version of hmmpfam, which is only targeted to SMP platform. Luckily, our EARTH version can work on all cases – uni-CPU clusters, SMP clusters, and SMP machines, with the binary compatibility, and still shows good performance in all cases.

In following sections, details about the parallel implementation of hmmpfam by Pthreads, OpenMP and EARTH will be given.

5.1 Pthreads HMMPFAM Implementation

In HMMER package, if the hmmpfam program is compiled with the switch "*-DHMMER_THREADS*" and linked with the Pthreads library, it is targeted to the SMP machine with multi-threading. The multi-threaded code is implemented by

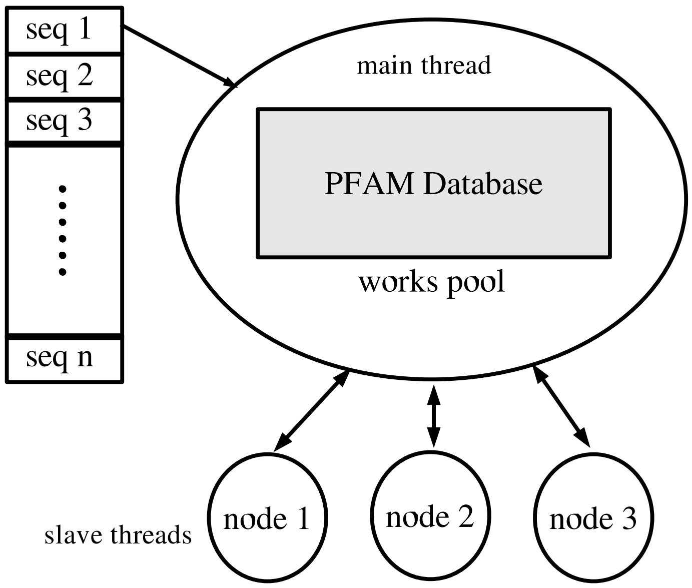

Figure 5.1: Hmmpfam Parallel Scheme of Pthreads Version

using POSIX threads APIs [34]. The parallelization scheme of this Pthreads version is shown in Figure 5.1.

At the beginning of each iteration, the *main* thread reads a sequence from the seqfile. Then the *main* thread creates *slave* threads, and the number of *slave* threads generated is equal to the number of processors available. All *slave* threads get jobs from the work pool, which contains all the work load. The sequence alignment between one HMM profile and the sequence is considered as one unit of job. Each time, the *slave* thread gets one job (one hmm profile) from the work pool, and scores the sequence comparing with this HMM profile, finally stores the output result in the top hits structures. Then this *slave* thread tries to get a new job from the work pool until all HMM profiles in PFAM database has been compared with this sequence. After all HMMs have been compared with the sequence, all *slave* threads terminated, the main thread reports the overall sequence hits, sorted by significance. At this point the computation for one sequence is completed, the main thread reads another new sequence from the seqfile and a new iteration begins. The time line of the whole execution is the same as the PVM implementation in the HMMer package, as shown in Figure 3.1.

The same as the PVM implementation, the computation granularity of Pthreads version, which is the comparison between one sequence and one HMM profile, is still very small. However, because the number of processors available for common SMP machines is not very large, normally 2 to 12. So even the granularity of computation is very small, in this case, it does not hurt performance too much. If we imagine running this Pthreads version on one SMP supercomputer with more than 32 CPUs, this will be a big problem.

The main overheads in this Pthreads implementation is the cost of mutex lock. There are two sources incurring the locks. Firstly, each time a *slave* thread tries to get a new job from the work pool, it needs to acquire the lock on HMM database input, and get the next HMM profile to work on, after get the new profile, it releases the lock, then other *slave* threads can get the lock, and then get the new job. Secondly, once a *slave* thread completes the computation, it needs to acquire the lock of the output resource, and then save the output in top hits structures, after it is done, the lock is released, and other *slave* threads can get in. It is obvious that these two kinds of mutex locks serialize the execution somehow. With the number of processor increases, more *slave* threads are working at the same time, this serialization caused by mutual exclusion is expected to be much more severe and finally it becomes hard to gain performance from multithreading execution.

Another overheads in this Pthreads version is the explicit barrier at the end of the computation of one sequence against the whole PFAM database. All processors need to wait the slowest one to complete its job. Then begin the next iteration together. And also only the *main* thread is responsible for sorting the results returned by *slave* threads, and reporting the overall sequence hits, sorted by significance. During this stage, all other processors have to wait without performing any computation task.

Even with all those drawbacks, this Pthreads implementation in HMMer package is able to get ideal speedup for SMP machines with small number of processors. However, due to the design in HMMER package, the pthread implementation of hmmpfam only works on SMP machine. When running on the distribute memory environment, for example, SMP clusters, the PVM implementation should be chosen to use. So for running on SMP machines and clusters, different version of code are used. From the aspect of pthread programming, the programmer should be aware that the memory is shared by all the processors, the shared resources have to be mutex protected to guarantee the correctness, and meanwhile to avoid excessive locking (deadlock) and inadequate locking (data races). Later, we will show that for the EARTH implementations, the same code works on SMPs, clusters and SMP clusters with compatibility, and the EARTH programmer does not need to worry about the locks.

5.2 OpenMP HMMPFAM Implementation

OpenMP [37] may be the most popular programming API supports multi-platform shared-memory parallel programming in C/C++ and Fortran on all architectures, including Unix platforms and Windows NT platforms. And it currently gains the support by almost all major SMP machine vendors. For comparison, it would be valuable to implement an parallel version of hmmpfam by using OpenMP API.

Beginning from the observation in section 3.2, I implement the OpenMP version of hmmpfam by the same parallel schemes presented in section 3.3. Still, only the first level parallelism is exploited in this implementation, that is the comparison between one sequence and the whole PFAM database considered as the basic unit of jobs. The parallel region in the OpenMP version of hmmpfam is shown in Figure 5.2:

```
1 #pragma omp parallel private(i) shared(tasks)
2   {
3 #pragma omp for
4   for(i = 0; i< num_seqs; i++)
5     worker(tasks[i]);
6   }
```

Figure 5.2: OpenMP Hmmpfam Work-Sharing Construct: Parallel *for*

Line 1: the fundamental OpenMP parallel construct, declares the following block of code is the parallel region, which will be executed by multiple threads. When the *parallel* directive is reached, a team of threads are created. Starting from the beginning of the parallel region, the code in the block is duplicated and all threads will execute that same code. And variable *i* is private for each slave thread, the *task* is shared by all threads. The *task* is an array built to contain all jobs, it can be considered as a work pool.

Line 3: the OpenMP "for" work-sharing struct, it specifies that the iterations of the loop immediately following it must be executed in parallel.

Line 5: the *work* function is used to complete the computation between sequence *i* and the whole PFAM database.

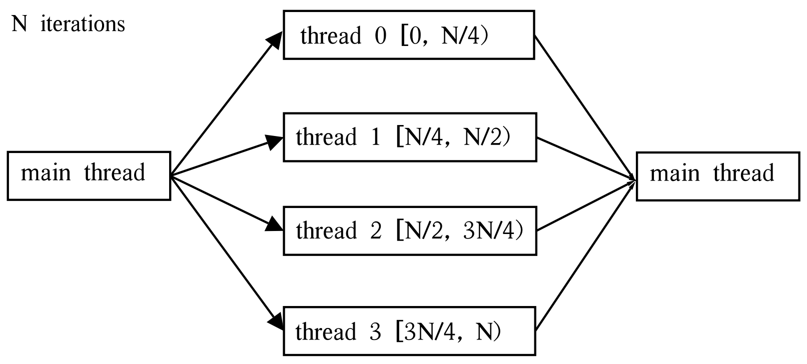

Figure 5.3: OpenMP Static Scheduler

OpenMP has several different schedulers to control how work is distributed among threads. The *static* scheduler divides iterations evenly among the threads, as shown in Figure 5.3. The scheduler divides the work load in to *chuck* size parcels, where *chunk* is a variable can be set by programmer, with default value 1. For the static scheduler, if there are N threads, each thread does every Nth chunk of work.

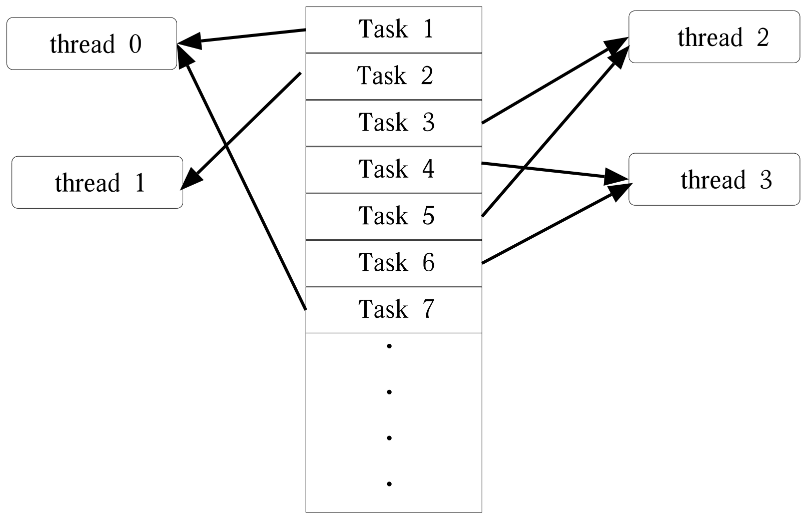

Figure 5.4: OpenMP Dynamic Scheduler

As shown in Figure 5.4, the *dynamic* scheduler also divides work load into *chunk* sized parcels. When a thread finishes one *chunk*, it grabs the next available *chunk*. The *dynamic* scheduler has more overhead, but potentially better load balancing is expected.

