MESSAGE-DRIVEN PARALLEL LANGUAGE RUNTIME DESIGN AND OPTIMIZATIONS FOR MULTICORE-BASED MASSIVELY PARALLEL MACHINES

BY

CHAO MEI

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2012

Urbana, Illinois

Doctoral Committee:

Professor Laxmikant V. Kalé, Chair
Professor David A. Padua
Professor Josep Torrellas
Dr. Pavan Balaji, Argonne National Laboratory

Abstract

Multicore chips have become the standard building blocks for all current and future massively parallel machines. Much work has been done in scientific and engineering HPC applications to exploit shared-memory multicore nodes. This thesis, in contrast, pays close attention to the parallel language runtime system–a software layer that supports the execution of parallel applications. The essential idea is to parallelize the language runtime with threads as a natural consequence of the same general approach in applications to take advantage of the shared memory on a multicore node. Using the asynchronous message-driven CHARM++ runtime system as an evaluation platform, we address the key question of how the runtime should be designed and how it can be optimized for multicore nodes on parallel machines so that applications running atop the runtime can achieve better performance with as few changes as possible.

Since the runtime performance on a single node is the basis for the overall runtime performance at scale, we have identified key factors for the runtime to run well on a single node, and developed corresponding optimization techniques. We have also developed the *CkLoop* library in the CHARM++ runtime, which showcases the necessity of a unified runtime that can make better support of the parallelism at different granularity.

Furthermore, we have explored the design space of work responsibility assignment among the threads in the multithreaded runtime. In the context of a runtime design of dedicated communication threads, we have investigated the consequent communication issues with the help from our extension to a performance analysis tool, and proposed methods that can resolve the issues. To achieve even better performance in applications, we have shown how developers can leverage new capabilities offered by the runtime, and developed new load balancing strategies that are more effective on multicore platforms.

Finally, we have demonstrated the performance improvement on real production-level scientific applications, including NAMD, a widely-used molecular dynamics simulation program, by using this multithreaded runtime on petascale massively parallel machines. In the case of the 100M-atom STMV simulation using NAMD, the multithreaded runtime leads NAMD to achieve about two-fold performance improvement on 224,076 cores of JaguarPF (Cray XT5), and about three times improvement in machine utilization on Intrepid (BlueGene/P). It also makes NAMD more scalable up to the full machine of JaguarPF and Titan (Cray XK6).

To my parents Jiwu Mei & Shizhen Tang,

and to my wife Yan Zhou,

for their love and support.

Acknowledgments

I would like to thank my advisor and mentor, Professor Laxmikant V. Kalé, for his support throughout my graduate studies and for giving me wonderful opportunities to work on exciting projects. His guidance and inspiration have been invaluable not only for my education, but also for my everyday life. I would also like to extend my gratitude to the rest of my committee Prof. David Padua, Prof. Joseph Torrellas and Dr. Pavan Balaji for their insightful feedbacks and constructive help.

My sincere thanks go out to my current and former colleagues who have made the years at Parallel Programming Laboratory a truly cheerful and memorable experience. Specially, I would like to thank Dr. Gengbin Zheng for helping me on numerous implementation issues and providing his valuable experience in the graduate school as a foreign student, Dr. Sameer Kumar for the offer of an internship at IBM T.J. Watson Research which significantly helped me shape the direction of my thesis. I am very grateful, in no certain order, to Eric Bohm, Dr. Celso Mendes, Yanhua Sun, David Kunzman, Aaron Becker, Josephine Geigner, Jonathan Lifflander, Phil Miller, Pritish Jetley and Lukasz Wesolowski for their reviews of my thesis work.

It was also a great pleasure to work with people, especially Dr. Jim Phillips, from the Theoretical and Computational Biophysics Group to learn much about the molecular dynamics and the widely-used scalable simulation program–NAMD.

Finally, I would like to thank my parents for all their love and encouragement along these years when I am far away from home to pursue this doctorate. I am also much indebted to my wife Yan Zhou and my great friends, Hanlin Ouyang in particular, for their unwavering support during the ups and downs of my graduate student life.

Table of Contents

List of Tables

List of Figures

1 Introduction

In this chapter, the motivation of this work is presented first before summarizing the issues that are investigated in this thesis. Afterwards, the contribution of this work will be listed and a brief description of the thesis organization will be given.

1.1 Motivation

Since the wide adoption of multicore chips by the industry, clusters or supercomputers built on these chips of various architectures have become the most popular option for HPC systems. According to the Top500 list of June 2011[1], more than 92.4% of those parallel machines have 8 cores or more per node. The percentage increases to 98.5% if we just look at the parallel machines deployed in 2010 or later. It is clear enough that near future massively parallel machines will be built from multicore nodes, and each node will consist of tens of cores. For example, BlueWaters, a heterogeneous Cray XK6 sustainable petascale supercomputer hosted in the University of Illinois at Urbana-Champaign, has 32 CPU cores per CPU-only node and 16 CPU cores per CPU-GPU-hybrid node. BlueGene/Q, the latest generation of BlueGene supercomputers, is installed with 16 cores per node but can support up to 64 logical CPUs (hardware threads) per node. These multicore-based highly parallel systems raise questions about parallel software development, especially how to write or tune parallel applications that will run efficiently on these platforms.

Many scientific and engineering applications, including the main consumers of the machine hours on these massively parallel machines, are written in MPI [1], and MPI continues to be the major programming option. Therefore, much work has been done to optimize MPI performance for multicore architectures. In particular, to reduce the message latency

[1]http://top500.org/lists/2011/06

within a node, approaches such as posix-shared memory [2] and OS-kernel-assisted direct copy [3, 4, 5] have been explored to exploit the shared physical memory among cores within a node. As a result, both legacy applications and newly developed ones automatically enjoy performance benefits gained from multicore-based nodes. Given that each MPI task is an OS process, this type of work essentially tries to deliver efficient inter-OS-process data exchanges on a multicore node. However, it is more natural and efficient to do data exchange via threads instead of OS processes because data owned by threads are in the same memory address space. Furthermore, sharing read-only data, which is a common optimization for shared memory, is made difficult due to the disjoint memory address spaces of MPI tasks. Additionally, other characteristics of multicore chips, such as resource sharing (e.g., multiple cores sharing L2/L3 cache), are generally ignored in this kind of work, thus missing performance optimization opportunities that could be used to further utilize multicore chips.

Additionally, much effort has been spent on combining shared-memory programming options, such as OpenMP [6], Pthread [7], Thread Building Block (TBB) [8] etc., with MPI to form a hybrid programming approach. Because of the simple expressiveness of OpenMP and compiler support, MPI+OpenMP is the most popular approach [9, 10]. OpenMP is used to access the shared memory of a node to avoid intra-node communication and parallelize computation-intensive loops, while MPI is used for inter-node communication. Such a hybrid approach to take advantage of multicore-based parallel machines is also beneficial because of the large memory footprint of pure MPI programs running on parallel machines that have a relatively small amount of memory per core. For example, Hopper at NERSC is installed with 24 cores per node, but every node only has 32GB of memory. On those machines, some pure MPI programs may run out of memory if every core of the node hosts a MPI task. MPI+OpenMP allows application developers to overcome the memory issue and utilize all the cores of a node at the same time. However, although hybrid programming is popular, it delivers mixed performance results [11]: some applications always perform best with pure MPI codes, while others perform best with a certain combination of MPI tasks and OpenMP threads on a multicore node. When launching a hybrid program, spare physical cores without hosting any MPI ranks are allocated in advance for the computation parallelized using OpenMP. However, if only a part of the computation in the application can be parallelized with OpenMP, then those spare cores are going to be wasted when the other part of computation is processed. Such waste may, therefore, lead to an overall worse performance. Moreover, to achieve better performance in the hybrid approach, it may require a significant amount of programming effort and shared-memory programming experience [10]. As a result, it is better to have an approach to exploit the multicore nodes with as little increase in the programming complexity as possible.

In addition to the large amount of work done related with MPI and its applications, work on other parallel programming languages has flourished as well. They also must address the programming challenge on these massively parallel machines, as it is projected that there could be millions of cores in a machine in the near future. For example, Partitioned Global Address Space (PGAS) languages, such as Unified Parallel C (UPC) [12], are emerging alternatives that allow shared memory-like programming on those multicore-based parallel machines. Furthermore, more interests are accruing on parallel programming languages, such as ParalleX [13], CnC [14], CHARM++ [15] etc., that use a data-driven or message-driven execution model. Such an execution model is considered to be more advantageous than the traditional message passing model represented by MPI with regard to exploiting the full computation power entailed by machines that have hundreds of thousands of cores or more [16]. For one of those languages to be accepted for a wide adoption, its runtime should first be able to run across different parallel machine platforms. More importantly, its runtime should exhibit high-performance, in which case, it is better to exploit multicore nodes. Additionally, as applications become more dynamic and complex in order to take advantage of the increasing computing capability, dynamic load balancing becomes more important. Therefore, load balancing strategies adopted by the runtime should consider the characteristics of multicore chips.

Considering all the relevant work to exploit the multicore nodes in the HPC community, this thesis studies ***the design and optimization techniques from the perspective of the language runtime to exploit the multicore nodes on massively parallel machines with***

minimal increase in programming complexity. We take the approach to parallelize the runtime system with threads as a natural consequence of the same general approach to optimize applications for a multicore node. It is straightforward that this approach is expected to provide better communication within a node because the message can now be delivered via a message pointer in the user space without any OS kernel involvement, and provide the capability of sharing certain data structures thanks to the single memory address space on a node.

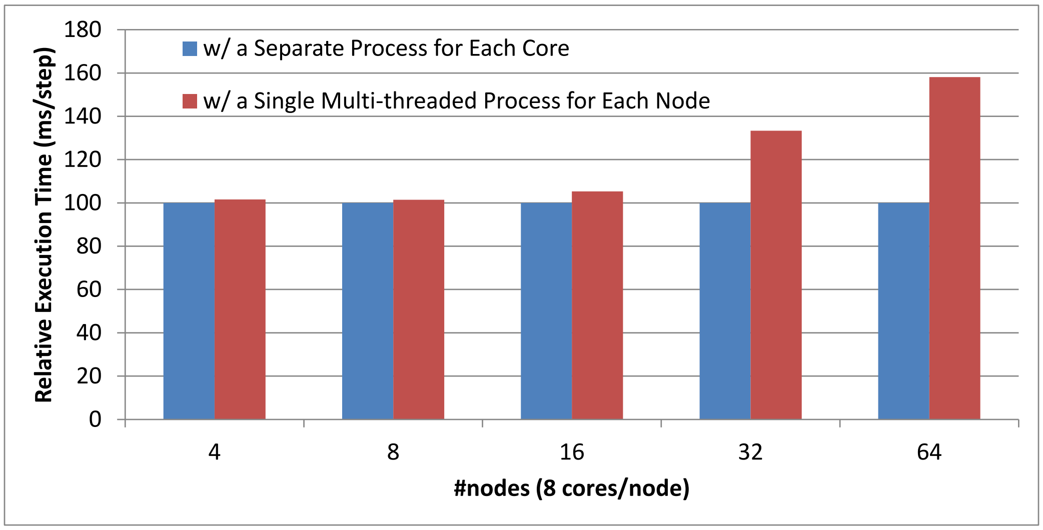

Figure 1.1: Performance Results of NAMD on Abe After the Earlier Work in CHARM++ to Exploit Multicore

Earlier versions of CHARM++ supported running in a mode that adopts the above approach as launching one process for each node and *pthread* threads for each core. However, just as in many experiments with MPI and OpenMP where the MPI everywhere performs better initially, the mode with a separate process for each core was typically faster. As one of the examples, consider the performance of NAMD [17], a widely used molecular dynamics simulation application written in CHARM++. We ran NAMD on Abe[2] with standard benchmark input Apoa1. In Figure 1.1, the execution time of NAMD with two versions is shown (normalized to the version with separate processes per core). Clearly, NAMD running with CHARM++ in a single-process-per-node mode performs worse and

[2]A cluster at NCSA, which retired in early of 2011:
http://www.ncsa.illinois.edu/UserInfo/Resources/Hardware/Intel64Cluster/

becomes much worse when the number of nodes increases.

Strongly motivated by the performance problem in the earlier implementation of the multithreaded runtime approach in CHARM++, this thesis explores relevant issues in the design and optimization of a user-transparent multithreaded message-driven parallel language runtime system. Built on the ealier work in CHARM++, this research uses CHARM++ as a platform for experimentation to study the performance issues and investigate corresponding solutions. We also develop benchmarks and use real scientific applications, such as NAMD, to assess the effectiveness of this multithreaded runtime and the related optimization techniques on multiple massively parallel machines, including those in the list of top ten world's fastest supercomputers.

Exploiting multicore nodes in the runtime can capture some performance optimization opportunities that are not exposed at the programming stage, hence it is complementary to efforts by application developers. Furthermore, through performance investigation in the CHARM++ runtime on multicore-based massively parallel machines, we develop insights into how a parallel language runtime can utilize multicore nodes, which is useful to implementations of other parallel language runtime systems. Optimization techniques developed to address performance issues identified in the runtime are generally applicable to other systems or applications that are targeted at being optimized for multicore-based parallel machines.

1.2 Description of Problems and Scopes

Based on possible benefits of a multithreaded runtime on parallel machines built from multicore chips, this work will explore the design space of a multithreaded runtime. In addition, this work investigates various issues and corresponding techniques to attack them both in single-node and multi-node environments. This investigation is done in the context of the asynchronous message-driven CHARM++ runtime, which is used as a platform to evaluate and implement the multithreaded runtime designs. In summary, the following issues are identified and addressed in my work:

1. *What are the key factors to achieve good performance for the multithreaded runtime on a single node, and what techniques are to be used to improve its performance?* We will examine common runtime data structures and identify common pitfalls in supporting them on a multicore node. In particular, we focus on the optimization for the communication within a node.

2. *What is the best way to exploit fine-grained single-node parallelism in an asynchronous multithreaded runtime?* We will study multiple mechanisms including using OpenMP, and show that it is better to exploit single-node parallelism with a mechanism we develop that reuses the existing multithreaded runtime framework. In essence, we propose a unified language runtime that can exploit parallelism at different levels of granularity.

3. *What are the performance issues of a multithreaded runtime when executing on multiple nodes, and what techniques should be employed to deal with them?* This issue requires investigation from several perspectives. First, we explore the design space on how the responsibilities of performing computation and communication should be assigned among runtime threads. Secondly, closely related to the first aspect, we study the communication substrate to be used in the multithreaded runtime and how it can be used better in an asynchronous message-driven runtime. We particularly focus on the case of MPI, as it is the *de facto* standard high-performance message-passing library available on every parallel machine. Finally, we investigate the general communication issues that are exhibited in a multithreaded runtime and develop techniques that are applicable to both the runtime system and applications.

4. *How can we leverage the capabilities provided by a multithreaded runtime to optimize other features of a parallel programming system or performances of user-level applications?* We will demonstrate this through two use case studies. One is about utilizing a language construct that exposes the multithreaded execution mode. And the other is about exploiting the dedicated communication processing power in the multithreaded runtime, especially to improve the responsiveness of the asynchronous non-blocking collective communication.

5. *What factors and strategies should load balancers consider differently for multicore-*

based parallel machines? We will study the differences between multicore chips and traditional single-core processors as well as the changes brought by a multithreaded runtime. Based on those differences, we present our insights and develop new load balancing strategies that are more effective on multicore-based parallel machines.