The *guided* scheduler works similarly as the dynamic scheduling, the difference is the *chunk* size varies dynamically. During running, the *chunk* sizes depends on the number of unassigned iterations and the *chunk* size is exponentially reduced towards to the specified value of *chunk* with each dispatched piece of the iteration space. The guided scheduler achieves good load balancing with relatively low overhead, by ensuring that no single thread will be stuck with a large number of leftovers, while the others are idle.

In our implementations, all three schedulers are tried and compared. And for all three schedulers, we set the *chunk* size as 1. The result will be shown in section 6.8.

5.3 EARTH HMMPFAM Implementation on SMP

Nowadays SMP clusters are becoming more and more popular due to the following reasons [69][70][71] :

1. the rapidly increasing commodity availability of multiprocessor nodes;
2. mature SMP operating systems;
3. low-latency and high-bandwidth interconnects;
4. superior price-performance ratio.

From the analysis of Chapter 4, we know that the performance is portable from the uni-processor cluster to the SMP machine and SMP clusters. Thus the cluster based solution of Hmmpfam based on EARTH PXM are also the solution for SMP machine and SMP cluster.

With the design of EARTH RTS 2.5, the layout of which is shown in Figure 2.5, multiprocessor node support is available for application coded by Threaded-C. To run fibers on all of the processors in an SMP computing node, it is only necessary to have multiple copies of the *Execution Modules* (EMs) running concurrently. When multiple EMs are active, each has its respective *Ready Queue* (RQ), *Token Queue* (TQ), *Event Queue* (EQ), and *Send Queue* (SQ). The RQ is managed by the EM itself. The *Receive Module* put messages to the EQ of the corresponding EM. EM put the message it generates to its own SQ, and then *Send Module* will poll this queue. The dynamic load balancing within an SMP node, is done by the *Token Manager*, as shown in Figure 5.5. The token manager is responsible for balancing the load among EMs.

As stated in Chapter 4 The advantage of Threaded-C is that the programmer does not need to be aware where the application is executed, the concept of node in Threaded-C is actually referring to the Execution Module (EM), which is a virtual

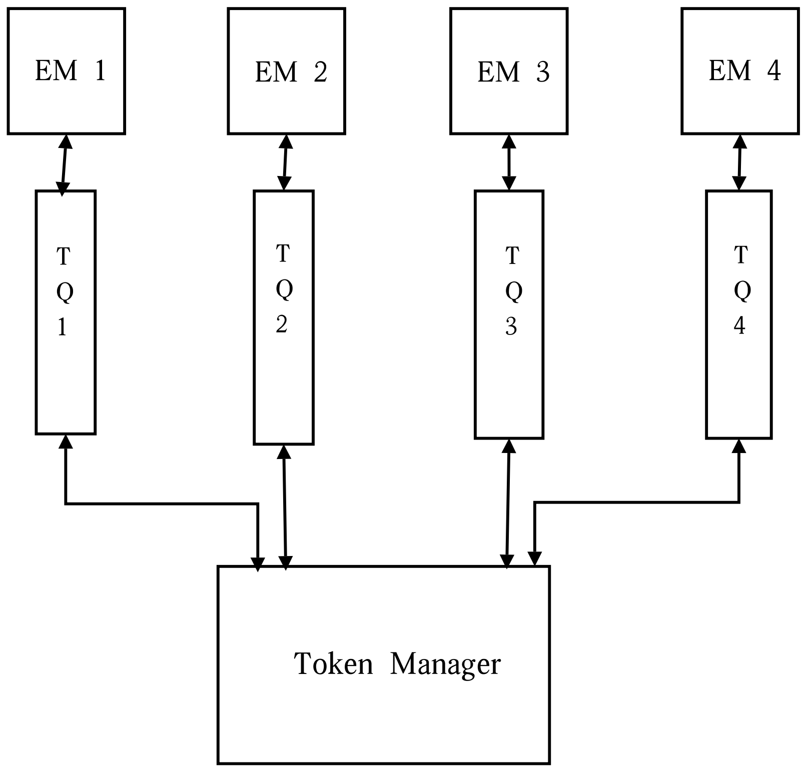

Figure 5.5: Load Balancing for SMP Node in RTS 2.5

node performing computation, instead of a physical node. When the RTS starts, the configuration of the underlying platforms is collected. RTS controls the communication among different physical nodes, and the memory copy among EMs in the same physical node, which is transparent to the Threaded-C programmer. Therefore, the Threaded-C coded applications are portable on clusters and SMP machines. Thus our hmmpfam implementation based on EARTH execution model, without any code changes, can be compiled and then executed with good performance on all platforms with the EARTH RTS support: large clusters, SMP machines, and SMP clusters, and achieves near linear performance on each platform.

Due to the reasons stated above, The Threaded C language is a perfect candidate for SMP clusters[72] programming language because it can fit to different platforms seamlessly.

So our hmmpfam implementation based on EARTH execution model illustrated in section 3.4 not only a cluster based solution, also an SMP based solution, more over it is an SMP cluster solution. The same code, without any changes, it is portable on all platform supported by the RTS: large clusters, SMP machines, and SMP clusters, and gains ideal performance on each platform.

The performance result of the Threaded-C implementation of Hmmpfam on both SMP machine and SMP clusters will be shown in the experimental result chapter.

Chapter 6

EXPERIMENTS AND RESULTS

6.1 Synopsis

The experiments described in this thesis are carried out by using the EARTH Runtime System 2.5 and three different beowulf clusters, two of which are advanced supercomputing clusters in Argonne National Laboratory, and one SUN SMP machine.

This chapter is organized as follows. Section 6.2 provides summary of main observations from experiments. The experimental strategy, platforms and data sets are described in Section 6.3. The EARTH implementation of Hmmpfam is compared with the PVM version in Section 6.4. The scalability results on two supercomputing clusters are presented in Section 6.5 with the performance analysis in Section 6.6. The robustness of the dynamic load balancing approach is discussed and demonstrated in Section 6.7. Section 6.8 compares SMP implementations by different APIs on one SMP machine. Section 6.9 demonstrates that the EARTH version of Hmmpfam can achieve both code portability and performance portability on different platforms. Finally, the conclusions are presented in Section 6.10.

6.2 Summary of Main Observations

The main observations from the experiments design in this chapter are:

Comparison with PVM version. Our cluster based solution of hmmpfam based on EARTH PXM achieves better scalable performance than the original PVM implementation in HMMer 2.2 package. When running on 18 dual-CPU computing nodes, the EARTH version is 40% faster than the PVM version as shown in Section 6.4.

Scalability. Our cluster based solution achieves scalable nearly linear speedup on two large supercomputing clusters, for a representative data set, it reaches an absolute speedup of 222.8 on 128 dual-CPU nodes, which is shown in Section 6.5.

Robustness. The new master-slave dynamic load balancer of EARTH RTS 2.5 helps the hmmpfam implementation achieve more robust performance than the static load balancer. Experiments in Section 6.7 shows the dynamic load balancing approach is much less sensitive to disturbance. When running on 18 dual-CPU nodes with disturbance to 4 CPUs, the performance degradation ratio of dynamic load balancing approach is less than 10

SMP solution. The cluster based solution of hmmpfam is also an SMP based solution. It achieves an absolute speedup of 8.20 by running on 10 processors of an SMP machines. Details are given in Section 6.8.

Performance Portability. The EARTH implementation of hmmpfam is code and performance portable to different platforms. Almost the same speedup is achieved on different platforms. For example, when running on 144 CPUs, the parallel hmmpfam based on EARTH on PXM can achieve the absolute speedup of 121.0 on a uni-processor cluster. When ported to a SMP cluster, it still achieves speedup of 127.1. More results are shown in Section 6.9.

6.3 Experimental Strategy, Platforms and Data Sets

6.3.1 Experimental Strategy and Framework

As most of other literatures in the area of parallel computing, the absolute speedup is used to compare the performance of parallel hmmpfam on different platform. It also used to demonstrate the scalability result on large supercomputing result. Normally, the less difference between the speedup curve and the linear speedup, the better performance is demonstrated.

To highlight the scalability of our approach, the experiments were conducted on two large clusters, one of which ranks 129 in the 22nd TOP500 supercomputer list [4].

To give more insight on the reasons why the EARTH implementation of hmmpfam can achieve very scalable performance, some simple mathematical analysis will be given under some ideal assumptions. When more real conditions added in, discussions are also given.

To compare the robustness of the dynamic load balancing approach with the static load balancing approach, a new term called performance degradation ration under disturbance is defined, the smaller this ration, the better robustness presented.

To demonstrate that the cluster-based solution of hmmpfam based on EARTH PXM is also an SMP solution, SMP implementations by different popular APIs are conducted on the same SMP machine. The comparison is also done by showing the speedup curves on the same figure, which is easily understood.

To argue that the EARTH hmmpfam is performance portable to different platform, the speedups on various platforms are shown in the same figure. If they are all very close in the figure, it means the performance is portable from one platform to others.