1.3 Thesis Contribution

In an era when exascale computing is expected within a decade, there is a growing consensus in the high-performance computing community that shared-memory programming should be adopted within a node while message passing should be used across nodes. This thesis examines the various issues that arise from the implementation of such a programming model. The findings and techniques described in this thesis to handle performance issues could shed light on the efforts of the community towards the realization of programming models for exascale computing. In short, this thesis makes the following contributions:

- A systematic study on a user-transparent multithreaded parallel language runtime that targets multicore chips for massively parallel machines, identifying its single-node and multi-node issues with regard to design and performance.

- Development of corresponding optimization techniques to address the various performance issues, including a better way of exploiting more fine-grained single-node parallelism and new load balancing strategies that consider the characteristics of multicore chips.

- A demonstration that a multithreaded runtime can improve the performance of real scientific applications, and it enables better scalability of a widely used molecule dynamics application up to almost all computing cores of full petascale parallel machines.

- A demonstration that application programmers can further leverage the capabilities offered by the multithreaded runtime to optimize other features of the parallel programming system.

1.4 Thesis Organization

In the remainder of this thesis, I will first describe the background of my work in chapter 2, which includes the introduction of the CHARM++ system, its differences from MPI as well as scientific applications and parallel machines that are used in the work. Afterwards, I will present the benefits of the multithreaded runtime in chapter 3. Then I will show studies on single-node performance issues in chapter 4, mainly focusing on problems in intra-node communication and their corresponding solutions. In chapter 5, I describe a better way of integrating more fine-grained single-node parallelism inside an asynchronous message-driven multithreaded runtime. After investigating the issues on a single node, chapter 6 will pay attention to runtime performance issues in multi-node environment, especially focusing on analysis of inter-node communication related problems and ways to handle them. This chapter ends with user-level optimization techniques to leverage the new capabilities provided by the multithreaded runtime. I will introduce new dynamic load balancing strategies improved for multicore chips in chapter 7 before presenting the evaluation of production-level scientific applications running with this mulithreaded runtime in chapter 8. Finally, I will show related work in chapter 9, and finish the thesis with conclusion and future work in chapter 10.

2 Background

This chapter starts with an introduction of CHARM++ runtime system including its programming model and execution model as it is the software infrastructure that lays the foundation of this work. Then, the major differences between MPI and CHARM++ are described in order to better understand CHARM++. In the end, the production-level scientific applications used in performance experiments are introduced as well as a brief summary of the parallel machines used to carry out the study.

2.1 Essentials of CHARM++ Runtime System

CHARM++ is an object-oriented C++-based parallel programming system, featuring asynchronous message-driven execution [18]. With the encapsulation of data and work in objects, a.k.a *chare*s in CHARM++, it naturally promotes the data locality that is needed to achieve good performance on multicore architectures. Additionally, it encourages over-decomposing the problem into many objects and letting the adaptive runtime map those objects to CHARM++ logical processors, called PEs (for processing elements). A CHARM++ PE is a logical control flow handling computation and communication, a counterpart of a task or rank in MPI. In CHARM++, a PE is mapped to a distinct core (or hardware thread). A CHARM++ implementation will not oversubscribe the node because the benefits of over-subscription [19] could be achieved by over-decomposing the problem instead.

Objects interacts through message passing via the invocation of an *entry method*. When an object A sends a message to another object B, the message is first delegated to object B's proxy located on the same processor with object A via the invocation of an *entry method*. Then, the computation identified by the message handler of the *entry method* is triggered by the receipt of this message on object B, and it is executed non-preemptively. The messaging mechanism in CHARM++ can use different low-level communication support, or even MPI if no specialized support is available for that underlying network hardware.

In order to check if there is any incoming message for an object on a PE, continuous message polling is used. Upon the receipt of the message, the associated *entry method* is executed in the same context. Additionally, as the runtime is executing on parallel machines connected via high-performance networks, the network progress is required to be made continuously. In essence, operations performed by a processor in the CHARM++ runtime are illustrated in the left part of Figure 2.1. Note that in the actual implementation of CHARM++ runtime, there are multiple message queues serving different purposes such as handling messages of different priorities. However, as shown in Figure 2.1, all those multiple queues are represented by one queue for simplification without sacrificing the description of essential control flow of a processor in CHARM++. Such continuous-polling type of work running on a PE discourages oversubscription as well because it creates severe resource contention and switching overheads.

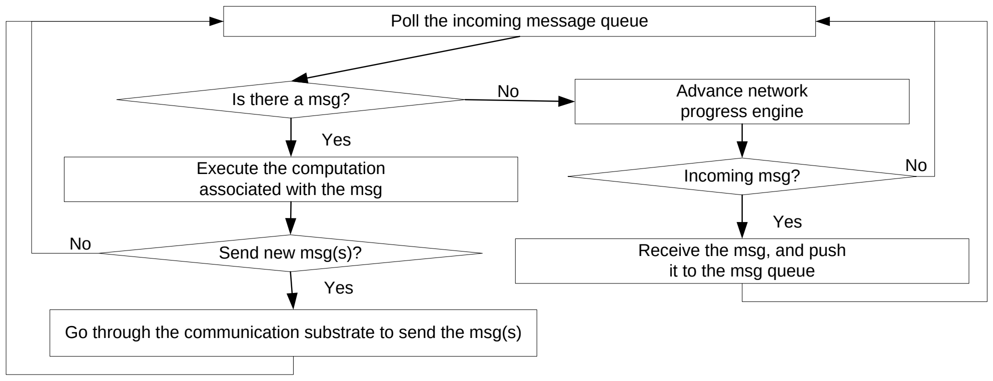
Poll the incoming message queue
Is there a msg?
No
Advance network progress engine
Yes
Execute the computation associated with the msg
Incoming msg?
No
Yes
No
Send new msg(s)?
Receive the msg, and push it to the msg queue
Yes
Go through the communication substrate to send the msg(s)

Figure 2.1: The Flow of a PE in non-SMP Mode

Traditionally, the PE is encapsulated in an OS process, and we call it the "*non-SMP*" mode of CHARM++. The key implementation characteristics of this mode are summarized as follows:

- An **OS process** represents a CHARM++ PE, and is mapped to a logical CPU (a core or a hardware thread).

- **Disjoint** virtual memory address space for all PEs on a physical node.

• **Alternative** computation and network communication per PE.

Note that messages between PEs in the same node are handled in the same way as inter-node messages. Based on the description above, it is clear that the concept of multicore nodes is not reflected in this traditional CHARM++ runtime.

Well before multicore chips dominated the consumer market, shared-memory multi-processor nodes had been used for high-end parallel machines for a long time in HPC. Thus, CHARM++ introduced the multithreaded runtime called "*SMP*" mode trying to take advantage of these parallel machines. In this mode, the PE is encapsulated into a thread, and all the threads that share the single memory address space consist of a CHARM++ logical *SMP node*. Now the CHARM++ logical node is associated with an OS process while it is the logical PE that is associated with the OS process in non-SMP mode. Within this CHARM++ logical node, data can be shared via pointers in CHARM++ programs. Although objects in CHARM++ typically do not exploit such sharing directly, CHARM++ system libraries may exploit this feature. A physical node may be partitioned into one or more CHARM++ logical nodes. In both modes, a PE is always mapped to a distinct physical core or hardware thread (i.e., two PEs are never mapped to the same physical core or hardware thread). However, the performance of "SMP" mode, prior to the work done in this thesis, was not satisfactory, as demonstrated by Figure 1.1. Furthermore, the non-SMP mode of CHARM++ was good enough to deliver excellent performance on real applications, thus diverting our limited energy to other important research issues. Given the wide adoption of multicore nodes, the SMP mode of CHARM++ started to gain attention. Therefore, this thesis is built on this basic structure of SMP mode with efforts on exploring the design space and rationalizing our implementation in addition to identifying performance issues in the runtime and investigating corresponding optimization techniques to address them.

2.2 Differences of Program Execution between MPI and CHARM++

In CHARM++ programs, a message to an object is sent asynchronously in the form of invoking a non-blocking entry method on that object's proxy, and the arrival of this message on that object triggers the associated work. The receiving side of this message (i.e., the object) has no idea of when this message is expected to arrive, and even may not be aware of where this message comes from, or what message is expected to be received. In other words, the receiver is blind to the message sending operations, and all the messages are "unexpected" from the perspective of the receiver. This style is a natural consequence of the message-driven programming model. The express advantage of this style creates a performance challenge, especially when it is being implemented on top of MPI. We address this challenge in section 6.2. Note that in CHARM++ programs, user-defined persistent communication operations also exist but are not commonly used. Messages then become "expected" under such circumstances.

In contrast, MPI programs use two-sided communication interfaces, in most cases, the message is involved with actions on both sending and receiving side. When a message is sent from the sending side, the receiver will correspondingly prepare for the arrival of this message at some time regardless of the sending time. Such "expected" message-passing mechanism in MPI is the main difference from the "unexpected" one in CHARM++.

Since MPI 2.0, one-sided communication interfaces have been provided to take advantage of the capabilities of RMA network hardware [1]. Such one-sided communication sounds like the communication in CHARM++ on the surface, but it is far more restricted in real usage. First, only a special memory region from a MPI rank could be used in MPI one-sided operations, and this memory region must be exposed by a collective call. Secondly, MPI one-sided operations must be synchronized either involving the sender and the receiver, or involving only the sender but requiring globally exclusive access to the special memory region. In contrast, the communication in CHARM++ is free of those restrictions. Such restrictions also prevent existing versions of MPI (1.1 or 2.0) from being the efficient compilation target for PGAS languages [20]. The MPI 3.0 may have features to alleviate these problems, but their utility in the context is not demonstrated yet.

2.3 Major Scientific Applications Used in the Study

In this study, the primary scientific application used intensively is **NAMD**–one of the most widely used biomolecular dynamics simulation application in the world [17]. Biomolecular simulation is a very important approach to understanding the functioning of biological systems. The atom-by-atom simulation of such systems helps us to determine the relationship between the structure of proteins and their functions, understand various biological processes, and facilitate rational drug design and nano-device construction. Generally, for this type of simulation, the individual timestep needs to be carried out in milliseconds in order to obtain interesting simulation results. Considering the variety of computations that must happen during this time and the communication dependencies between them, it is highly challenging to achieve such a short timestep. NAMD is developed in the CHARM++ parallel programming system and uses a hybrid parallelization scheme that is similar to ones described in [21, 22]. In this method, atomic coordinates and velocities are stored and propagated by static spatial domain objects called "patches" while the calculation of interactions between atoms is decomposed into independently migratable "compute objects".

The work also studies the performance of a cosmological application–**ChaNGa** [23] that is used for the simulation of the evolution of the universe. ChaNGa handles forces generated by both gravitational and hydrodynamic interaction. It already considers multicore-related optimizations at the application level, and makes extensive use of the CHARM++ language constructs that are designed for the SMP mode. With these optimizations, during the computation of the forces it leverages the shared memory and avoids all communication within a node. Communication is present during other phases of the iterative process, such as during construction of the global tree.

2.4 Large-scale Parallel Machines Used in the Study

In this study, we have used the following large-scale parallel machines shown in Table 2.1. The first column of the table indicates the name of the machine which will also be used in the thesis to represent the platform. The second column shows the place where the machine is hosted. The remaining columns illustrate the architecture characteristics of the machine. Note the last parallel machine JYC is a smaller system having the same configuration with the NSF-funded sustainable petascale machine–BlueWater.

Name	Location	Memory per node	#Cores per node	Network	Processor Type
JaguarPF	ORNL	16GB	12	Cray Seastar2+	AMD Opteron 8435 2.6GHz 6-core Istanbul
Kraken	NICS				
Hopper	NERSC	32GB	24	Cray Gemini	AMD Opteron 6172 2.1GHz 12-core Magny-Cours
Intrepid	ANL	2GB	4	IBM Proprietary	IBM BlueGene/P 0.85GHz 4-core PowerPC 450
Taub	UIUC	24GB	12	Infiniband	Intel Xeon X5650 2.66GHz 6-core Nehalem
Titan	ORNL	32GB	16	Cray Gemini	AMD Opteron 6274 2.2GHz 16-core Interlagos
JYC	NCSA		32		

Table 2.1: Parallel Machines Used in the Study

3 Benefits of A Multithreaded Runtime System

This chapter describes the benefits we have identified that a multithreaded runtime will bring to programmers and applications. First, we have a short summary of the multi-threaded runtime (referred to as the "SMP" mode). The idea behind such a parallel language runtime is a natural consequence of the common practice in single-node shared-memory programming that an application is parallelized using threads to harness the computation power of a multicore chip. In the context of a language runtime, the control flow on each logical CPU is now encapsulated into a thread, such as Pthread [7], instead of an OS process. We regard that the threads spawned by the same OS process now consists of a SMP node. The single memory address space shared by those threads provides a natural way to exploit the shared physical memory on a multicore node. Speaking of MPI, this SMP mode corresponds to the scenario in which each MPI rank is mapped to a thread in runtime. It differs from the hybrid mode of MPI+pthreads or MPI+OpenMP where each MPI rank is still mapped to a process while threads spawned by the process are not considered to be MPI ranks.

Based on our experience with the multithreaded runtime, the following benefits have been identified that a multithreaded runtime could at least bring to the applications and their developers:

- **Low-overhead and faster intra-node communication:** because all the threads are in the same memory address space, the intra-node communication turns to simply passing the message pointer in CHARM++. Comparing this mechanism with process-based intra-node communication, this method first avoids the double memory copy as used in methods via posix-shared memory such as [2], and secondly saves overheads on switching between user mode and kernel mode in the OS as well as packing/unpacking control information in the single-copy mechanism assisted by OS kernel [3, 4].

#Nodes	140	560	2240	4480	8960	17920
#Cores	1680	6720	26880	53760	107520	215040
non-SMP (MB)	838.09	698.33	798.14	987.37	1331.84	1760.86
SMP (MB)	280.57	141.83	122.41	126.03	131.84	157.76
Reduction Rate	2.99	4.92	6.52	7.83	10.10	11.16

Table 3.1: Comparison of Average Memory Footprint in SMP and non-SMP Mode for 100M-atom Simulation in NAMD

- **Memory footprint of programs reduced:** there are two sources of saving memory in SMP mode. First, memory usage in the system-level may be reduced, especially for the communication library. Take the SMP mode built on MPI as an example, as the number of MPI processes in a program is reduced by the node size in SMP mode, the memory taken for inter-node communication channels setup, pre-allocated message buffers is reduced correspondingly. Secondly, the single memory address space shared by PEs in the same node allows a single copy of data structures that are read-only or ones that are written only during the startup of the application or during certain execution phases. The memory footprint of the program will be reduced further by being aware of this possibility of sharing data structures in SMP runtime mode. For instance, in NAMD, the molecule object that contains static physical attributes of atoms, and the map object that tracks the distribution of patch and compute objects are optimized to be shared in a node. Thanks to this memory sharing, the 100M-atom simulation could start to use more than one core per node on Intrepid where only 2GB of memory is available per node. If simulating this huge molecule system in the non-SMP mode, however, about 2GB is required per CHARM++ PE. Consequently, just one PE could be launched per node on Intrepid so as to meet the memory constraints, leaving three cores of out of four wasted without any computation. The following table 3.1 shows the actual memory footprint consumption of the 100M-atom simulation both in non-SMP mode and SMP mode on JaguarPF, illustrating the benefit of SMP mode in memory footprint reduction. Note that the table is excerpted from my published paper [24]. According to table 3.1, we first can observe that the overall memory usage of each mode will first decrease and then increase when the number of nodes used grows. This results from a mix of three factors: a) the whole input data distributed on every core is reduced with the increase in the number of cores used; b) the memory usage of some data structures, including those that are shared across cores within a SMP node, grows linearly with the number of cores used; c) the CHARM++ runtime, including the lower communication library (MPI in this testcase) also requires more memory when the simulation scales up. Furthermore, shown by the last row in the table, it is clear that there is a growing memory reduction rate with the increase of nodes used for simulation. In short, we obtained an average of $\sim$7X reduction rate for memory consumption in SMP mode.