6.3.2 Computational Platforms

The comparison of PVM hmmpfam version and EARTH version is tested on **COMET cluster** at CAPSL, University of Delaware. COMET consists of 18 nodes, each containing two 1.4 GHz AMD Athlon processors – total of 36 processors – and 512MB of DDR SDRAM memory. The interconnection network for the nodes is a switched 100Mbps ethernet.

Other experiments are conducted on two large clusters. The **Chiba City cluster** [73] is a scalability test-bed for the High Performance Computing and Computer Science communities at Argonne National Laboratory. The cluster is comprised of 256 computational servers, each with two 500MHz Pentium III processors and 512MB RAM memory. The interconnects for high performance communication are both a fast ethernet and a 64-bit Myrinet.

The **JAZZ** [74] is a teraflop-class computing cluster, provided by Laboratory Computing Resource Center at Argonne National Laboratory. The JAZZ cluster consists of 350 computing nodes, each with a 2.4 GHz Pentium Xeon processor; and 175 nodes with 2 GB of RAM, 175 nodes with 1 GB of RAM. All nodes are interconnected by fast ethernet and Myrinet 2000. JAZZ is ranked 129 in 22nd TOP500 Supercomputing List [4].

The comparisons of different hmmpfam SMP approaches are performed on **dbi-rna1** at Delaware Biotechnology Institute. **dbi-rna1** is a Sun Sunfire 4800 Server with 12 SPARC 750MHz CPUs, and 24 gigabyte memory. This is a famous SMP machine shipped by SUN.

6.3.3 The Data Sets Used in Experiments

For comparison of the PVM version and the EARTH version of parallel hmmpfam, we use a HMM database containing 585 profile families, and a sequence file with 250 sequences. This benchmark is referred as data set-1 in following sections.

For testing both the static and dynamic load balancing version of EARTH hmmpfam, we use a HMM database containing 50 profile families, and a sequence file containing 38192 sequences. This benchmark is referred as data set-2 in following sections.

For comparison of the Pthreads, OpenMP and EARTH version of parallel hmmpfam on SMP machine, we use a HMM database containing 50 profile families, and a sequence file containing 2276 sequences. This benchmark is referred as data set-3 in following sections.

6.4 Comparison With the PVM Implementation

First test is conducted to compare the scalability of the PVM version and the EARTH version on COMET Environment using test data set-1. Figure 6.1

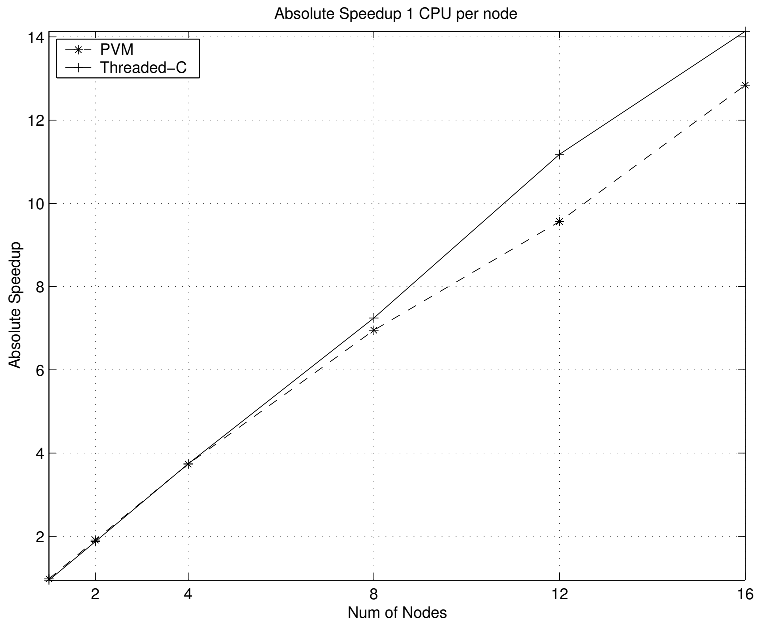

Figure 6.1: The Comparison of PVM Version and EARTH Version on COMET

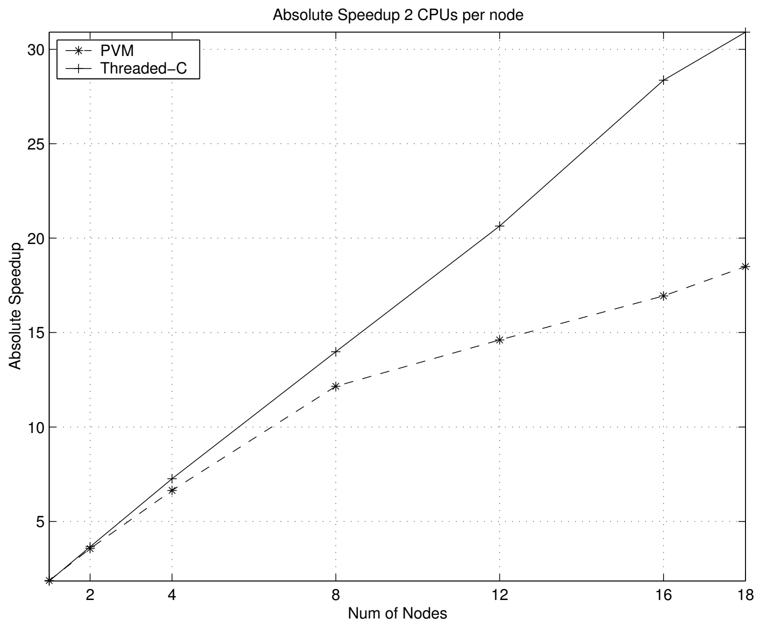

Figure 6.2: The Comparison of PVM Version and EARTH Version on COMET

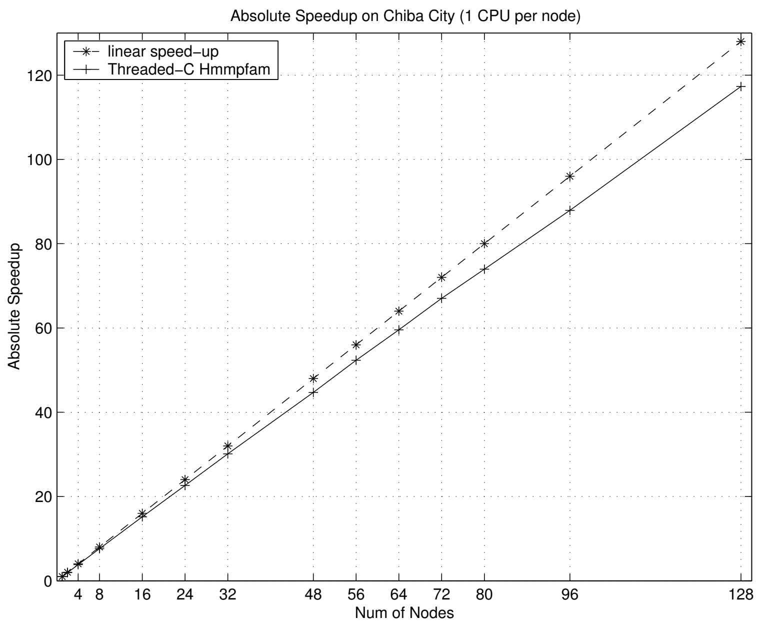

Figure 6.3: Static Load Balancing on Chiba City

shows the absolute speed up curve when both PVM version and EARTH version are configured to use only 1 CPU per node in COMET, while figure 6.2 shows the results for dual CPUs per node configuration. From the figures, it is easily seen that our new version has much better scalability, especially in dual-CPU per node configuration. For example, with 16 nodes and 2 CPUs per node configuration, the absolute speedup of the PVM version is 18.50 while the speedup of our version is 30.91, which means 40% reduction of execution time. This is due to the fact that our implementation increases the computation granularity and avoids most communication cost and internal barrier.

6.5 Scalability on Supercomputing Clusters

The Section 6.4 already shows that the EARTH implementation of hmmpfam is more performance scalable. Now we are concern about whether the speedup of this parallel hmmpfam can keep linearly increasing with the increase of number of processors. To demonstrate the scalability of this hmmpfam version based

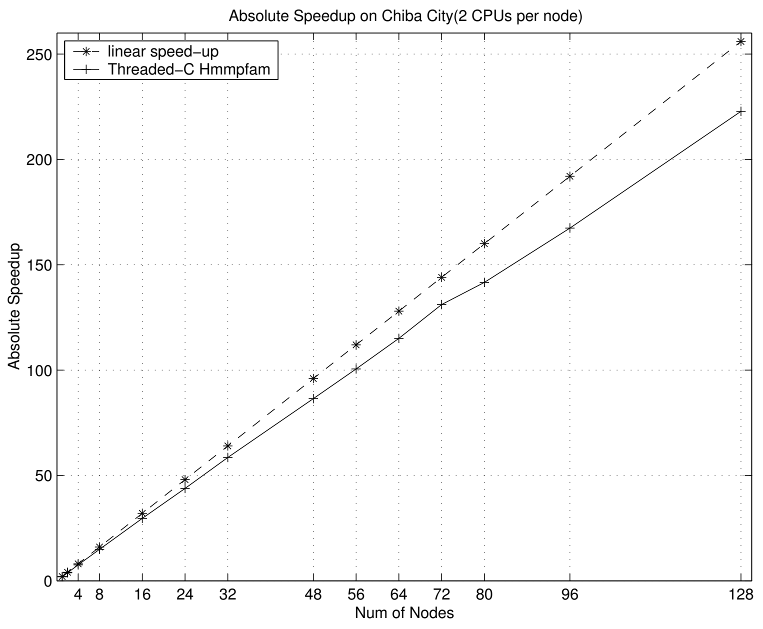

Figure 6.4: Static Load Balancing on Chiba City

on EARTH PXM, the experiments should be done on some large supercomputing clusters, especially on the cluster, which has a high position on the TOP500 [4] supercomputer list. If running on a large number of processors, the speedup is still very close to the linear speedup curve, we can then claim that our approach can achieve scalable performance on current advanced supercomputing platforms.

Thus the second test and the third test are conducted to show the performance of our EARTH version hmmpfam on large clusters – Chiba City cluster and Jazz cluster, using test data set-2. The results of both static load balancing and dynamic load balancing schemes are shown in figure 6.3 to figure 6.6, where figure 6.3 and figure 6.4 show the results for static load balancing on 1 CPU per node and 2 CPUs per node configuration, figure 6.5 and figure 6.6 are the results for dynamic load balancing. The two methods do not have much difference in the absolute speedup. This may due to the fact that subtasks are relatively similar in size, which means static load balancing can also achieve good performance. Both of them show

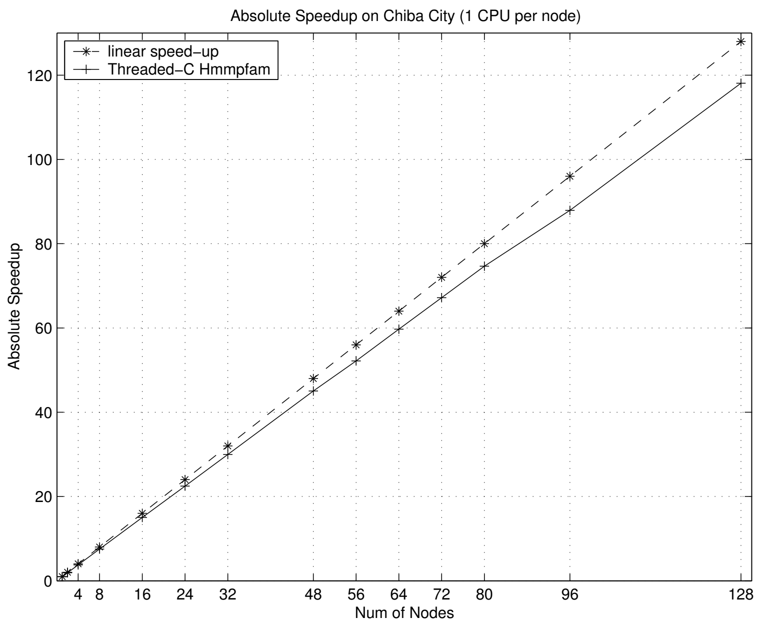

Figure 6.5: Dynamic Load Balancing on Chiba City

near linear speedup, which demonstrates that in our new parallel scheme, the serial part only occupies a very small percentage of the total execution. As long as the test data set is big enough, the speedup is expected to keep near linear up to 128 nodes on Chiba City Cluster. The test results on Jazz cluster is shown in figure 6.7. The speed up curve shows our implementation can still get near linear speed up on 240 nodes, which are with modern INTEL XEON CPUs.

After achieving the scalable performance as just shown, it is worth discussions on the reasons why the performance can be scalable by the new parallel scheme. What is advantages of the new scheme? How the new parallel scheme overcomes the disadvantage of the old scheme used by the PVM implementation? Those questions are going to be visited in the next section.

6.6 Performance Analysis for The Cluster Based Solution

In this subsection, a simple analysis and comparison of the proposed new approach and the PVM approach is presented. To simplify the problem, the following

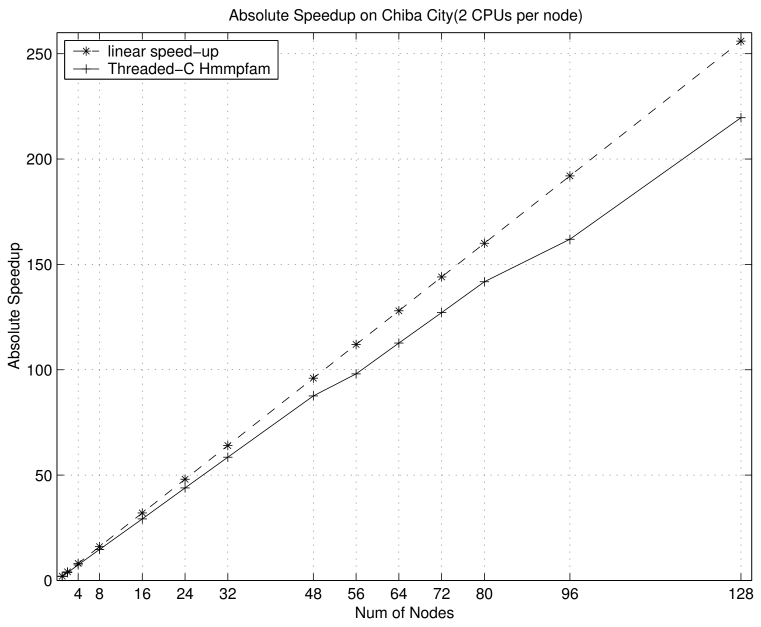

Figure 6.6: Dynamic Load Balancing on Chiba City

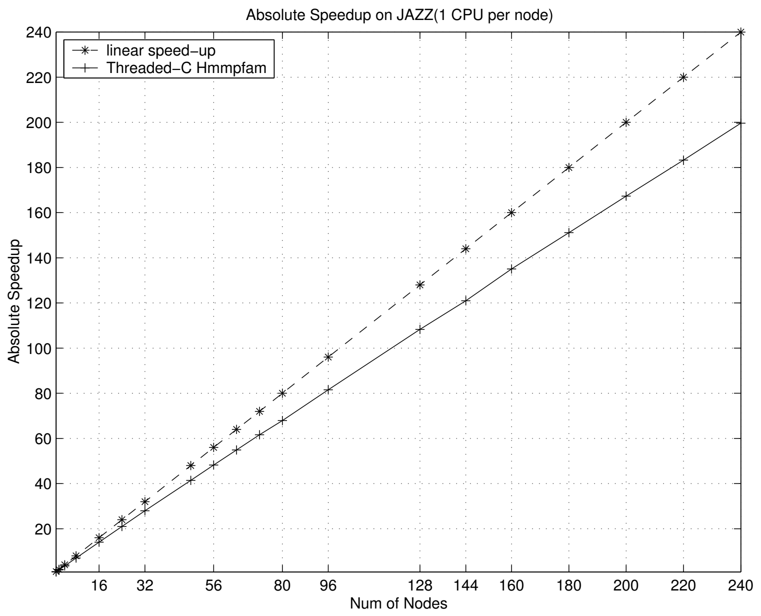

Figure 6.7: Dynamic Load Balancing on JAZZ

assumptions are made:

1. The number of processors is p ;
2. n profiles exist in profile DB and k sequences;
3. The computation of one sequence v.s. one profile takes same amount time, let us denote it as T_0;
4. Denote the communication cost for one round trip communication as T_c;
5. The master node can always respond the requests from slaves concurrently and immediately, and the bandwidth is always sufficient.

For the original PVM approach, the basic task unit is computation of one seq v.s one profile, totally there are $k * n$ such tasks. Each needs T_0 computation time and T_c communication time. Thus, the total work load is:

$$WL = k * n * (T_0 + T_c) \tag{6.1}$$

and the total wall time is

$$T_{total} = k * n * (T_0 + T_c) * (1/p) \tag{6.2}$$

For our new approach, one basic task unit is computation of one seq v.s whole database including n profiles. Totally there are k such tasks. Each task needs $n * T_0$ Computation time and T_c Communication time since only one communication is necessary for one task. Thus, the total work load is

$$WL = k * (n * T_0 + T_c) \tag{6.3}$$

and the total wall time is

$$T_{total} = k * (n * T_0 + T_c) * (1/p) \tag{6.4}$$

From the above analysis, it can be easily seen that the workload saved by our approach is:

$$WL_{save} = k * (n - 1) * T_c \tag{6.5}$$

The wall time saved is

$$T_{save} = k * (n - 1) * Tc * (1/p) \tag{6.6}$$

From the formula 6.6, it can be seen that the larger number of k and n means the larger improvement of our approach. In most cases, the k and n are very big, thus our approach will have significant improvement, and also achieves good scalability as previous sections shows.

Besides the reasons analyzed in the preceding formulas, there are several other factors that contribute to the better performance of out approach. Firstly, when the number of the slave nodes are very large, a lot of requests from the slaves to the master may happen at the same time. Since the master node has to handle the requests one by one and the communication bandwidth of the master node is limited, the assumption 5 may not be valid anymore. That is, the master node can not respond the requests from slaves concurrently and immediately anymore. As mentioned in Section 3.1, the PVM approach regards the computation of one sequence against one profile as a task, and the computation time for this task is very short, so the slave nodes send request to the master very frequently. Our approach regards one sequence against whole database as one task unit and has larger computation time for each task unit, the requests occurs less frequently. Thus, for the PVM approach, the chance of many requests blocked at the master node is much higher than our approach. Secondly, since the computation of ranking and sorting are performed at master node for the PVM approach, during this stage, all the slaves are idle, while in our approach, the ranking and sorting are distributed to the slaves, thus slaves will have less idle time waiting for response from the master node. When ported to large supercomputing clusters, those reasons stated above also stands for the scalability of our approach. As long as there are enough jobs to keep the processors usefully busy, the near linear speedup are expected to be kept.