- **Reduced job startup time:** in SMP mode, only one process is created for a SMP node composed of multiple cores, which also implies that there is only one instance of the underlying communication library. In constrast, in non-SMP mode, the number of processes created is the same with the number of cores requested. As a result, the job launch time will be significantly reduced, especially for very large scale runs. It has been demonstrated by our experiment of running NAMD on JaguarPF using MPI as the communication substrate for CHARM++. In particular, for a run with 224,076 cores, it took about 1 minute to launch the job in SMP mode while it took about 6 minutes in non-SMP mode! This is because the total launched MPI ranks in SMP mode is reduced by 12 times! In SMP node, we created one MPI rank per physical node (12 cores/node) whereas each physical core hosts one MPI rank in non-SMP mode.

- **Transparency to application developers under the same programming model:** with parallelizing the language runtime into a multithreaded one, it is transparent to application developers because they will still continue to write programs under the same programming model. The same application that runs in non-SMP mode will almost run flawlessly in SMP mode, and vice versa. This is especially feasible in CHARM++, because the programmers write in terms of objects (chares) without any reference to processors and then the runtime will adaptively map those objects to processors. Therefore, the software implementation of a processor hardly affects the above user applications.

In addition, particularly for CHARM++, since the data and work have been encapsulated in the parallel object, it naturally promotes the data locality which is required to achieve good performance in multithreaded shared-memory programming. As a result, maintaining the same programming model is more favorable than the hybrid programming approach. Admittedly, to make the best of the multicore chips, an additional different programming model such as the shared-memory programming model may have to be used so as to achieve the best application performance. However, by exposing the additional programming model to developers in a very limited way through certain constructs or features of the parallel language that the multithreaded runtime is associated with, we believe that the burden on the developers to use an additional programming model for the multicore platforms can be reduced.

In spite of these significant benefits, the SMP mode of CHARM++ and the multithread MPI implementation have not been very popular. A large fraction of the applications still continue to use the non-SMP mode where each core constitutes an OS process which represents a MPI rank or a CHARM++ PE. This is in large because the performance is no better in SMP mode as shown by Figure 1.1, and sometimes it is even significantly worse as we will see in section 4.1. A major objective of this dissertation is to understand the performance loss seen in practice, and explore optimization techniques to eliminate the performance loss so that the potential benefits of SMP mode described in this chapter could be retained.

4 Investigation of Single-node Runtime Performance

The performance of the runtime on a single node is the basis of the overall performance of the runtime in a massively parallel machine. In this chapter, we will examine the following two issues:

• What are the key factors that affect the performance of a multithreaded runtime on a single multicore node?

• Based on the identified factors, what are the optimization techniques that can be employed to improve the performance?

We investigate these issues through studying the multithreaded CHARM++ runtime (i.e., the SMP mode). We examined common data structures used in the runtime and pitfalls of shared-memory multithreading programming. We have used "kNeighnor", a communication intensive synthetic benchmark written in CHARM++, to evaluate each optimization technique we developed to address a certain performance issue. At the end of this chapter, performance results on real scientific applications are presented. This part of work has been published in paper [25]. Although the corresponding optimization techniques developed for the identified performance issues in this chapter look straightforward from the current perspective, because the CHARM++ system evolved in the last two decades, those optimization techniques are not supported or developed at the time when implementing the feature that turns to be problematic in performance nowadays. From this point of view, the case study presented in this chapter also serves as a summary of optimization techniques that are available currently to improve performance for a multithreaded runtime effectively.

4.1 Study Environment Setup and Initial Performance

The benchmark–kNeighbor creates a certain number of objects distributed on the parallel machine, and arranged in a 1-dimensional array. In each iteration, each object sends a message to its k neighbor objects on both sides in a wraparound fashion. When each object has received all the expected messages ($2 * K$), it proceeds to the next iteration. Throughout the study described in this chapter, k is set to 3 and the number of objects is set to be same with the logical CPUs of the node size (as the program is evaluated on a single SMP node) so that every PE has exactly one object. Clearly, this benchmark emphasizes on the performance of the intra-node communication. As the multithreaded runtime is expected to have smaller latency for intra-node communication, we also compare the performance of kNeighbor with that in non-SMP mode using the same configuration in order to demonstrate such benefit enabled by the multithreaded runtime. The iteration time reported is averaged over ten thousand times.

We have conducted this study on multiple multicore single-node platforms, including AIX running on IBM Power5 and Linux running on both Intel and AMD processors. Those platforms are summarized below:

- Platform A: AIX 6.1/IBM Power 5, a 16-core (SMT=2) node
- Platform B: Ubuntu 8.04/Intel Nehalem Xeon E5520, a 8-core (SMT=2) node
- Platform C: Ubuntu 8.04/Intel Harpertown Xeon E5405, a 8-core node
- Platform D: Ubuntu 8.04/AMD Barcelona Opteron 2356, a 8-core node
- Platform E: CentOS 5.4/Intel Dunnington Xeon E7450, a 24-core node

The hardware configurations of those platforms are representative of current large-scale parallel machines. Based on our study, we have found the results are similar for all the architectures, so we refer only to platform C in the following discussions unless there are some different results on other platforms. The eight cores of the representative platform C are illustrated in figure 4.1 together with their topology viewed by the OS. We categorize data transfers between cores as “intra-die”, “inter-die” and “inter-chip”, ordered by increasing data transfer latency. In our experiments, we have generally left one core free to accommodate noise from OS daemons.

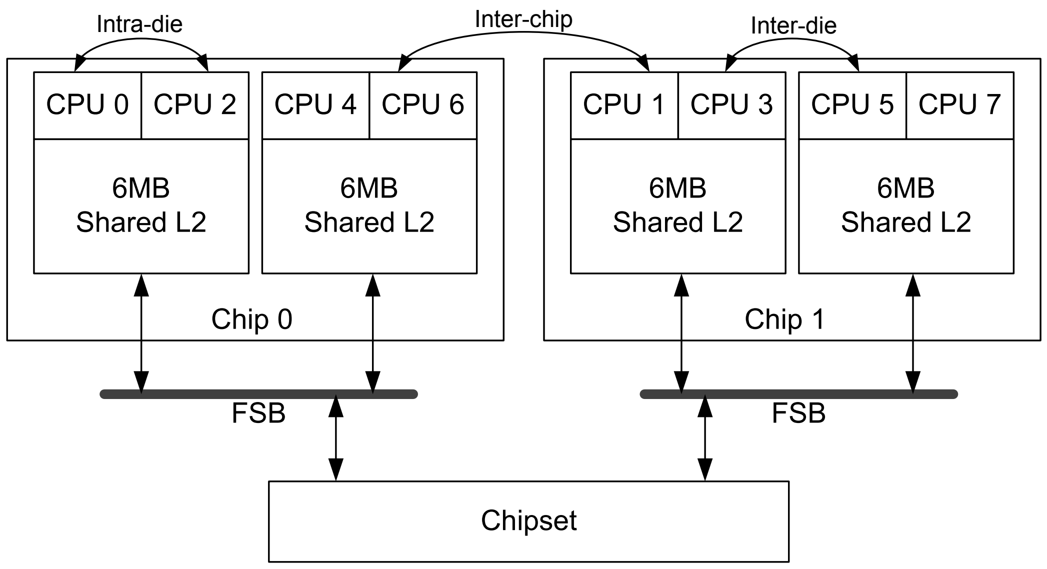

Figure 4.1: The CPU Topology of the Single-node Platform

We describe each identified key performance factor followed by the corresponding optimization technique. The effectiveness of each optimization is illustrated by reporting the performance before and after on the kNeighbor benchmark. Figure 4.2 shows the performance comparison between the non-SMP modes and the very initial SMP mode implementation in CHARM++ for the kNeighbor benchmark running on the representative platform C with varying message sizes. Note "PXSHM" in the figure indicates the non-SMP mode that have used posix-shared memory to optimize the inter-process communication within a physical node, while the "non-SMP" represents the general non-SMP mode that uses the underlying message-passing library (TCP/IP in this case) to do the inter-process communication. It is clear that the performance of SMP mode surprisingly lags far behind that of the non-SMP modes (i.e., $\sim$10X slower compared with "PXSHM", and $\sim$3X slower compared with "non-SMP") in spite of the better intra-node communication mechanism via threads which should entail that the SMP mode outperforms non-SMP modes.

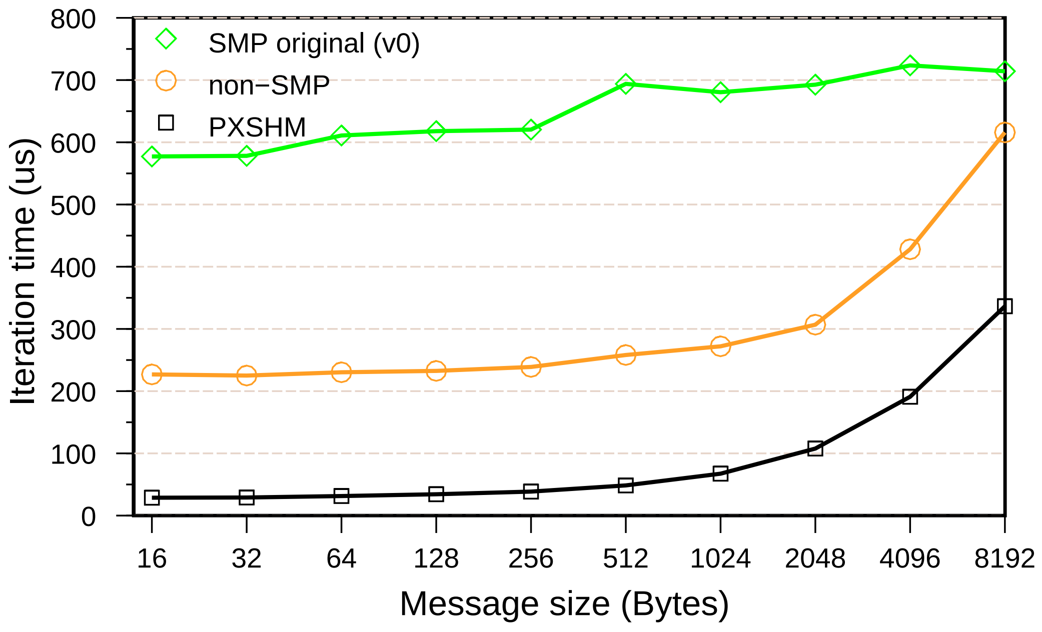

Figure 4.2: Performance Comparison between non-SMP Mode and Initial SMP Mode

4.2 Removing Extra Invocations of Network Progress Engine

A parallel language runtime has to handle network communication work in order to run on distributed-memory parallel machines. As a design choice, each control flow (i.e., a PE in CHARM++ or a rank in MPI) calls the network progress engine after sending a message to make sure the outgoing messages gets put on the network right away, and to check for incoming messages for responsiveness. The implementation of the non-SMP mode of CHARM++ adopts such a design. In the initial implementation of the multithreaded CHARM++ runtime (i.e., the SMP mode), we kept this design and focused on ensuring the thread-safety of the runtime. Through running the kNeighbor benchmark, we observed a very high overhead due to expensive network progress engine calls. Realizing that the message-passing within a single node involves no network communication as they are performed via memory pointers, we reduce the overhead caused by the network progress engine by skipping its invocations for intra-node messages but keep the calls for network messages. The results of this optimization as compared to the original scheme are shown in figure 4.3. On average, we see about 35% improvement. Observing the figure, we find that kNeighbor performs better in SMP node than in non-SMP mode for message sizes larger than 8KB, but the performance in SMP mode is still worse for message sizes below 8KB, which leads to further investigation.

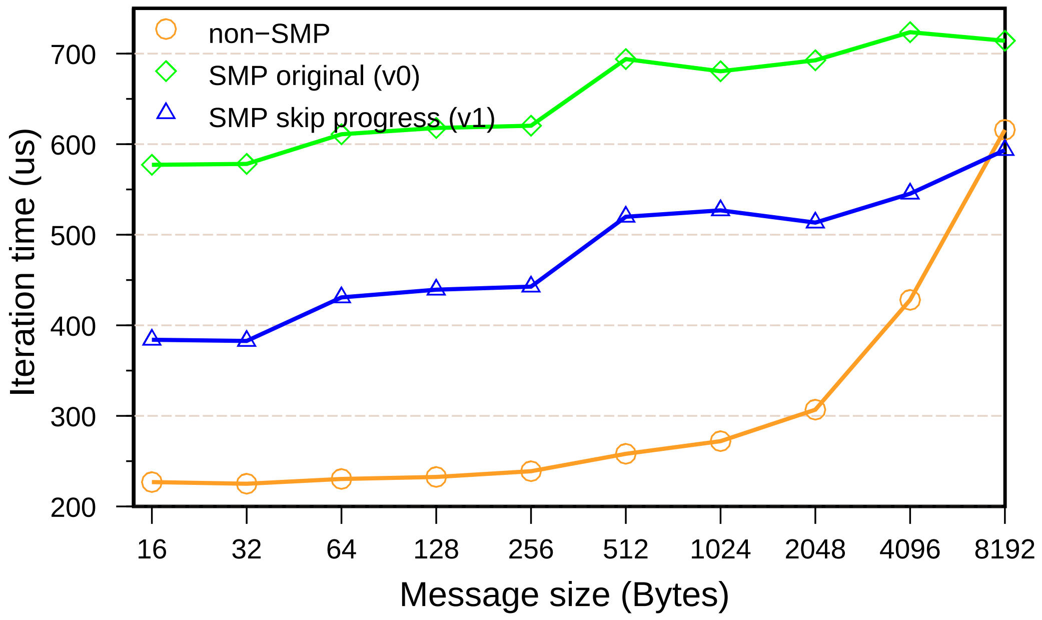

Figure 4.3: Performance before/after Skipping Network Calls

It is worth mentioning that this particular performance optimization reminds us that transforming a parallel language runtime system into a multithreaded one to take advantage of multicore chips does not just involve engineering the whole runtime to be thread-safe. It also invites us to re-think the design of the multithreaded runtime, especially in the trade-off of dividing work responsibility among different threads. Such design space exploration will be discussed in details in section 6.1.1.

4.3 Reducing Contention in Making Runtime Multithread-Safe

We looked at performance issues in the approach we take to make the SMP mode of CHARM++ thread-safe as we intended to optimize the intra-node communication. We concluded that efficiently handling locking and synchronization among threads in the runtime is the key factor for obtaining fast, fine-grained intra-node communication, which are reflected in three aspects as described in the following subsections.

4.3.1 Memory management

In CHARM++, we found that the self-implemented default memory allocator has used a lock to protect every *malloc* and *free* call to have robust support for multithreading in the runtime. It was no surprise that kNeighbor performed very poorly because of the severe lock contention due to message allocation, amplified by the intensive communication. After switching to the memory allocator provided by OS, which is robustly thread-safe to be directly used in CHARM++ SMP mode, we see a significant performance improvement as 2.4 times on average for kNeighbor's iteration time illustrated by the third curve in figure 4.4. According to figure 4.4, it is obvious that the performance of kNeighbor in the current SMP mode becomes much better than the initial one after switching to OS-provided memory module (the third curve vs. the first curve), and it even exceeds the performance in the non-SMP mode (the second curve). However, compared with PXSHM (the fourth curve), the performance of SMP mode is still worse by 4.3 times on average for message sizes below 4KB. As a result, the challenges of investigating performance issues in SMP mode on a single node still remain for beating the performance of the PXSHM mode.

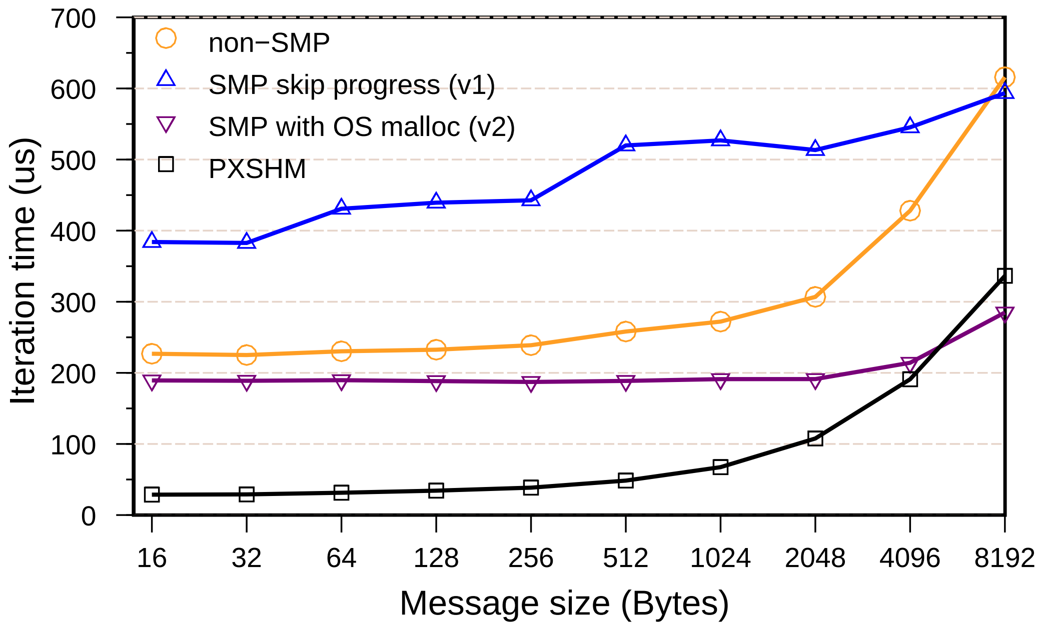

Figure 4.4: Performance Comparison before/after Using a Better Memory Allocator

The above finding that the contention in the memory management plays an important

#threads	A(us)	A/M(us)	B(us)	C(us)	D(us)	E(us)
1	1.06	1.03	0.78	0.80	1.13	0.68
2	2.23	1.02	1.30	1.44	1.53	2.03
4	6.06	1.05	3.95	2.14	2.36	2.73
8	15.35	1.03	8.71	3.69	4.72	7.06
16	36.89	1.06	22.63	n/a	n/a	14.58
24	n/a	n/a	n/a	n/a	n/a	21.31
32	210.96	1.02	n/a	n/a	n/a	n/a

Table 4.1: Memory Allocation Time on Different Platforms.

role in the performance of the multithreaded runtime led us to investigate how the OS-provided memory allocator performs when the number of threads increases in a process. We synthesized a benchmark that every thread continuously allocates memory of the same size and deallocates at the end for 100,000 times. We run the benchmark with the varying number of threads, and take the average memory allocation time on 5 different multicore platforms mentioned earlier.

Table 4.1 shows that the default OS memory allocators tend not to scale when the number of threads increases up to the maximum number of logical CPUs on each platform. However, we should be cautious to interpret these results as real applications may not involve a great number of simultaneous memory allocations. The results here are very likely to imply the upper-bound of overhead in memory management under the multithreaded environment. Based on table 4.1, among all the five platforms, the AIX 6.1/Power 5 platform A, illustrated by the second column, did the worst with the default setting. Setting system environment variable “MALLOCMULTIHEAP” under AIX, however, improves the performance significantly as shown in the third column. In this case, the time for each memory allocation remains almost identical, indicating there is no more contention . The idea behind this setting is to use multiple heaps, one serving several threads to reduce contention on heap allocation. Although this mechanism will increase the overall memory usage and memory fragmentation, it is still a good optimization to consider thanks to the significant performance improvement it results in. Observing that the memory allocation does not scale well either on Linux plaforms, and Linux itself by default does not provide an alternative as AIX does to the best of my knowledge, we suggest a better default memory management be adopted on parallel machines running Linux. This will not only benefit multithreaded programs but also the language runtime discussed in this work.

In summary, as threads share the same memory address space, it is critical to deal with multiple, simultaneous memory allocation or deallocation requests in order to achieve thread-safety with low contention while keeping a high memory utilization (i.e., low internal/external fragmentation) at the same time. In the context of a message-passing based parallel language runtime system, the message latency includes the time spent on the memory allocation for this message. Consequently, it is necessary to check whether the overhead introduced by the contention in memory management on a particular platform slows down the runtime performance. We have noticed that much work has been done to address this issue [26, 27, 28], and we will evaluate that work in the context of the multithreaded runtime in the future.

4.3.2 Granularity of critical sections

It is easier to use a big critical section to make runtime thread-safe. However, this leads to a serialization of program and poor performance. After carefully reviewing the code base of CHARM++ runtime, we removed unnecessary locks and reduced the granularity of critical sections to minimize synchronization overhead, e.g., we put the lock only around the part of the function that is not thread-safe instead of blindly putting it around the whole function body. Indeed, this is a trade-off between productivity and performance, because larger critical sections tend to be safer, while reducing their scope requires analyzing complex interactions for race conditions, and tedious debugging efforts. However, for a runtime that is at the foundation of a parallel programming system, and one that is used very often, the effort to improve performance is worthwhile.

This turned out to be a significant performance improvement as indicated by figure 4.5. Compared with the previous SMP version, kNeighbor benchmark speeds up by an average of 2.7 times up to message size 2KB. Beyond that, we can observe a trend that the performance gain is diminishing, but still with 35.1% improvement on average for message sizes 4KB and 8KB. We believe such performance trend is caused by the fact that the execution

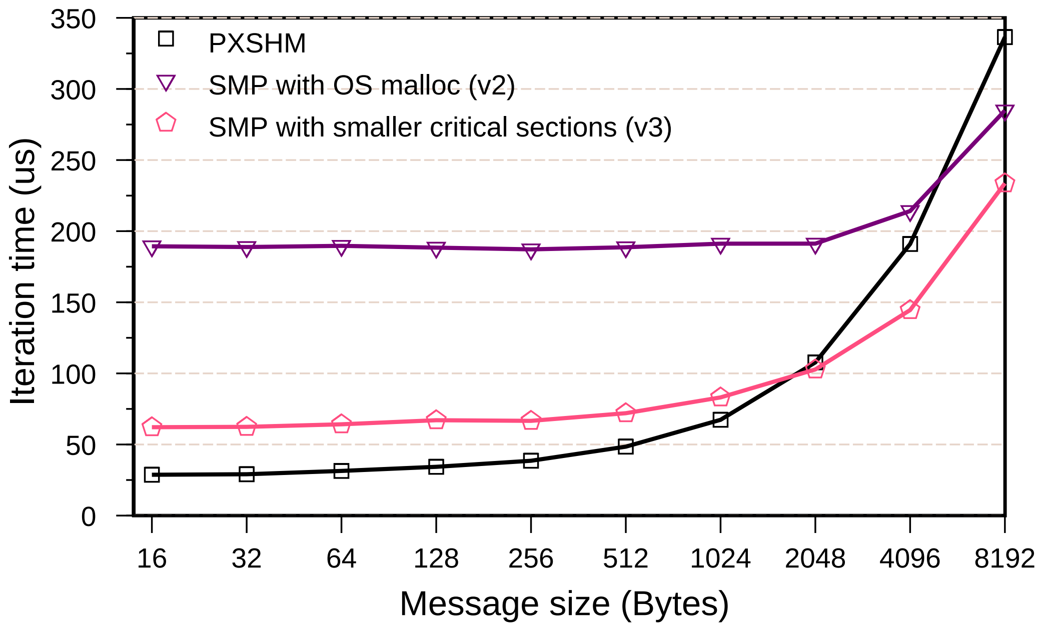

Figure 4.5: Performance before/after Reducing Granularity of critical sections.

time of kNeighbor begins to be dominated by touching every byte of the message on the receiver side. Comparing with the PXSHM version, we can see now the two performance lines cross at a message size smaller than 2KB. Additionally, it is clearly shown that the PXSHM mode has a steeper execution time increase rate than that in SMP mode. This is because we cannot avoid an extra copy from the POSIX-shared memory region to the user space for receiving the message, while such a copy is not needed in SMP mode as only the message pointer is passed to the receiver. However, due to the Non-Uniform Memory Access (NUMA) effect exhibited by an increasing amount of multicore nodes on parallel machines and the “first-touch” memory allocation policy, if the message receiver accesses the data often, it may be better to make a message copy instead of only sending a memory pointer. This will be left to be examined in the future work. The single-node performance issues examined in this chapter are restrained within one NUMA node where every core has the same memory access latency.

4.3.3 Message queues

Producer-Consumer Queue (PCQueue) is a commonly used data structure in parallel language RTS to synchronize multiple threads. For example, it is used in the Cilk [29] scheduler for work-stealing. CHARM++ RTS uses PCQueue for various purposes to synchronize the worker threads and communication thread with messages. For example, the communication thread as a producer pushes a message into a worker thread's message queue for processing. The simplest way to ensure correctness is to enforce a lock every time a thread accesses the queue. This was the original implementation in CHARM++, and it suffered severe thread contention, especially when the number of producers increased.

Lock free implementation is possible for PCQueue, and has been used in the past [30, 31]. In CHARM++, each PCQueue always has a single consumer (the thread for which the messages are destined), but can have multiple producers (all the other cores in the node). First, by using memory fences [32], we can preserve the correctness of the PCQueue operations while allowing producers and consumers to overlap. To take care of the multiple producers scenario, a lock to be shared among the producers is still used. Memory fence operations, however, are highly architecture specific. To simplify the implementation of PCQueue, we developed a portable API that consists of two functions: CmiMemoryReadFence() and CmiMemoryWriteFence(), which serialize the load and store operations respectively. These two APIs call platform specific memory fence instructions, for example *lfence* and *sfence* on X86-based platforms, *mf* for IA64 platforms, and *eieio* for PowerPC platforms. We use the lock-based scheme as a fallback implementation for the cases when memory fence is not supported. Replacing locks with less expensive memory fence operations does improve the performance as shown in the second curve in figure 4.6. We observe up to 9.7% improvement, especially for messages smaller than 2KB, compared with the performance before using memory fences, as represented by the top curve.

As mentioned above, a lock still remains on the producers' side of the queue. Removing this lock could further improve the performance. While a totally lock-free implementation is possible for multiple-producer-single-consumer, the overhead associated with this implementation was significantly high, and the implementation was not stable, especially on some architectures. To address this issue, we introduce multiple queues into the runtime, one for each producer and consumer pair, so that the pure lock-free PCQueue could be used. Clearly, this optimization comes at the cost of a consumer having to poll all the queue pairs,

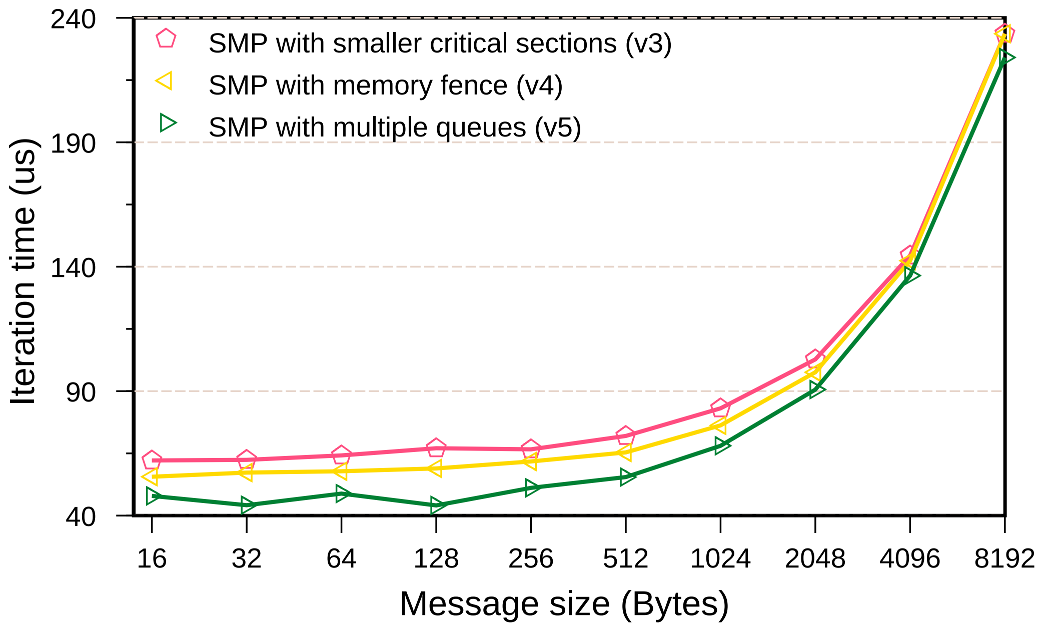

Figure 4.6: Performance Comparison with Memory Fence and Multiple Queues Respectively

which can be as many as the number of cores on a node. Consequently, we observed that the overhead increases as the size of a multicore node grows in our experiments. On the representative platform C, we see that the benefit of removing locks outweighs the incurred polling overhead (about 19.5% improvement on average in kNeighbor benchmark), which is illustrated in the bottom curve of figure 4.6. On the other hand, on platform E with 24 cores, kNeighbor shows no speedup at all with this optimization due to the higher polling overhead. As cluster nodes become increasingly large, further investigation is required to reduce the overhead of producers locks.

4.4 Avoiding Cache False Sharing in the Multithreaded Runtime

Cache false sharing is an important factor in achieving good cache performance, which is critical to application performances, for shared-memory multithreaded programs. Since the message queue as mentioned in section 4.3.3 is one of the most accessed data structures in the runtime, we first optimized its data field layout to reduce cache false sharing. We noticed that two neighboring member fields in the data structure are updated by different threads, therefore, we inserted padding bytes between the two fields to ensure each of them is on a separate cache line.

In addition, another source of cache false sharing in a multithreaded runtime could originate from handling those variables that are similar to the variables that requires privatization in OpenMP. In CHARM++, such PE private variables are used in its runtime implementation and user applications. Applications do not need to be changed for execution in the SMP mode from the non-SMP mode, thanks to the annotation of variables for distinguishing between PE private and shared variables (noted by Cpv and Csv macros respectively). For example, "CpvAccess(*var*)" is to access a PE private variable named "*var*". In non-SMP mode, Cpv variables are simply global variables of a process. But in SMP mode, since CHARM++ PEs on the same node share the memory address space, the implementation has to be different so that each PE (i.e., a thread in this case) has a dedicated copy of such variable.

The initial implementation of PE private variables relies on an array of size equal to the number of PEs on a SMP node to represent the PE private variables. Each rank of PEs uses its own copy of the variable in the array, e.g., accessing a PE private variable *var* is expanded to "*var*[*myrank*]". This was appropriate in early years of CHARM++, since fast thread-private variables were not supported in pthreads. However, this approach becomes inappropriate nowadays because it has a significant disadvantage of cache "false sharing" in SMP mode. Considering two PE private variables of two neighboring PEs, they stay on the same cache line, hence, the update on either of the two variables will invalidate the cache lines inside each other's private cache. This problem was identified with the help of Intel Performance Tuning Utility (PTU)[1].

Platform	A (ns)	B (ns)	C (ns)	D (ns)	E (ns)
TLS	0.40	1.27	1.5	1.75	1.26
Array-based	51.58	17.52	10.03	9.61	8.50

Table 4.2: Time of Updating PE Private Variable on Different Platforms.