To conclude, from the simple theoretical analysis, with big k and n, our approach will achieve better scalability; in the real case, our approach may perform even better.

6.7 Robustness Experimental Results

One of the advantages of the dynamic load balancing approach is its robustness. The experiments are conducted to show that the program with dynamic load balancing is less affected by the disturbance. The Blastall [75] program is used as the disturbance source since this program is another commonly used computation intensive bioinformatics software.

The execution time for both static and dynamic approach with and without disturbance is measured. Let T denote the execution time without disturbance, and T' denote the execution time with disturbance. Define the *performance degradation ratio under disturbance* (PDRD) as:

$$PDRD = (\frac{T' - T}{T}) \times 100\% \tag{6.7}$$

The PDRD is computed and plotted for both static and dynamic approaches. A smaller PDRD indicates the performance is less influenced by the introduction of disturbance, thus implies the higher implementation robustness.

For robustness experiment, the data set-1 is used on COMET cluster. Figure 6.8 shows the result when only one Blastall program is running on 1 CPU to disturb the execution of hmmpfam, and the Fig. 6.9 shows the result when two CPUs of one node are both disturbed. Figure 6.10 and Fig. 6.11 show the result when 2 computing nodes are disturbed. From the figures, it is apparent that the dynamic

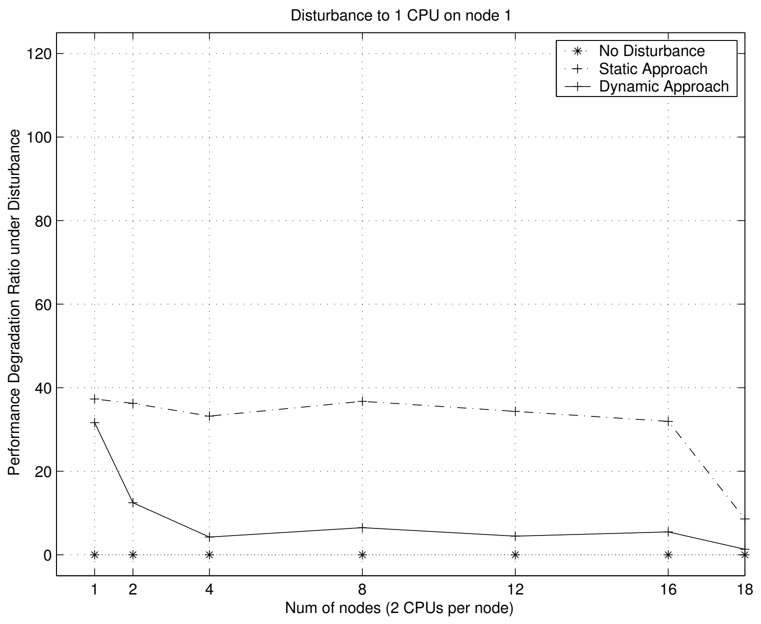

Figure 6.8: Performance Degradation Ratio under Disturbance to 1 CPU

load balancing program is less affected by the disturbance and thus has higher robustness.

6.8 Experiment Results on SMP Platform

The fourth test is conducted to compare the performance of the POSIX Threads version in HMMER package, the OpenMP version, and the EARTH version on the SMP machine – dbi-rna1 by using test data set-3. Figure 6.12 shows the absolute speedup curves of the OpenMP implementations. Three different schedulers – static, dynamic, and guided, are compared. From the figure, the version with dynamic scheduler shows the best performance among all three, and the performance of static scheduler is worse than the other two schedulers.

Figure 6.13 shows the SMP based hmmpfam implementations with different APIs – POSIX Threads, OpenMP and EARTH. For the OpenMP implementation, we choose the version with dynamic scheduler for comparison since dynamic OpenMP version is the best performed one in all three OpenMP implementations.