[1]http://software.intel.com/en-us/articles/intel-performance-tuning-utility/

To address the issue, we used the thread local storage (TLS) scheme[2] either implicitly if the "__thread" keyword is supported by the compiler and assembler, or explicitly through function calls such as "pthread_setspecific/pthread_getspecific" on Unix-like platforms or "TlsSetValue/TlsGetValue" on Windows. To set an idea of how TLS performs against the array-based solution, we evaluated the time taken to update a PE private integer variable on the five multicore platforms as used in this study with 8 PEs (i.e., 8 threads in total on the node). Table 4.2 illustrates the significant advantage of TLS over the array-based implementation because the latter suffers greatly from cache false sharing induced by the cache-coherence protocol. For example, the array-based performance is about 6 times slower than the TLS-based one for the representative platform C.

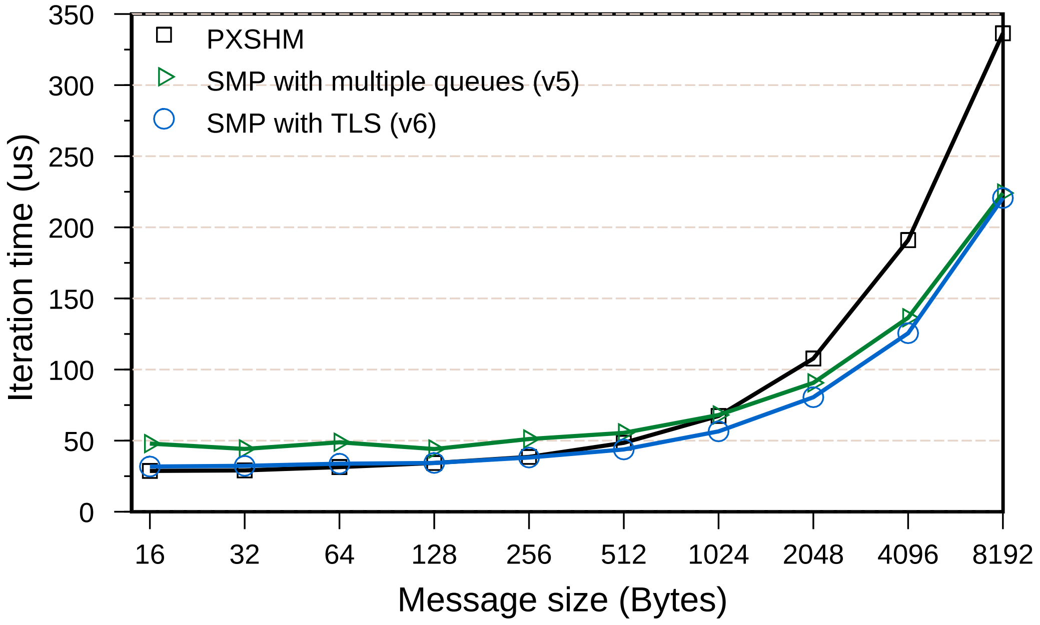

Figure 4.7: Performance before/after using TLS.

Figure 4.7 shows kNeighbor's performance improvement by 26.5% on average after switching to TLS scheme for PE private variables in the runtime system. We can notice a decreasing return for this optimization for messages of relatively larger size as touching every byte of the message on the receiver side begins to dominate the execution time as also mentioned in section 4.3.2. In addition, the performance of SMP mode has been improved closer to that of PXSHM mode considering that PXSHM only outperforms SMP for very

[2]http://en.wikipedia.org/wiki/Thread-local_storage

small messages below 128 bytes.

4.5 Setting CPU Affinity

On the multicore node, the way an operating system binds a process or a thread to a core, and how the OS migrates or schedules a process or a thread among cores has great impacts on the performance.

There has been extensive research on the scheduling algorithms in operating systems for multcore systems to improve overall performance by using process and thread affinity [33, 34]. However, while optimizations performed in this category tend to improve the overall performance of a multicore system and its utilization, they may not benefit the particular application of concern. Instead of being intrusive, most operating systems on multicore systems adopt soft affinity, also called natural affinity, which is the tendency of a scheduler to try to keep processes on the same CPU as long as possible. However, this is merely an attempt; if it is ever feasible, the processes certainly will migrate to another processor. For example, using the same kNeighbor benchmark, we observe that the OS keeps changing the core of a particular thread on a 8-core machine, as shown in figure 4.8.

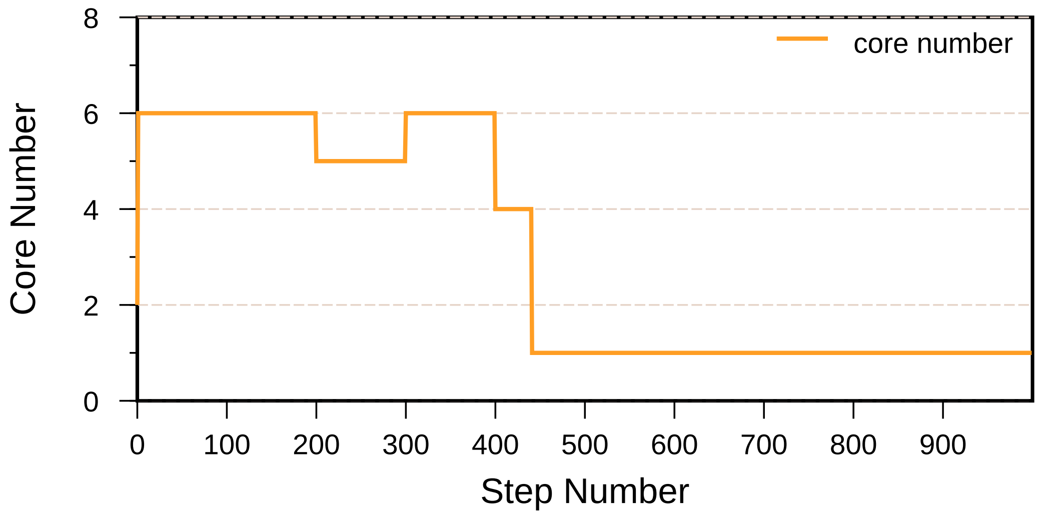

Figure 4.8: OS Keeps Changing the Core of a Thread.

Motivated by this observation, we conducted an experiment to see how performance is affected if we fixed PEs to their cores in SMP mode. There are three potential benefits enjoyed by setting a fixed CPU affinity for PEs of a parallel language runtime:

- The cache performance could be better since OS is prevented from moving a PE (i.e., a process or a thread) to a core which has cold cache. When PEs bounce among cores, they constantly cause unnecessary cache traffic such as invalidation, cold misses etc. Therefore, in performance critical situations, it makes sense to enforce the affinity as a hard requirement.

- OS moving PEs around may conflict with application load balancing effort as an unexpected migration of a PE, together with its work, could change the already balanced load.

- In terms of collecting performance data, setting the same CPU affinity for every run will result in more consistent performance results, which is essential for any useful analysis on the data.

The result of fixing threads to cores is shown in figure 4.9 (lines) using the same kNeighbor benchmark running on a 8-core multicore desktop using various message sizes. We observe up to 15% performance improvement in the total execution time by just doing that! To better understand this, we measured the L1 cache misses for the same runs, as illustrated in the same Figure (bars). We see that for small messages, the number of L1 cache misses are reduced by around 20% by binding threads to their cores, while the reduction of cache misses decreased to around 10% for larger messages. We also observed similar performance boost with several other applications such as NAMD as well. This demonstrated that simply enforcing hard thread affinity is beneficial to applications.

With the encouraging results of fixing the affinity of PEs, a further study was performed on how different affinity bindings affect the application performance. Intuitively, mapping communicating PEs to closer cores in the memory hierarchy incurs less data transfer latencies between cores, which could lead to better overall application performance.

The impact of bindings on performance really depends on the communication pattern of the application. For the kNeighbor benchmark when k=3 running on 7 cores of a 8-core machine, performance does not vary by the different bindings. This is because each element communicates with all 6 other elements on 6 different cores, making bindings unimportant. However, when k=1, every element only communicates with its two neighbors, the binding

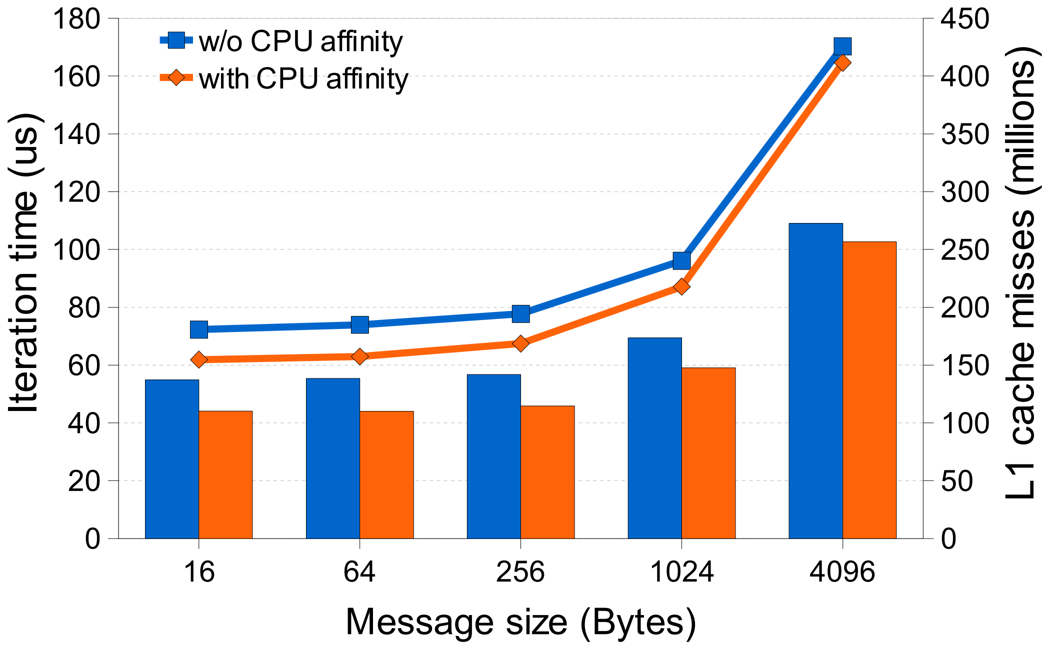

Figure 4.9: *k*Neighbor L1 Cache Misses and Iteration Time

shows significant impact on the performance. For example, when message size is 256 bytes, a mapping of 0,1,2,3,4,5,6 yields a iteration time of 13.37 us, while a mapping of 0,2,4,6,1,3,5 yields a iteration time of 11.66 us.

The execution time difference is primarily due to the different number of inter-chip, inter-die and intra-die messages. In the case of mapping 1, there are 4 inter-die messages, and 24 inter-chip messages per iteration in total. In comparison, the second mapping caused fewer inter-chip messages (reduced from 24 to 8) with cheaper messages of the other two types (increased from 4 to 8, and 0 to 12 respectively). Therefore, the overall performance of the second mapping is better. In order to be able to set the CPU affinity across different platforms, we developed a portable function API (`CmiSetCPUAffinity`) in CHARM++ runtime system to allow programmers to manually bind threads to logical CPUs. The implementation uses the low level system call to bind threads, for example:

- `pthread_setaffinity_np` for Linux and pthreads,
- `bindprocessor` for IBM AIX, and
- `SetThreadAffinityMask` for Fibers on Windows.

In essence, the optimal affinity setting really depends on the communication pattern of the application. It would be an interesting work to develop an automatic and adaptive affinity binding scheme in the language runtime, rather than a static binding scheme based on some *a priori* knowledge of a communication graph of the application [35].

4.6 Decreasing the Number of Memory Accesses

Motivated by the fact that the performance of kNeighbor benchmark is still a tiny bit worse in SMP mode than that in PXSHM mode for very small messages, we further analyzed the fine-grained intra-node communication with Intel PTU, trying to get a deeper insight into its performance, such as finding the most expensive instruction blocks in terms of CPU cycles. We identified that the push/pop operations on the message queue still constituted a high overhead. These operations, despite being simple and short, still had quite a number of cycles per instruction because they contain multiple memory accesses that turn to be particularly expensive due to the frequent execution in this case. This overhead manifests itself when message sizes are small. Thus, we simplified the data structure of the message queue to reduce the number of memory accesses. Consequently, as demonstrated in figure 4.10, the kNeighbor in SMP mode improved by 8.1% on average for messages up to 1KB. We have omitted data points from message size 2KB because this optimization shows negligible improvement due to the fact of touching message data mentioned in the end of section 4.3.2. Compared with the PXSHM mode, it now performs equally well for very small messages and much better for message sizes beyond 512B (due to the copy-free message delivery in SMP mode)! Although this is demonstrated on an Intel architecture, we found our optimization generally helps kNeighbor on other platforms we have access to.

In summary, we have identified a series of performance factors, such as the contention incurred by making the runtime thread-safe, false sharing of cache lines and CPU affinity setting etc., for a mulithreaded runtime based on the large amount of effort we spent on optimizing the SMP mode of CHARM++. We either developed new techniques or applied existing techniques to address each performance issue in the runtime, such as using memory fence to avoid locks, using TLS to implement PE private variables to avoid false

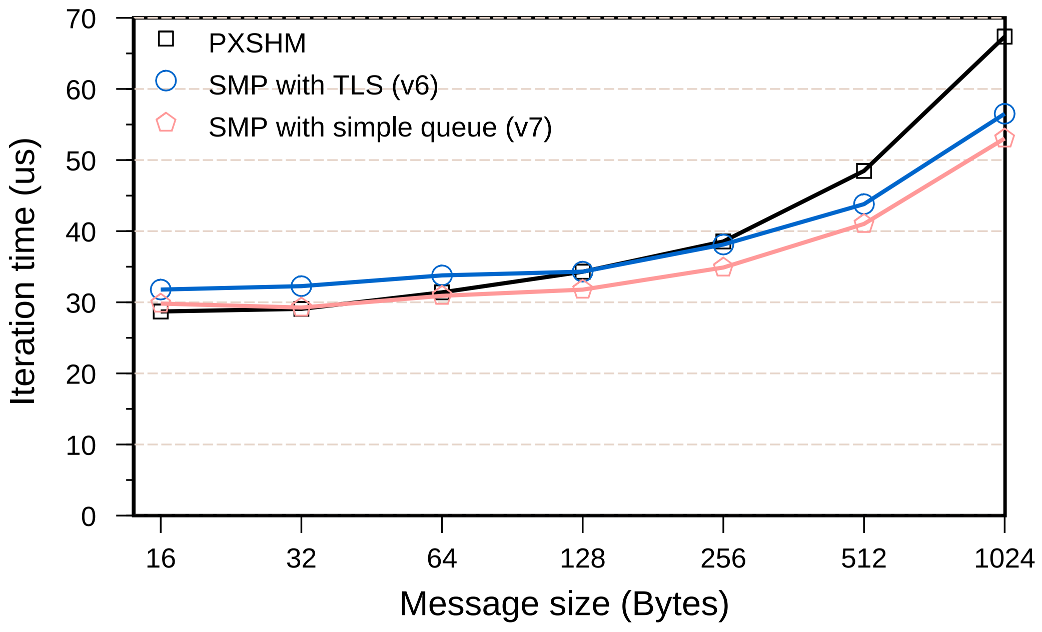

Figure 4.10: Performance before/after Using the Simplified PCQueue

sharing, etc. Figure 4.11 shows the performance result of kNeighbor after our efforts on optimizing the single-node performance. The performance in SMP mode beats both the PXSHM mode and the plain non-SMP mode (omitted for clearer comparison) respectively by about 20.7% and 486.6% on average across the message sizes tested. Comparing with the initial performance of SMP mode, we have obtained amazingly about a 14.4 fold performance improvement! We have achieved a much better intra-node communication, living up to the expectation we have for a multithreaded runtime.

Many optimizations developed here for a more efficient multithreaded runtime are essentially a trade-off between productivity and performance. Simple techniques, hence easy programming, are enticing to be used to implement a just thread-safe runtime, while an efficient one requires more sophisticated techniques, analyzing complex interactions for race conditions and tedious debugging efforts. Nonetheless, the runtime system is at the foundation of a parallel programming system, and it is used very often. We believe it is quite worthwhile exerting great efforts to improve its performance.

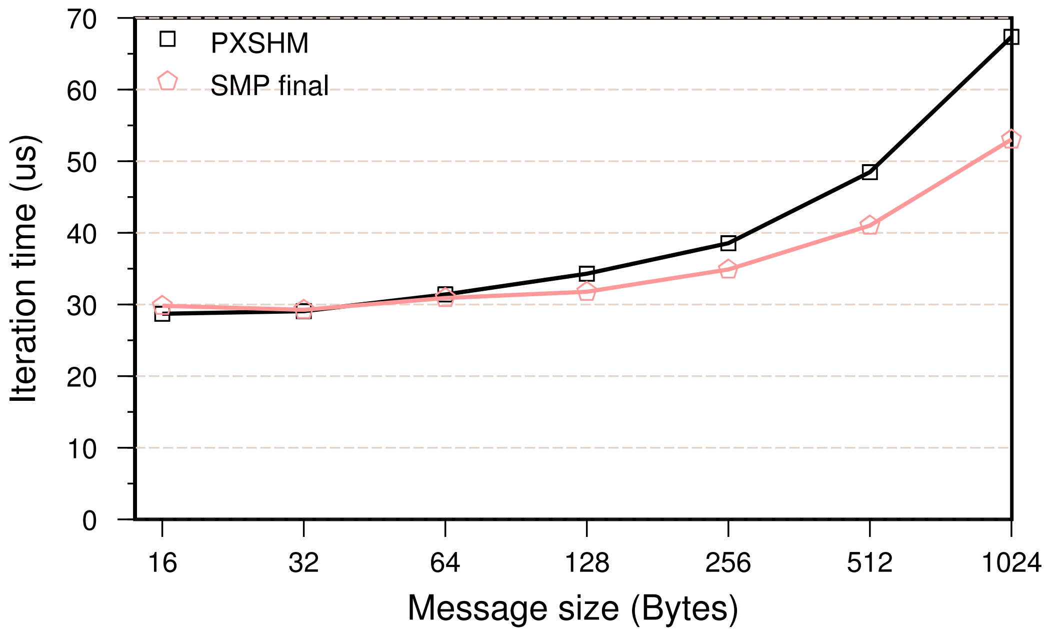

Figure 4.11: Performance Comparison between non-SMP Mode and Final SMP Mode

4.7 Performance Studies of Applications

In addition to synthetic benchmarks, we used two production-level scientific applications to demonstrate the performance impact of our optimization techniques described as follows:

- **NAMD:** Figure 4.12 shows the performance results of NAMD for the standard benchmark Apolipoprotein-A1 (ApoA1) molecule system on two multicore platforms C and E described in the section 4.1. On these two platforms, performance changes in NAMD, due to the switch in CHARM++ runtime mode from non-SMP to SMP and the aforementioned series of optimization techniques, are quite representative. On a platform having a smaller number of cores per node such as C, NAMD in SMP mode is better by 5.2% than it is in non-SMP mode. In contrast, on the platform that has a larger number of cores per node such as E, NAMD in SMP mode demonstrates more benefits as it beats the non-SMP one by 21.1%. Such difference in performance improvement is primarily due to the difference in the number of cores in the node. Since the SMP mode reduces the message latencies significantly within a node, a larger node size, implying more chances for an application to have messages sent within a node, will benefit more as demonstrated in this case. In addition, we can see from the figure that the optimization of having multiple queues mentioned in section 4.3.3 incurs a slight performance degradation for NAMD on the 24-core platform E because of the increased polling overhead for message queues. Finally, we noticed that using the OS-provided memory management instead of the old memory module as mentioned in section 4.3.1 alone contributes the most performance gains for both platforms.

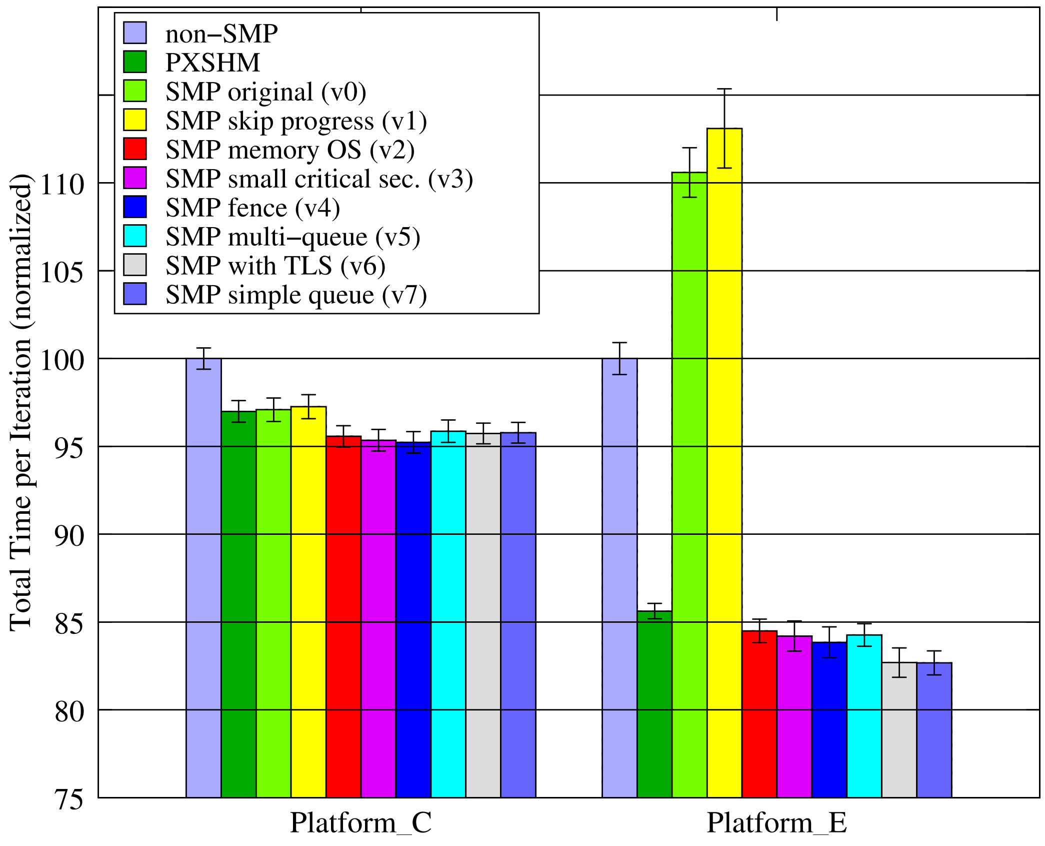

Figure 4.12: NAMD Performance on a Single Multicore Node

- **ChaNGa:** Figure 4.13 also shows two executions of ChaNGa with two different datasets on platform C. The first dataset consists of nearly five million particles highly clustered in the center of the simulation (dwf1); the second consists of about 110,000 particles uniformly distributed in space (cube300). The first system takes about 500 seconds to perform 3 iterations, while the second requires about 30 seconds to perform 5 iterations of the algorithm, and is more communication intensive. We can see that the performance of SMP was worse than non-SMP with the initial SMP version. As in NAMD, switching to the OS memory system provides the greater benefit to ChaNGa. This is due to two reasons: 1) since all threads allocate memory at the same time, by releasing the locks, the total time spent allocating memory is greatly reduced; 2) since the memory is allocated from separate pools for different threads, the resulted memory blocks are less spread in the address space, and accessing it is faster.

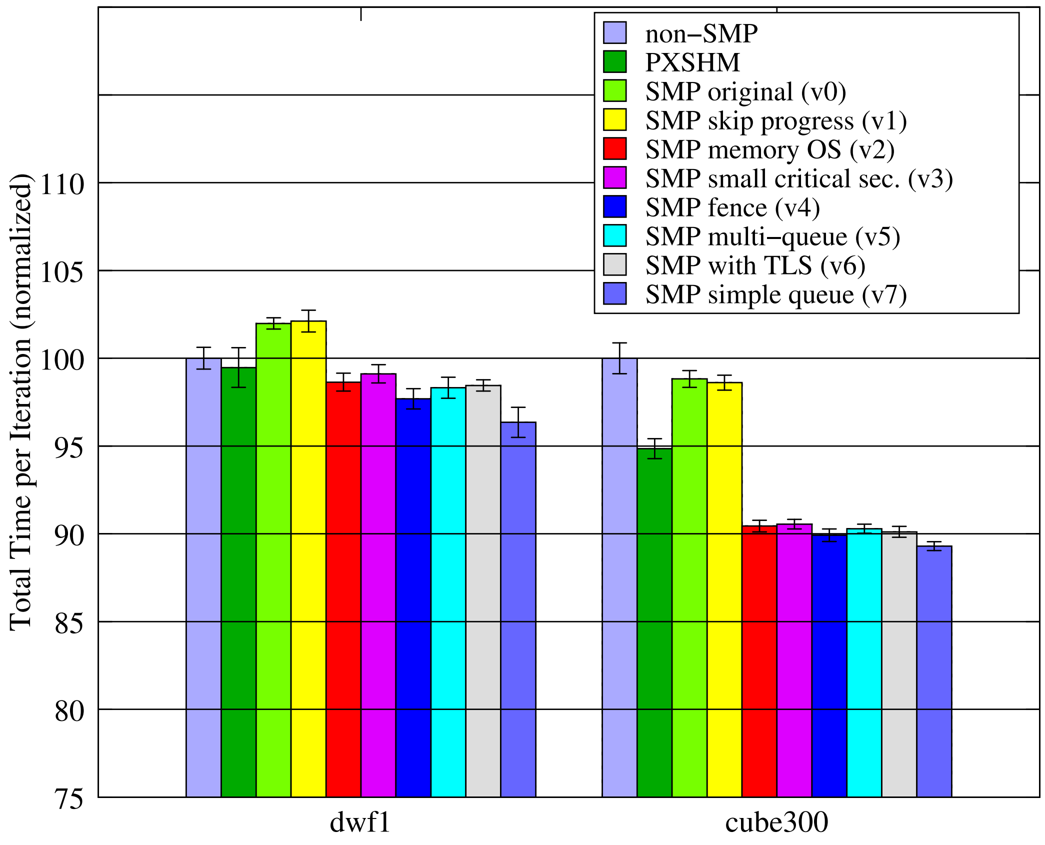

Figure 4.13: ChaNGa Performance on a Single Multicore Node

For the first dataset, the performance benefit is only 4% from non-SMP mode (6% from the original SMP implementation). This is mainly due to the fact that the majority of the time is spent computing forces, without any communication. For the smaller dataset, which constitutes a more typical computation/communication ratio when scaling simulations to large machines, the improvement is 6% from non-SMP mode with POSIX shared memory, and 11% when the processes are communicating through the OS kernel.

We can see that by simply switching from non-SMP (with or without POSIX shared memory) to SMP mode the performance is automatically boosted by 9%. When applying the optimizations described in this paper, another 2% is gained. As mentioned, ChaNGa has many internal optimizations for SMP mode, and therefore benefits less from the optimizations of the runtime system. However, the improvements just mentioned are still very significant considering that the application did not need to be changed to obtain them.

5 Exploiting Fine-grained Single-node Parallelism

In real scientific and engineering applications, different modules may be parallelized with different granularity in order to achieve a balance between computation and communication. In the case of evaluating the strong scalability of the application, one module may scale worse than others, thus becoming the performance bottleneck. To resolve this issue, a more fine-grained parallelism needs to be exploited within a node so that the network communication cost will remain the same. Additionally, we identified that the slight load imbalance in applications that occurs only under certain runs could be mitigated within a node. Therefore, in this chapter, we will explore how to handle such more fine-grained parallelism in the context of the multithreaded message-driven parallel language runtime system. Essentially, we will answer whether we could simply use an existing but different parallel language construct together with the multithreaded message-driven language runtime system.

5.1 Motivation

As mentioned above, a real application may consist of different modules. In some cases, a single computation work in a module is not initially worth being parallelized into very fine-grained computation tasks. This may be due to several reasons. First, it is possible that the computation work is not regarded as a bottleneck in the initial design phase, and the work does not affect the target of program scalability. Secondly, the work may require significant programming effort, which is not worth doing considering the expected performance improvement. However, when scaling the application to tens to hundreds of thousands of cores, such computation work becomes the performance bottleneck and requires further decomposition into more fine-grained tasks. Furthermore, a parallel computation component may have to strike a balance between the granularity of computation and communication. The computation could be parallelized into more fine-grained tasks to have its execution time reduced, but, at the same time, it may cause a significant increase in the network communication time hence making a negative impact on the overall performance.

One portion of the PME calculation [36, 37] in NAMD demonstrates such a use case illustrated by green bars in Figure 5.1. The figure is a timeline snapshot of a 100-million-atom molecule system simulation by NAMD in the CHARM++ SMP mode on 45,056 cores of JaguarPF. The green bars (those highlighted in the square) that last around 10ms clearly show a performance issue because while the processor is busy with the computation, its neighboring processors are idle. Through examining the codes, we found the computation mainly consists of parallelizable loops, and part of the codes could be restructured to form loop-level parallelism. So, idle neighboring processors could be utilized to distribute the computation load. In addition, we also identified that if such computation is parallelized across different nodes, the communication became much more expensive which could offset the performance improvement by the reduced computation cost. Therefore, such computation is ideally to be parallelized within a SMP node as it will not incur cross-node communication.

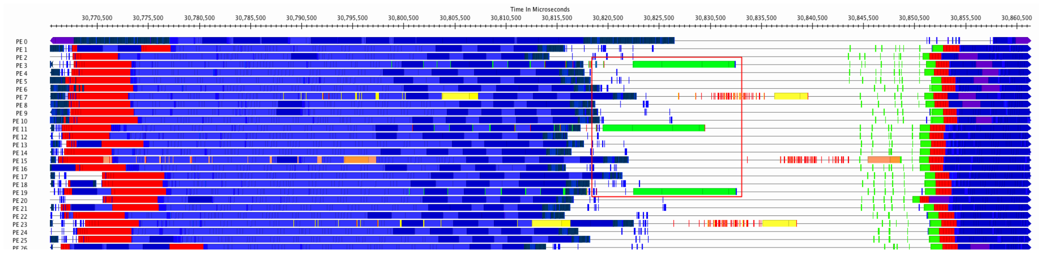

Figure 5.1: Snapshot of NAMD Timeline Indicating the Need of Exploiting Intra-node Parallelism

The slight load imbalance in an application that appears only in a run on a certain number of PEs also motivates us to exploit the more fine-grained single-node parallelism. Programming models that promote writing parallel programs irrespective of the underlying physical processors, such as CHARM++, enjoy the flexibility of enabling applications to run on any number of PEs instead of being bound to numbers of a specific pattern, say a square number of PEs. This flexibility turns to be very useful because of the fault tolerance requirement. As parallel machines become increasingly larger, the mean time to failure decreases accordingly, increasing the probability of a PE failure during execution. Therefore, for the application to tolerate the failure, it should be able to run any number of PEs. However, this flexibility may cause a slight load imbalance in applications on some certain number of PEs because of the number of computation tasks (i.e. parallel objects in the context of CHARM++) and the corresponding load cannot be evenly distributed. In other words, in this scenario, there is trailing computation only on some PEs. If such trailing computation could be parallelized within a SMP node, the slight load imbalance could be mitigated. As an additional challenge, the same computation may not always be the trailing one. For example, it may be the case on some PEs, but may not on others. This requires the conditional exploit of the single-node parallelism.