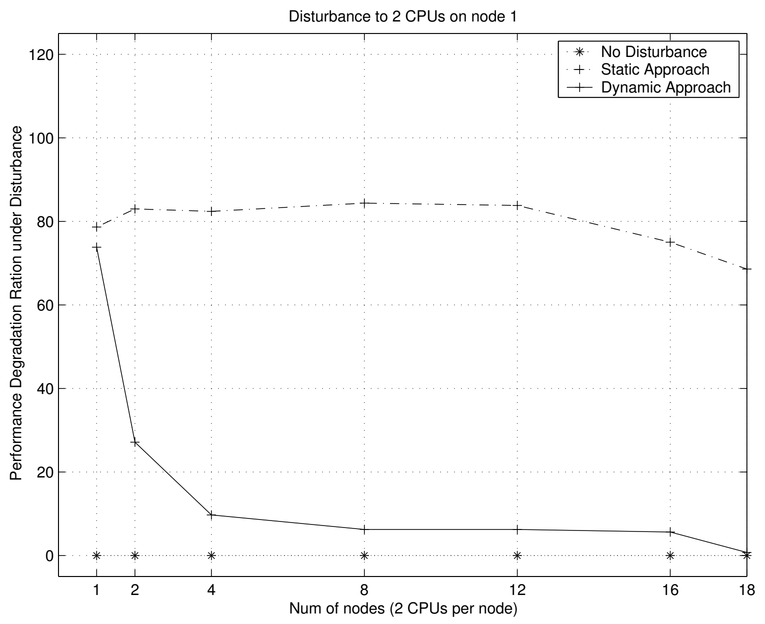

Figure 6.9: Performance Degradation Ratio under Disturbance to 2 CPUs on 1 node

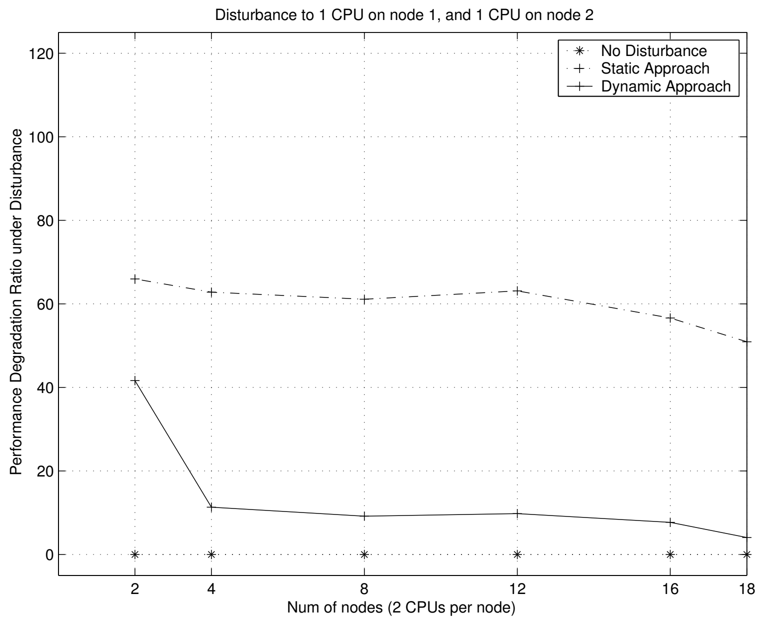

Figure 6.10: Performance Degradation Ratio under Disturbance to 2 CPUs on 2 node

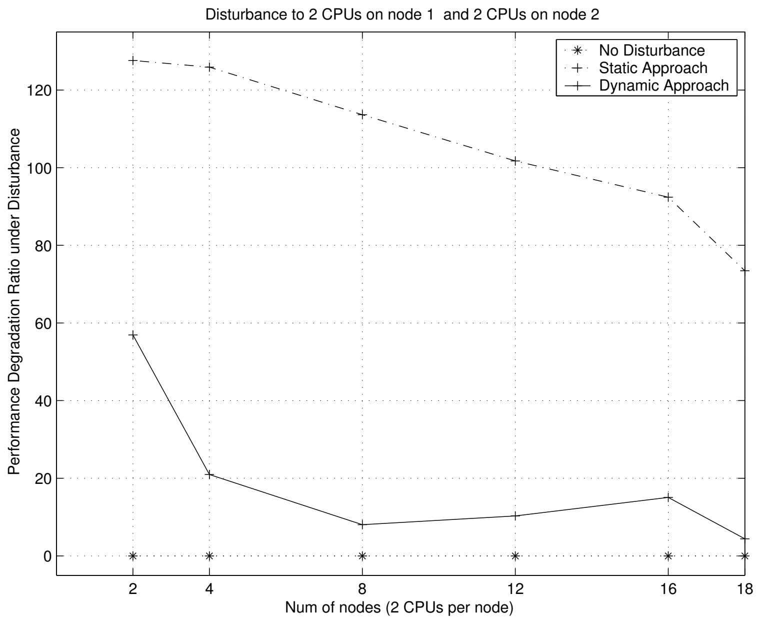

Figure 6.11: Performance Degradation Ratio under Disturbance to 4 CPUs on 2 node

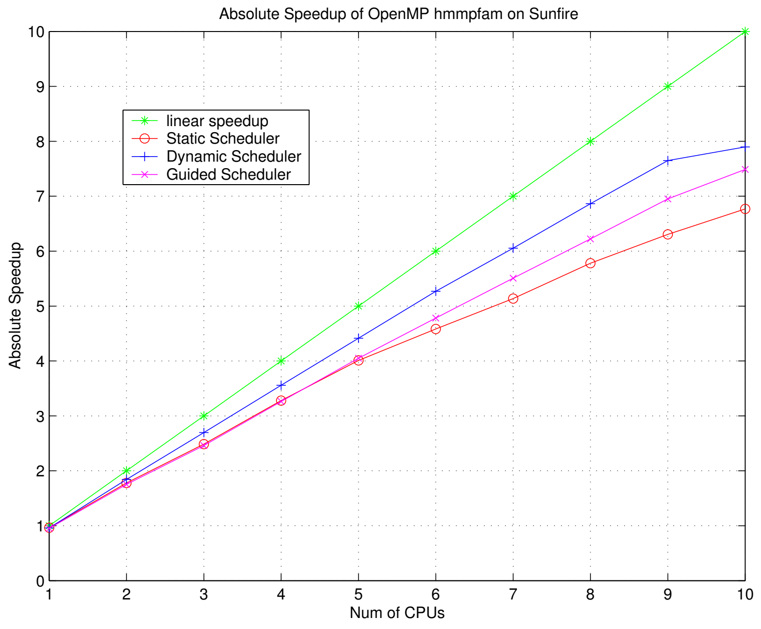

Figure 6.12: Absolute Speedup of OpenMP HMMPFAM Implementations with Different Scheduler

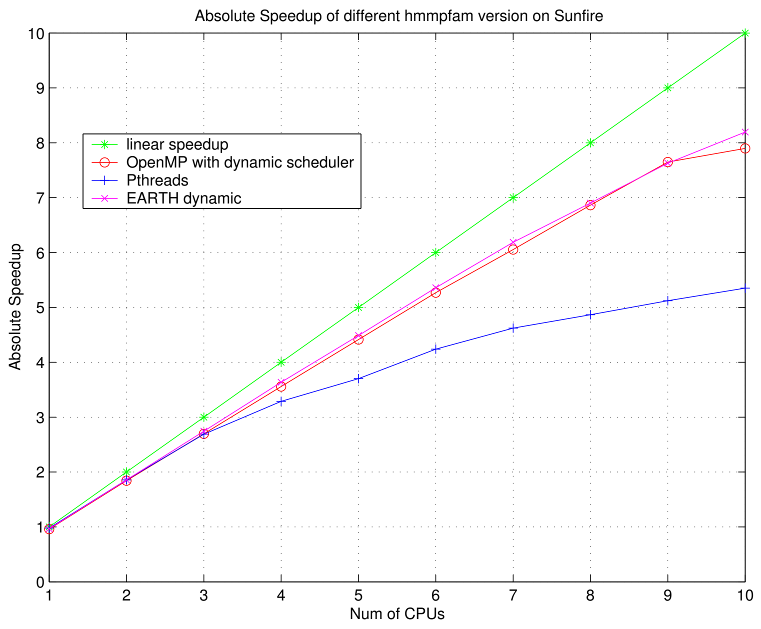

Figure 6.13: Absolute Speedup of SMP based HMMPFAM Implementations with Different APIs

It is obvious that the EARTH and OpenMP version beats the Pthreads version in HMMER package. The EARTH version and OpenMP version gains similar performance, and the EARTH version is a little better. However, the OpenMP can be only used in SMP machine, not applicable to distribute memory environment like clusters, meanwhile, the EARTH version can achieve good performance both on supercomputing clusters and SMP machines.

6.9 Performance Portability of Hmmpfam Based on EARTH PXM

The analysis in Chapter 4 concludes that the EARTH program is both code portable and performance portable to different platforms. Previous sections in this chapter already shows that the same code without modification is able to be executed on uni-processor cluster, SMP cluster and SMP machines. Thus the code portability is already proved for Hmmpfam coded by Threaded-C.

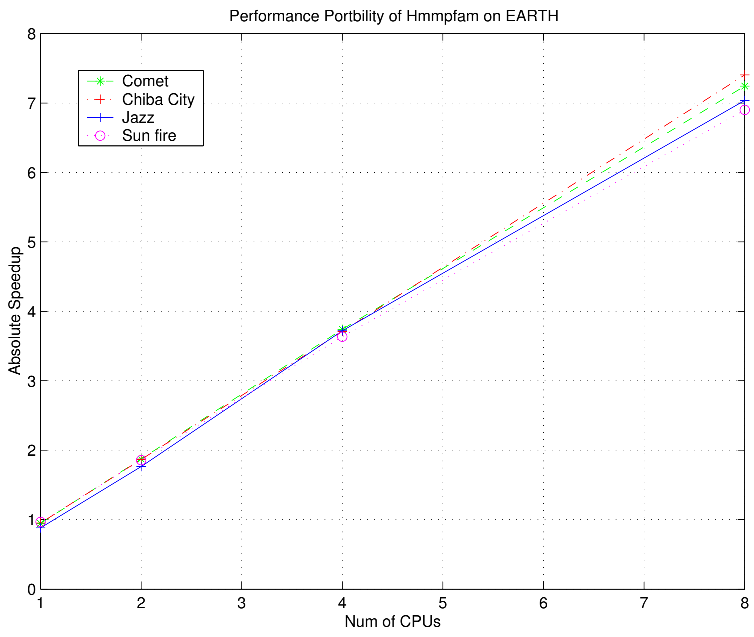

Figure 6.14: Performance Portability of HMMPFAM on EARTH for Different Platforms

For proving the performance portability of Hmmpfam on EARTH, the experimental data collected from previous experiments is reorganized. In order to present an easy view of the results, the speedup curves from different platforms is put into the same figure, as shown by Figure 6.14.

The platforms presented in this figure includes uni-processor cluster (JAZZ), SMP clusters (Chiba City and Comet), and SMP machine (Sun Fire). However, almost the same speedup curves for different platforms shown on this figure, which demonstrates that the performance of Hmmpfam based on EARTH PXM is portable to those platforms supported by EARTH RTS 2.5

Figure 6.15 compares the performance of hmmpfam on uni-processor clusters (JAZZ) and SMP cluster (Chiba City) with large number of CPUs. In Chiba City cluster, each machine contributes two CPUs to the computation, for example, the number of 128 CPUs means 64 dual-CPU machines. This figure shows similar absolute speed curve achieves on these two different platforms.

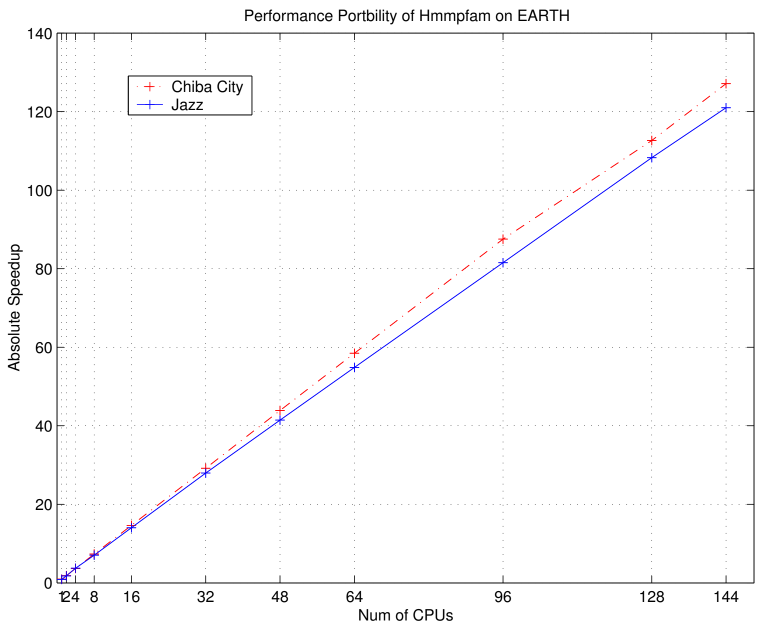

Figure 6.15: Performance Portability of HMMPFAM on EARTH from Uniprocessor Cluster to SMP Cluster

6.10 Conclusions Drawn from Experiments

Compared with the original PVM version of hmmpfam, Section 6.4 shows that our hmmpfam implementation based on EARTH achieves better scalable performance. And our cluster based solution for hmmpfam also presents near linear speedup on two large supercomputing clusters as shown in Section 6.5. The discussions for reasons why our cluster based solution of hmmpfam on EARTH can achieve good scalable performance on supercomputing clusters are presented in Section 6.6. Experiments illustrated in Section 6.7 proves that with the help of newly implemented master-slave dynamic load balancer of EARTH RTS 2.5, the dynamic load balancing approach of parallel hmmpfam performs more robust than the static load balancing approach. Section 6.8 demonstrates SMP based solution of hmmpfam by using different parallel APIs. The EARTH implementation shows as good performance as the OpenMP implementation, and beats the Pthread implementation in HMMER package. Finally, in section 6.9, previous experimental data is reorganized to demonstrate that the performance of hmmpfam implementation based on EARTH is code portable and performance portable to different platforms.