5.2 Problems with Using OpenMP

Motivated by the aforementioned scenarios that expose the performance improvement opportunities, we should find a way to incorporate fine-grained single-node parallelism in this new multithreaded runtime. More specifically, we should explore tools and methods for parallelizing the computation in the form of loops in this new runtime within a SMP node. The most common and the simplest approach to parallelize a loop in a node is to apply OpenMP [6]. The question arises: is it optimal to accomplish this task as exploiting the intra-node parallelism by using OpenMP in this case? We have identified the following two performance problems:

1. **Coordination between OpenMP runtime and the CHARM++ runtime**: Note that OpenMP runtime has its own set of threads independent of threads in CHARM++ runtime. The latter ones continuously, during execution without yielding, poll the message queue for being promptly responsive to incoming messages that trigger the computation on their each assigned cores. Obviously, without coordination between the two language runtimes, it is expected that threads from OpenMP runtime will contend resources with those from

the CHARM++ runtime. Such contention will reduce performance when using OpenMP to parallelize the loop-based computation.

2. **Interference between two OpenMP-loop tasks on the same node**: In CHARM++ programs, each PE may have very different control flow from other PEs. In a parallel run of a job, this implies the possibility that the computation bottleneck may have non-uniform distribution across all CHARM++ SMP nodes. For some nodes, it may exist on multiple PEs within a node, and the computation starts at roughly the same time. In the mean time, for some other nodes, the bottleneck may just exist on a single PE within a node. Generally, it is not common practice to apply OpenMP in such a case. The interference between two OpenMP loop parallelization tasks within a node may lead to very uneven performance.

A simple benchmark has been developed to illustrate these problems. In this benchmark, each PE has one CHARM++ object which could be configured at runtime to execute a simple loop as:

```
for(i=0; i<iterations; i++) result += sqrt(1+cos(i*1.75));
```

To prevent the loop from being optimized out by the compiler, the variable “result” has been declared in such a way that it could be used later in the program. It is clear this loop intensive of floating-point operations could be easily parallelized with OpenMP using compiler-supported “pragma” directives. We measure the sequential time of this loop and the corresponding parallelized time in OpenMP with 4 *OpenMP threads*. We use GCC 4.6.2 to compile the benchmark linking with AMD Core Math Library (ACML) 4.4.0, and test the benchmark on a single node of Hopper. In CHARM++ SMP mode, we fully subscribe the physical node with 24 threads as each thread is mapped to a distinct physical core. We set the base case as the execution time of this benchmark in CHARM++ non-SMP mode where only one PE is launched so that OpenMP threads will not be interfered by any other CHARM++ PEs. Therefore, comparing the performance between the SMP mode and the base case will shed light on the problem in the first aspect. Configuring the benchmark to have two objects run the loop simultaneously will let us understand the problem in the second aspect.

configuration	iterations=4000	iterations=8000
Sequential	189.12us	384.30us
One OpenMP loop w/o CHARM++ contention	59.18us	108.64us
One OpenMP loop w/ CHARM++ contention	81.25us	147.56us
Two OpenMP loops w/ CHARM++ contention	61.20us/102.92us	109.76us/184.96us

Table 5.1: Performance of OpenMP in CHARM++ SMP mode

Table 5.1 shows the results of this experiment. Comparing the execution time of the SMP case (indicated by the fourth row) with just one object running the OpenMP loop against the base case(indicated by the third row), the execution time increases by about 37.30% and 35.82% respectively in two test cases (i.e., one loop with 4000 iterations and another with 8000 iterations). It clearly demonstrates that OpenMP threads are interfered by the continuously running CHARM++ PEs in SMP mode. Therefore, without coordination between the OpenMP runtime and the CHARM++ runtime, simply using OpenMP in CHARM++ for fine-grained parallelization is not able to achieve the best performance. In the case of two OpenMP loops running simultaneously by two objects on the same SMP node, indicated by the last row of table 5.1, the execution time of one loop is close to the base case while the other one is almost doubled. It is not desirable to have such a significant imbalance in execution time. In short, it is not appropriate to directly use OpenMP in a multithreaded runtime because of the contention between the OpenMP runtime and the CHARM++ runtime as well as the interference between the two simultaneously-running OpenMP loops which may occur in real applications.

The above discussion focuses on the issues arising from using OpenMP in the general CHARM++ execution model as we launch a PE on every logical CPU. However, if we execute CHARM++ with OpenMP in the same way as we execute the hybrid MPI+OpenMP programs where logical CPUs are spared for OpenMP threads, obviously, the aforementioned issues will disappear. But is this a good way to handle the single-node parallelism in CHARM++? We also identified two problems with this approach:

- The percentage of computation that is performed on a single PE, and that could be parallelized by OpenMP may be quite small in the parallel program, such as the case with NAMD where the majority of computation performed by a parallel object is serialized. Therefore, it is waste to have logical CPUs spared for OpenMP threads under those circumstances, in which case, no CHARM++ PEs are running on those logical CPUs.

- Generally, the granularity of a parallel program decomposition is automatically controlled by the number of PEs as more PEs result in more fine-grained computation. Therefore, with the same number of cores, running the "hybrid" CHARM++/OpenMP programs will have more coarse-grained computation than the same "pure" CHARM++ program as cores spared for OpenMP threads are not considered to be PEs by the CHARM++ runtime. Since OpenMP parallelization of the larger-grained computation does not observe the data locality–a key performance factor on multicore platforms–while the CHARM++ one of the smaller-grained computation does, the hybrid program could suffer worse cache performance hence worse performance than the same pure one. This is illustrated by the experiment of a CHARM++ Jacobi2D code with major computation loops parallelized with OpenMP. The program is an iterative 2D-array stencil code with block 2D decomposition. In every iteration, each element sets its value to the average of its four neighbors with itself expressed as $A^{n+1}_{[i][j]} = (A^n_{[i-1][j]} + A^n_{[i][j]} + A^n_{[i+1][j]} + A^n_{[i][j+1]} + A^n_{[i][j-1]})/5$. On a Cray XT6 node that has two AMD Interlagos chips, we tested the Jacobi2D code with a 1024×768 double floating-point matrix. We launched 4 SMP nodes, each containing 6 PEs for the pure CHARM++ mode, while for the hybrid CHARM++/OpenMP program, we also launched the same number of SMP nodes, each containing just 1 PE but with 6 OpenMP threads. As the decomposition is related to the number of PEs, the block size is set to 128×128 in the pure CHARM++ program while the block size is set to 512×384 in the hybrid one. The average execution time for the pure one is 0.409 ms/step, about 64% faster than the hybrid one as 0.671 ms/step. As the OpenMP part works on the larger block size and parallelizes the computation with 1D-decomposition on the block (i.e. putting a *parallel for* directive on the outer loop that iterates through the slowest changing array index), it produced a worse memory access pattern in terms of reusing cache than that of the over-decomposed pure CHARM++ program which uses 2D-decomposition instead. In case the decomposition granularity does not change for the hybrid execution, then the overhead of parallelizing each smaller-grained computation with OpenMP, as well as the overhead of scheduling for each piece of computation concentrated on a single PE, will obviously degrade the overall program performance compared with the pure execution.

In short, the direct usage of OpenMP inside a multithreaded runtime such as CHARM++ does not lead to optimal performance. This suggests using a same software stack to manage the parallelism at different granularity, such as the user-level software framework "Lithe" [38]. Specifically, in this case, the OpenMP threads and the CHARM++ threads that represent PEs have to be coordinated to perform each computation of different granularity.

5.3 A Unified Runtime to Exploit Single-node-level Parallelism

To resolve the problems described in the previous section, we use existing CHARM++ PEs to execute the parallelized loop computation instead of spawning OpenMP threads. Therefore, the coordination among different threads from the two runtime systems is totally unnecessary. In other words, we develop a library, named "*CkLoop*", that supports OpenMP loop parallelization on top of the CHARM++ runtime so that it becomes a unified multithreaded runtime which handles both the fine-grained parallelism within a node (as the targeted loop-level parallelism in this thesis) and the relatively coarse-grained parallelism across different nodes.

The *CkLoop* library simply has two APIs shown as follows:

```
/* Initialize the library, only need to be called once */
1. CkLoop_Init();
/* The function call to parallelize the code within a node */
2. CkLoop_Parallelize(funcPtr, /*ptr to the execution of a chunk*/
                       int argc,void *argv, /*args to func*/
                       int low,int high,int step, /*the loop info*/
                       int sync, /*whether doing implicit barrier*/
                       int redOp,void *redBuf /*for reduction*/
                      )
```

As shown by the above APIs, the *CkLoop* only supports the parallelization of a loop and a limited number of reduction operations such as floating-point related arithmetic operations because of the practical requirement from applications we have experimented with. In particular, slightly different from OpenMP which requires implicit barrier after each parallel region, *CkLoop* provides an option of skipping this barrier depending on a user-specified "sync" flag. Currently, developers will have to do the code transformation manually to parallelize the loop using this library. However, with programming directives, compilers can help automatically transform codes similar to how OpenMP is supported.

We have explored two different ways for the *CkLoop* implementation, each corresponding to static scheduling and dynamic scheduling as provided by OpenMP respectively. Considering the computation in the form of loop is parallelized into M chunks where each chunk will contain approximately the same amount of loop iterations, and those computation chunks are to be distributed onto P PEs within a node, the two schemes for implementation are described as follows:

- **Static Scheduling Scheme**: M chunks will be distributed in blocks so that each PE will have approximately M/P chunks to execute. To trigger the execution of each block of chunks on PEs other than the PE where the parallelization is initiated, each block of chunks is encoded into a message (i.e., an entry method associated with a special CHARM++ object) as CHARM++ adopts the message-driven execution where each PE is continuously polling incoming messages. Specifically, each message will be encoded with the range of loop iterations each chunk corresponds to and a *task descriptor* that represents the overall parallelizable loop including a function pointer to the code that executes a chunk of computation. Since we are only considering the loop parallelization in the same memory address space, the task descriptor is shared among all P PEs.
After messages for every other chunk have been constructed, they will be delivered to every other $P - 1$ PEs respectively. The PE originating the parallelization will first execute the block of chunks assigned to itself, and then the PE must wait until all blocks of chunks have been executed before it proceeds to the next statements after the loop. This is similar to the implicit barrier imposed after the OpenMP parallel region.
Because we are dealing with fine-grained parallelism for the best possible performance, in-

stead of more complicated schemes which involve awaking/suspending underlying threads, we use a counter-based busy-waiting scheme. Basically, we put the counter in the task descriptor so that each PE has access to it. The counter is incremented when each PE finishes its assigned computation. The originating PE will do a busy-wait, continuously checking the counter value until it equals to M.

- **Dynamic Scheduling Scheme**: each chunk is encoded into a message that is similar to the one used in the static scheduling scheme. Then, each such message is pushed into the CHARM++-provided general node-level message queue which is shared among all PEs on a SMP node. When a PE becomes idle, it will start to poll this shared node-level queue. If there is a message, the PE will pop the message out of the queue and then execute the associated computation. After the originating PE pushes all M messages into the queue, it will start to poll this node-level queue continuously and execute the obtained chunk. It will keep polling the shared node-level queue until all M chunks have been executed, which is determined by the same counter-based busy-wait scheme as described in the static scheduling scheme.

Both schemes are suitable if all PEs on the same node except the one that initiates the parallelization work are idle when the parallelization happens. However, this cannot be always guaranteed, especially in applications such as NAMD which consist of different computation modules. The execution of those modules are mixed across all PEs. If one computation inside one module becomes the performance bottleneck, it is possible this computation has some overlap with another computation of other modules on a SMP node. For example, in NAMD when the PME module becomes the bottleneck, a part of its execution overlaps with computation of other modules (such as the non-bonded computation) on some SMP nodes, while the same part has no overlap on other SMP nodes (i.e., neighboring PEs on the same node are idle when this computation happens). If the static scheduling scheme is used under such scenarios, the PE that has other computation will not pick up the message that triggers the execution of the parallelized task until the current computation finishes. Therefore, the completion of the parallelized loop is delayed, and the CPU time is wasted in the busy-waiting period on the originating PE. In contrast, the dynamic scheduling scheme fits this situation better. Only idle PEs will check the node-level queue and execute the chunk. The completion time of the parallel loop, as a result, does not depend on busy PEs that are working on other computations. Based on the consideration above, the dynamic scheduling scheme is selected as the default one when parallelizing the computation within a node in CHARM++ runtime.

However, the dynamic scheduling scheme has its own potential problems. First, since the node-level queue is modified (as messages are popped out of the queue) by multiple idle PEs, there is a contention overhead associated with the scheme, especially when the number of idle PEs are large. Secondly, the internal overhead from the CHARM++ runtime, such as the implicit message construction for *parameter marshalling* [39], the overhead caused by scheduling a node-level message etc., cannot be ignored if the computation bottleneck targeted to be parallelized is in the range of hundreds of microseconds. In the following section, optimization techniques are presented to reduce the various overheads of the dynamic scheduling scheme.

We also address the problem of exploiting the single-node parallelism conditionally as motivated in the end of section 5.1 by detecting if there are any idle PEs within a node. In the scheduler of every PE, we add a PE-private variable that indicates the state the of PE as whether it has no incoming messages (i.e., in idle) or it is performing the computation associated with the message. The shared memory address among PEs on a SMP node allows this status variable to be accessed by other PEs on the same node. As a result, we implement a function that queries this status variable of every other PEs on a SMP node to check whether there are idle PEs. If the function returns true as there are idle PEs, we think it is worth exploiting the single-node parallelism at the time in spite of the possibility there are incoming messages in a very short time on idle PEs. Tested on a SMP node with 16 PEs, this query function costs very little overhead, as around 200ns per query in the worst case when every other PE is busy with work.

5.4 Scheduling Optimizations for the Unified Runtime

To measure the overhead of the dynamic scheduling scheme, we still use the benchmark mentioned before which contains the following parallelizable loop:

`for(i=0; i<iterations; i++) result += sqrt(1+cos(i*1.75));`.

The overhead is calculated as

$$\sigma = T_p - T_s/P \tag{5.1}$$

where σ represents the overhead of the scheme, T_p is the parallel execution time of the loop, P is the number of PEs used for parallelization and T_s is the sequential execution time. To make the loop parallelization beneficial as $T_p < T_s$, it is clear that the overhead (σ) should be reduced as much as possible. With equation 5.1, we derive the following relationship between the overhead and the sequential execution time indicating when it is worth parallelizing a loop that takes T_s to finish sequentially across P PEs.

$$T_p < T_s \Leftrightarrow \sigma + T_s/P < T_s \Leftrightarrow \sigma < T_s * (1 - 1/P) \Leftrightarrow T_s > \sigma * P/(P-1) \tag{5.2}$$

For example, if it takes 100us to execute a loop, to make parallelizing this loop beneficial on 4 PEs, then the maximum overhead is $100 * (1 - 1/4) = 75$us. On the other hand, if the overhead of this parallelization scheme is 6us on 4 PEs, then any loop that takes longer than $6 * 4/(4 - 1) = 8$us could be parallelized to achieve better performance.