Chapter 7

CONCLUSIONS

We implemented a new cluster-based solution of HMM database searching tool on EARTH (Efficient Architecture for Running Threads) and demonstrated significant performance improvement over the original parallel version based on PVM. Our solution provides near linear scalability on supercomputing clusters. On a cluster of 128 dual-CPU nodes, the execution time of a representative testbench is reduced from 15.9 hours to 4.3 minutes. Comparison between the static and dynamic load balancing approach shows the latter is an more robust and practical solution for large time consuming applications running on clusters.

Without the modification of the code, on the SMP machine, the same EARTH hmmpfam can also achieves the same and even better performance compared with the OpenMP implementation, which is one the most popular APIs for parallalizing applications on SMP machine. Experimental results shows that the EARTH version of hmmpfam performs much better than the Pthreads version in HMMER packages on SMP machines. This approves that the parallel hmmpfam based on EARTH PXM can achieves good performance not only on supercomputing clusters, but also on SMP servers, and SMP clusters. An important advantage of EARTH is that the programmer does not need to know the platform details but still achieve ideal performance on all supported platforms. From the implementation of Hmmpfam, we shows that the application based on EARTH is both code portable and performance portable to different platforms.

This new implementation could allow researchers to analyze biological sequences at a much higher speed and also make it possible for scientists to analyze problems that were previously considered too large and too time consuming. We presented and evaluated two different parallel schemes which are targeted to advanced supercomputing clusters. One is based on the round-robin algorithm; the other is based on the dynamic load balancing mechanism. The parallelization implementation in this paper motivated the addition of robust dynamic load balancing support into EARTH model, which proves that applications could be the driving force for design of architecture and programming model.

Sequence family classification and HMM database searching is very important to the biological research community. With the help of supercomputing resources and cluster-based solution for applications, researchers can now save research time and get new discoveries more quickly than ever. Many other bioinformatics applications are very time consuming and in essence embarrassingly parallel, which makes them very suitable applications for cluster computing. Porting of hmmpfam to EARTH model provides very promising results, thus further research includes the porting of other bioinformatics applications such as multiple sequence alignment, phylogenetic tree generation to EARTH platform.

BIBLIOGRAPHY

[1] HMMER: sequence analysis using profile hidden Markov models. [Online]. Available: http://hmmer.wustl.edu/

[2] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM: Parallel Virtual Machine – A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press, 1994.

[3] T. Sterling, D. Becker, and D. Savarese, "BEOWULF: A parallel workstation for scientific computation," *Proceedings of the 1995 International Conference on Parallel Processing (ICPP)*, pp. 11–14, 1995.

[4] The 22nd TOP500 Supercomputer List for November 2003. [Online]. Available: http://www.top500.org/list/2003/11/

[5] J. T. L. Wang, Q. heng Ma, and C. H. Wu, "Application of neural network to biological data mining: A case study in protein sequence classification," *Proceedings of KDD-2000*, pp. 305–309, 2000.

[6] S. Eddy, "Profile hidden markov models," *Bioinformatics*, vol. 14, pp. 755–763, 1998.

[7] S. Levinson, L. Rabiner, and M. Sondhi, "An introduction to the application of the theory of probabilistic functions of a markov process to automatic speech recognition," *Bell Syst.Tech.J.*, vol. 62, pp. 1035–1074, 1983.

[8] P. Baldi and S. Brunak, *Bioinformatics: The machine learning approach.* Cambridge, Massachusetts: The MIT press, 2001.

[9] A. Bateman, E. Birney, L. Cerruti, R. Durbin, L. Etwiller, S.R. Eddy, S. Griffiths-Jones, K.L. Howe, M. Marshall, and E.L.L. Sonnhammer, "The pfam protein families database," *Nucleic Acids Research*, vol. 30, no. 1, pp. 276–280, 2002.

[10] E.L.L Sonnhameer, S.R. Eddy, and R. Durbin, "Pfam: A comprehensive database of protein domain families based on seed alignments," *Proteins*, vol. 28, pp. 405–420, 1997.

[11] E.L.L Sonnhameer, S.R. Eddy, E. Birney, A. Bateman, and R. Durbin, “Pfam: Multiple sequence alignments and hmm-profiles of protein domains,” *Nucl. Acids Res.*, vol. 26, pp. 320–322, 1998.

[12] K. B. Theobald, “EARTH: An efficient architecture for running threads,” Ph.D. dissertation, May 1999.

[13] H. H. J. Hum, O. Maquelin, K. B. Theobald, X. Tian, X. Tang, and G. R. G. et al., “A design study of the EARTH multiprocessor,” *Proceedings of the Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 59–68, 1995.

[14] N. Luscombe, D. Greenbaum, and M. Gerstein, “What is bioinformatics? a proposed definition and overview of the field,” *Methods Inf Med*, vol. 40, 2001.

[15] B. D. et al, “Genbank,” *Nucleic Acids Res*, vol. 28, no. 1, pp. 15–18, 2000.

[16] H. Berman, J. Westbrook, Z. Feng, G. Gilliland, T. Bhat, H. Weissig, I. Shindyalov, and P. Bourne, “The protein data bank,” *Nucleic Acids Research*, vol. 28, pp. 235–242, 2000.

[17] Genomics Glossaries & Taxonomies: In Silico & molecular modeling glossary. [Online]. Available: http://www.genomicglossaries.com/content/molecular_modeling_gloss.asp

[18] A. Krogh, M. Brown, I. S. Mian, K. Sjolander, and D. Haussler, “Hidden markov models in computational biology: Applications to protein modeling,” *Journal of Molecular Biology*, vol. 235, pp. 1501–1531, 1994.

[19] M. Gribskov, A. D. McLachlan, and D. Eisenberg, “Profile analysis: Detection of distantly related proteins,” *Proc. Natl. Acad. Sci.*, vol. 84, pp. 4355–4358, 1987.

[20] M. Gribskov, R. Luthy, and D. Eisenberg, “Profile analysis,” *Meth. Enzymol.*, vol. 183, pp. 146–159, 1990.

[21] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman, “Basic local alignment search tool,” *Journal of Mol. Biol.*, vol. 215, pp. 403–410, 1990.

[22] Pfam :Home (Saint Louis). [Online]. Available: http://pfam.wustl.edu

[23] K. B. Theobald, J. N. Amaral, G. Heber, O. Maquelin, X. Tang, and G. R. Gao, *Overview of the Threaded-C language*, capsl techincal memo 19 ed., Department of Electrical and Computer Engineering, University of Delaware, Newark, Delaware 19716, March 1998. [Online]. Available: ftp://ftp.capsl.udel.edu/pub/doc/memos

[24] K. B. Theobald, C. J. Morrone, M. D. Butala, J. N. Amaral, and G. R. Gao, *Threaded-C language reference manual (release 2.0)*, capsl techincal memo 39 ed., Department of Electrical and Computer Engineering, University of Delaware, Newark, Delaware 19716, September 2000. [Online]. Available: ftp://ftp.capsl.udel.edu/pub/doc/memos

[25] P. Kakulavarapu, O. Maquelin, and G. R. Gao, *Design of the runtime system for the Portable Threaded-C language*, capsl techincal memo 24 ed., Department of Electrical and Computer Engineering, University of Delaware, Newark, Delaware 19716, July 1998. [Online]. Available: ftp://ftp.capsl.udel.edu/pub/doc/memos

[26] C. J. Morrone, "An EARTH runtime system for multi-processor/multi-node Beowulf clusters," Master's thesis, Univ. of Delaware, Newark, DE, May 2001.

[27] C. Shen, "A portable runtime system and its derivation for the hardware SU implementation," Master's thesis, Univ. of Delaware, Newark, DE, December 2003.

[28] P. B. Hansen, "Concurrent programming concepts," *ACM Computing Surveys*, vol. 5, pp. 233–245, Dec. 1972.

[29] L. Lamport, "Concurrent reading and writing," *Communications of the ACM*, vol. 20, no. 11, pp. 806–811, Nov. 1977.

[30] H. Cai, O. Maquelin, P. Kakulavarapu, and G. R. Gao, "Design and evaluation of dynamic load balancing schemes under a fine-grain multithreaded execution model," Orlando, Florida, January 1999.

[31] H. Cai, "Dynamic load balancing on the earth-sp system," Master's thesis, Univ. of McGill, Montreal, May 1997.

[32] K. P. Kakulavarapu, "Dynamic load balancing issues in the earth runtime systems," Master's thesis, Univ. of Delaware, Newark, DE, Dec 1999.

[33] H. H.-J. Hum, "The super-actor machine: a hybrid dataflow/von Neumann architecture," Ph.D. dissertation, May 1992.

[34] D. R. Butenhof, *Programming with POSIX(R) Threads.* Addison-Wesley Pub. Co., 1997.

[35] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* Cambrdge, MA: MIT Press, Oct. 1994.

[36] P. Pacheco, *Parallel Programming with MPI.* San Francisco, CA: Morgan Kaufmann, 1997, http://www.usfca.edu/mpi (source programs available) and http://www.mkp.com.

[37] R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan, and J. McDonald, *Parallel Programming in OpenMP.* Morgan Kaufmann, 2000.

[38] W. D. Gropp and E. Lusk, *User's Guide for MPICH, a Portable Implementation of MPI*, Mathematics and Computer Science Division, Argonne National Laboratory, 1996, aNL-96/6.

[39] H. Tang and T. Yang, "Optimizing threaded MPI execution on SMP clusters," in *Proceedings of the 15th ACM International Conference on Supercomputing (ICS-01).* New York: ACM Press, June 17–21 2001, pp. 381–392.

[40] H. Tang, K. Shen, and T. Yang, "Compile/run-time support for threaded MPI execution on multiprogrammed shared memory machines," in *Proceedings of the 1999 ACM Sigplan Symposium on Principles and Practise of Parallel Programming (PPoPP'99)*, ser. ACM Sigplan Notices, A. A. Chien and M. Snir, Eds., vol. 34.8. A.Y.: ACM Press, May 4–6 1999, pp. 107–118.

[41] S. Sistare, R. vandeVaart, and E. Loh, "Optimization of MPI collectives on clusters of large-scale SMPs," in *SC'99: Oregon Convention Center 777 NE Martin Luther King Jr. Boulevard, Portland, Oregon, November 11–18, 1999*, ACM, Ed. New York, NY 10036, USA and 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA: ACM Press and IEEE Computer Society Press, 1999.

[42] T. Takahashi, F. O'Carroll, H. Tezuka, A. Hori, S. Sumimoto, H. Harada, Y. Ishikawa, and P. Beckman, "Implementation and evaluation of MPI on an SMP cluster," in *Parallel and Distributed Processing – IPPS/SPDP'99 Workshops*, ser. Lecture Notes in Computer Science, vol. 1586. Springer-Verlag, Apr. 1999.

[43] E. D. Demaine, "A threads-only MPI implementation for the development of parallel programs," in *the 11th International Symposium on High Performance Computing Systems (HPCS'97)*, Winnipeg, Manitoba, Canada, July 10-12 1997, pp. 153–163.

[44] TOMPI, a Threads-Only MPI Implementation. [Online]. Available: http://theory.lcs.mit.edu/ edemaine/TOMPI/

[45] C. Santos and J. Aude, "PM-PVM: A portable multithreaded PVM," in *13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, San Juan, Puerto Rico, April 12-16 1999.

[46] H. Zhou and A. Geist, "LPVM: a step towards multithread PVM," *Concurrency: Practice and Experience*, vol. 10, no. 5, pp. 407–416, Apr. 25 1998.

[47] A. Ferrari and V. Sunderam, "Multiparadigm Distributed Computing with TPVM," *Concurrency - Practice and Experience*, vol. 10, no. 3, pp. 199–228, Mar. 1998.

[48] A. J. Ferrari and V. S. Sunderam, "TPVM: distributed concurrent computing with lightweight processes," in *Proceedings of the Fourth IEEE International Symposium on High Performance Distributed Computing, August 2–4, 1995, Washington, DC, USA*, IEEE, Ed. 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA: IEEE Computer Society Press, 1995, pp. 211–218.

[49] H. Lu, Y. C. Hu, and W. Zwaenepoel, "OpenMP on network of workstations," in *Proc. of Supercomputing'98*, Oct. 1998. [Online]. Available: http://www.cs.rice.edu/˜willy/papers/sc98.ps.gz

[50] A. L. Cox, Y. C. Hu, H. Lu, and W. Zwaenepoel, "OpenMP for networks of SMPs," in *Proc. of the Second Merged Symp. IPPS/SPDP 1999)*, Apr. 1999. [Online]. Available: http://www.cs.rice.edu/˜willy/papers/ipps99.ps.gz

[51] Y.-S. Kee, J.-S. Kim, and S. Ha, "ParADE: An OpenMP programming environment for SMP cluster systems," in *SC2003: Igniting Innovation. Phoenix, AZ, November 15–21, 2003*, ACM, Ed. New York, NY 10036, USA and 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA: ACM Press and IEEE Computer Society Press, 2003.

[52] M. Sato, H. Harada, A. Hasegawa, and Y. Ishikawa, "Cluster-enabled OpenMP: An OpenMP compiler for the SCASH software distributed shared memory system," *Scientific Programming*, vol. 9, no. 2-3, pp. 123–130, 2001.

[53] Y. Ojima, M. Sato, H. Harada, and Y. Ishikawa, "Performance of cluster-enabled OpenMP for the SCASH software distributed shared memory system," in *Proc. of the 3rd IEEE/ACM Int'l Symp. on Cluster Computing and the Grid (CCGrid'03*, May 2003, pp. 450–456.

[54] M. Sato, H. Harada, A. Hasegawa, and Y. Ishikawa, "Cluster-enabled OpenMP: an OpenMP compiler for software distributed shared memory system SCASH," in *JSPP'01.* Information Processing Society of Japan, June 2001, pp. 15–22.

[55] H. Löf, Z. Radovic, and E. Hagersten, "THROOM — running POSIX multithreaded binaries on a cluster," Department of Information Technology, Uppsala University, Tech. Rep. 2003-026, Apr. 2003.

[56] P. Jamieson and A. Bilas, “CableS: Thread control and memory system extensions for shared virtual memory clusters,” *Lecture Notes in Computer Science*, vol. 2104, 2001. [Online]. Available: http://link.springer-ny.com/link/service/series/0558/bibs/2104/21040170.htm; http://link.springer-ny.com/link/service/series/0558/papers/2104/21040170.pdf

[57] L. Smith and P. Kent, “Development and performance of a mixed OpenMP/ MPI quantum Monte Carlo code,” *Concurrency: Practice and Experience*, vol. 12, no. 12, pp. 1121–1129, Oct. 2000.

[58] L. Smith and M. Bull, “Development of mixed mode MPI / openMP applications,” *Scientific Programming*, vol. 9, no. 2-3, pp. 83–98, 2001, ePCC.

[59] F. Cappello and D. Etiemble, “MPI versus MPI+openMP on IBM SP for the NAS benchmarks,” in *Proceedings of Supercomputing’2000 (CD-ROM).* Dallas, TX: IEEE and ACM SIGARCH, Nov. 2000, lRI.

[60] F. Cappello, O. Richard, and D. Etiemble, “Performance of the NAS benchmarks on a cluster of SMP PCs using a parallelization of the MPI programs with OpenMP,” *Lecture Notes in Computer Science*, vol. 1662, 1999.

[61] G. Jost, H. Jin, D. an Mey, and F. F. Hatay, “Comparing the OpenMP, MPI, and hybrid programming paradigms on an SMP cluster,” in *the Fifth European Workshop on OpenMP (EWOMP03)*, Aachen, Germany, Sepetember 2003.

[62] R. Rebenseifner, “Hybrid parallel programming: Performance problems and chances,” in *the 45th Cray User Group Conference*, Ohio, May 12-16 2003.

[63] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren, “Introduction to upc and language specification,” *CCS-TR-99-157*, May 13 1999.

[64] T. El-Ghazawi and S. Chauvin, “UPC benchmarking issues,” in *Proceedings of 2001 International Conference on Parallel Processing (30th ICPP’01).* Valencia, Spain: Universidad Politecnia de Valencia, Sept. 2001.

[65] T. El-Ghazawi and F. Cantonnet, “UPC performance and potential: A NPB experimental study,” in *SC’2002 Conference CD.* Baltimore, MD: IEEE/ACM SIGARCH, Nov. 2002, pap316.

[66] R. W. Numrich and J. Reid, “Co-Array Fortran for parallel programming,” *ACM SIGPLAN FORTRAN Forum*, vol. 17, no. 2, pp. 1–31, Aug. 1998.

[67] P. Bała and T. W. Clark, "Pfortran and Co-Array Fortran as tools for parallelization of a large-scale scientific application," *Lecture Notes in Computer Science*, vol. 1900, 2001. [Online]. Available: http://link.springer-ny.com/link/service/series/0558/bibs/1900/19000511.htm; http://link.springer-ny.com/link/service/series/0558/papers/1900/19000511.pdf

[68] J. Reid, "Co-array Fortran for full and sparse matrices," *Lecture Notes in Computer Science*, vol. 2367, 2002. [Online]. Available: http://link.springer-ny.com/link/service/series/0558/bibs/2367/23670061.htm; http://link.springer-ny.com/link/service/series/0558/papers/2367/23670061.pdf

[69] Y. Sun and D. Bader, "Broadcast on clusters of SMPs with optimal concurrency," University of New Mexico, Albuquerque, NM, Tech. Rep. AHPCC Technical Report 2000-013, June 2000.

[70] R. Samanta, A. Bilas, L. Iftode, and J. P. Singh, "Home-based SVM protocols for SMP clusters: Design and performance," in *Proc. Of IEEE 4th International Symposium on High-Performance Computer Architecture (HPCA98)*, 1998.

[71] W. E. Johnston, "High-speed, wide area, data intensive computing: a ten year retrospective," in *Proceedings of IEEE International Symposium on High-performance Distributed Computing, pp.280-291*, 1998, pp. 280–291.

[72] C. Li, "EARTH-SMP: Multithreading support on an SMP cluster," Master's thesis, Univ. of Delaware, Newark, DE, Apr. 1999.

[73] The Argonne Scalable Cluster. [Online]. Available: http://www-unix.mcs.anl.gov/chiba/

[74] The Argonne JAZZ Cluster, Laboratory Computing Resource Center (LCRC). [Online]. Available: http://www.lcrc.anl.gov/jazz/

[75] NCBI blast. [Online]. Available: http://www.ncbi.nlm.nih.gov/BLAST/