#threads	4	8	12	16	20
Base	16.17	53.96	132.09	227.52	433.61
OpenMP w/ CHARM++ Contention	21.77	39.81	62.19	81.01	96.06

Table 5.2: Initial Overhead (us) of *CkLoop* Library

On a single node of Hopper, table 5.2 shows the initial overhead of the dynamic scheduling scheme that uses the general node-level message queue provided by CHARM++ runtime as well as the overhead of using OpenMP with CHARM++. According to table 5.2, we can see the overhead of both the initial implementation of the scheme and the OpenMP version increase significantly with the number of threads. In SMP mode of CHARM++, since each thread corresponds to a PE, the number of threads is equal to the the number of PEs per node (simply abbreviated as *ppn*). In the case *ppn* is 8, the overhead of *CkLoop* becomes worse than that of OpenMP. When *ppn* is increased to 12, the overhead of the scheme exceeds 100us! In NAMD, when PME becomes the bottleneck, its computation usually takes about, at most, hundreds of microseconds, some of which only takes less than 100us. Therefore, it is necessary to reduce the execution overhead of the dynamic scheduling scheme.

First, we identified that the general node-level queue provided by CHARM++ has additional overhead because of some features that are not needed in this case, and those features result in intensive lock contention. Therefore, we implemented a simplified node-level queue tailored for this dynamic scheduling scheme. Table 5.3 shows the significant overhead reduction after this optimization. We can see that the larger the *ppn* value is, the more overhead reduction is achieved by this optimization, mainly due to less lock contention.

#threads	4	8	12	16	20
Base	16.17	53.96	132.09	227.52	433.61
Simplified Node-level Queue	7.10	20.23	31.31	42.08	51.57
Speedup	2.28X	2.67X	4.22X	5.41X	8.41X

Table 5.3: Overhead (us) after Using a Simplified Node-level Queue

Secondly, as mentioned in section 4.3.1, the memory allocation is more expensive in the multithreading environment. We find there are implicit memory allocation for sending each message that triggers the execution of a chunk of loop iterations. So, we pre-allocate memory buffers for those messages and recycle the memory space of those messages when the PE finishes processing the chunk of computation. By performing this optimization, we also remove additional small overheads with regard to message scheduling in CHARM++, such as the extraction and look-up of message handler, message rescheduling according to message priority, etc. Table 5.4 shows the moderate improvement in reducing the overhead of the implementation of the dynamic scheduling scheme after recycling the message memory buffer.

#threads	4	8	12	16	20
Simplified Node-level Queue	7.10	20.23	31.31	42.08	51.57
Recycling Message Memory Buffer	5.26	17.53	25.66	34.49	43.57
Speedup	1.35X	1.15X	1.22X	1.22X	1.18X

Table 5.4: Overhead (us) after Recycling Message Memory Buffer

Furthermore, in the initial design, each message that triggers the execution of a loop chunk will be populated with the range of iterations this chunk corresponds to, and then pushed into the node-level queue. This work is done sequentially only on the PE where the parallelization task is initiated. To remove this sequential bottleneck, we distribute the calculation of the loop iteration range onto the PEs that will execute the chunk. Specifically, the message now only contains the *task descriptor*, but inside the descriptor, a new integer "index" is added to represent the last chunk that has been executed. The idle PE first atomically increments this "index" to obtain the index of the chunk it will work on, and then calculates the range of loop iterations based on this index value. With this distributed calculation scheme, we removed lock operations that are required to operate on the node-level queue. In addition, the node-level queue that are shared by all parallelization tasks is disintegrated into separate "micro" node-level queues as represented by each *task descriptor*. Both changes reduce the resource contention on shared resources in the multithreading runtime. As shown by table 5.5, this optimization also significantly reduces the overhead, particularly when the SMP node size is large, mainly owing to the replacement of lock operations with atomic operations.

#threads	4	8	12	16	20
Recycling Message Memory Buffer	5.26	17.53	25.66	34.49	43.57
Separate Micro Node-level Queue	2.30	4.15	6.19	7.46	9.24
Speedup	2.29X	4.23X	4.15X	4.62X	4.71X

Table 5.5: Overhead (us) after Using Separate Micro Node-level Queue

Finally, considering the number of PEs per SMP node could be large, instead of sending the computation-triggering message to every other PEs just by the originating PE, we adopt the sending scheme using spanning tree if the SMP node size exceeds a threshold (with default value to 8). Table 5.6 shows that there are more benefits of such notification via spanning tree in reducing overhead when ppn is larger. This is expected as the cost of the original notification scheme is linear to the ppn value as the originating PE has to send $ppn - 1$ messages while the cost is distributed among intermediate PEs in the spanning tree scheme. However, it is possible that an intermediate PE of the spanning tree is busy with other computation. In this case, the message will not reach every idle PEs on a SMP node in time, thus limiting the benefits of parallelizing the computation. To attack this issue, it

#threads	4	8	12	16	20
Separate Micro Node-level Queue	2.30	4.15	6.19	7.46	9.24
Notification via Spanning Tree	2.30	4.16	5.78	6.67	7.98
Speedup	1.00X	1.00X	1.07X	1.12X	1.16X

Table 5.6: Overhead (us) after Using Spanning Tree for Notification

is desirable to have an adaptive scheme of spanning tree creation. Although the adaptive scheme is possible to implement in CHARM++, the overhead of such adaptivity is too high to be useful.

In summary, table 5.7 shows the overhead of different schemes to exploit the single-node parallelism along with the increase of ppn values or the number of OpenMP threads. In the table, the second row shows the significant overhead of the initial implementation

#threads	4	8	12	16	20
Base *CkLoop*	16.17	53.96	132.09	227.52	433.61
Optimized *CkLoop*	2.30	4.16	5.78	6.67	7.98
OpenMP w/ CHARM++ Contention	21.77	39.81	62.19	81.01	96.06
OpenMP w/o CHARM++ Contention	11.51	33.97	58.03	76.86	88.06

Table 5.7: Final Overhead (us) of *CkLoop* Library

of the dynamic scheduling scheme while the third row illustrates the drastic improvement after applying the optimizations described in this section. Comparing this with the direct use of OpenMP as shown in the last two rows, it is clear that the scheme we developed has far less overhead. In addition, the data of the last two rows once again demonstrate the performance issue arising from the lack of thread coordination between CHARM++ PEs and OpenMP threads as the plain OpenMP scheme has less overhead. Figure 5.2 illustrates the overhead reduction rate based on the final dynamic scheduling scheme. Generally, our final optimized scheme is more advantageous with more threads.

It is quite surprising that the case of OpenMP without contention from CHARM++ also suffers much more overhead than that of our scheme *CkLoop*. We have found that it is related to the way of launching the OpenMP program. The performance data in the last row of table 5.7 are obtained by launching the program using the same way as launching a parallel job by using the "*aprun*" command. However, if we execute the OpenMP program simply by calling its binary name, then the overhead of OpenMP without CHARM++

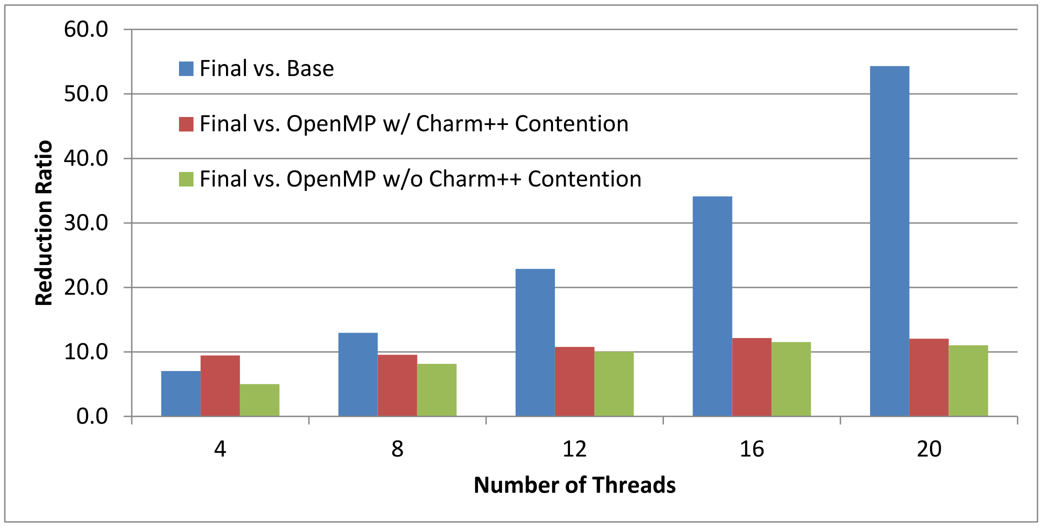

Figure 5.2: Comparison of Overhead Reduction with the Final Dynamic Scheduling Scheme

contention is almost identical with that of the optimized *CkLoop* as shown in the third row of table 5.7. Although we are still unclear about why the OpenMP performs much better without being launched by "*aprun*", it again shows the performance issue of the lack of coordination between OpenMP threads and those threads that represent CHARM++ PEs and the necessity of a unified runtime system to support different levels of parallelism.

5.5 Evaluation of Application Performance

In this section, we evaluate the effectiveness of this dynamic scheduling scheme for integrating the fine-grained single-node parallelism into the CHARM++ runtime. First, we present the performance results obtained from the simple loop benchmark. Afterwards, we show the effectiveness of this scheme in the Jacobi2D program and the PME computation of NAMD including the impact on the overall performance of NAMD.

#threads	4	8	12	16	20
CkLoop	49.93	28.57	22.80	19.51	18.76
OpenMP w/ CHARM++ Contention	81.39	65.09	75.60	103.74	114.61
OpenMP w/o CHARM++ Contention	60.13	57.16	65.33	78.37	92.59

Table 5.8: Parallel Performance of a Loop in Different Schemes

To begin with, we perform the test of our scheme *CkLoop* and compare it with the results of using OpenMP on the simple parallelizable loop as:

`for(i=0; i<N; i++) result += sqrt(1+cos(i*1.75));.`

We set $N = 4000$ and the serial loop performance is 189.12us. As shown by table 5.8, *CkLoop* achieves much better parallel performance. The OpenMP ones degrade when the number of threads reaches 8 or beyond while the scheme we describe in this chapter always results in a decreasing parallel execution time. Figure 5.3 clearly illustrates the scalability of different schemes in exploiting the single-node parallelism in the CHARM++ runtime.

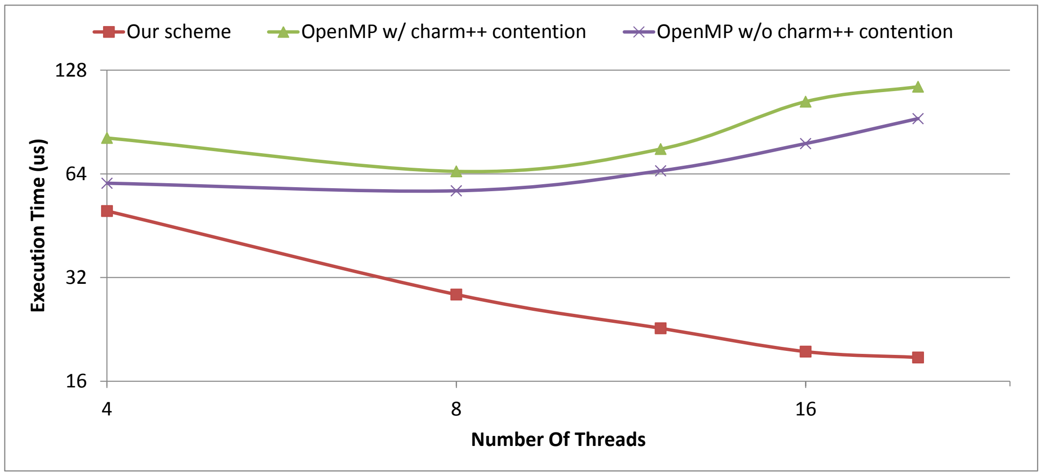

Figure 5.3: Scalability Comparison of the Parallelized Loop Performance in Different Schemes

Secondly, we evaluate how *CkLoop* performs in case there are two simultaneous loop instances that require to be parallelized in response to the second problem mentioned in section 5.2. Table 5.9 shows our evaluation of two schemes with different number of threads for parallelizing the same loop that iterates 4000 times. In the test, two loops are placed on two immediate neighboring PEs, and each loop is partitioned into chunks with the same number of threads. The "Diff." columns in the table clearly show that *CkLoop* gives much lower absolute difference in the execution time of the two loops than the OpenMP scheme does. This means our scheme of exploiting single-node parallelism incurs less load imbalance in the presence of multiple simultaneous instances. Analytically, given n threads and the loop partitioned into n parts, without any other simultaneous loop instances, the ideal execution time is t/n. Since we now have two such loop instances that are simultaneously parallelized with n threads and with the even distribution, one thread will have to execute two chunks of a loop while other threads will just have to execute one chunk of the loop. As a result, each parallelized loop is expected to finish in $2t/n$ and perfectly overlap each other. For example, taking $n = 4$ as an example, the loop where $N = 4000$ is divided into 4 chunks, hence each taking $189.12/4 = 47.28$us. Therefore, the expected execution time of the loop in this case is 94.56us, very close to the values we present in the table.

#threads	*CkLoop*		Diff.	Using OpenMP		Diff.
	Loop Inst. 1	Loop Inst. 2		Loop Inst. 1	Loop Inst. 2	
4	109.66	103.68	5.98	81.60	264.88	183.28
8	50.69	66.72	16.04	64.93	238.51	173.59
12	43.56	41.90	1.66	76.76	238.80	162.04
16	28.81	37.08	8.27	96.11	242.41	146.30
20	28.29	28.54	0.25	114.03	248.43	134.40

Table 5.9: Parallel Performance Comparison of Two Simultaneous Loop Instances in Different Schemes

Furthermore, we use Jacobi2D program as mentioned in section 5.2 to evaluate the effectiveness of exploiting the single-node parallelism conditionally. Since the main computation kernel of the Jacobi2D program is performed on every PE, we cannot statically determine which kernel will be the trailing computation in priori. In addition, the number of blocks the problem is partitioned into can be easily configured in CHARM++ as each block will correspond to a parallel object. Therefore, Jacobi2D is a good test case for this purpose. We performed the test on a 2-socket Intel Xeon E5520@2.27GHz quad-core node (8 physical cores in total) with $ppn = 7$ (i.e. exploiting single-node parallelism with 7 threads). Table 5.10 shows the results with fixed block size 300×300 but with different total number of blocks. In the table, the second row shows the number of trailing

Grid size	15 (3X5)	20 (4X5)	25 (5X5)	30 (6x5)
#trailing computation kernels	1	6	4	2
Plain	2.66	3.00	4.00	4.73
w/o conditional *CkLoop*	2.56	3.48	4.43	5.40
Speedup over plain	4.06%	-13.67%	-9.75%	-12.48%
w/ conditional *CkLoop*	2.18	3.05	3.89	4.62
Speedup over plain	18.28%	-1.70%	2.78%	2.31%

Table 5.10: Performance (ms/step) of Exploiting Single-node Parallelism Conditionally in Jacobi2D

computation kernels in this experiment. For example, when the number of blocks is 25, evenly distributed to 7 PEs, then four of them will have 4 blocks and the other three will have 3 blocks. Therefore, four PEs will perform the trailing computation block after every PE completes three computation blocks. Obviously, a slight load imbalance is caused by the trailing computation. The third row shows the plain CHARM++ performance without exploiting the single-node parallelism in the computation kernel. Since the trailing computation is not statically known in priori, one way of parallelizing the computation is to do it blindly by parallelizing every computation during execution. However, it slows down the execution in most cases as shown in the fourth row. In contrast, it is beneficial to do it on demand only in case there are idle PEs on the same node in order to save parallelization overhead. Demonstrated by the last two rows of the table, the conditional approach achieves better performance than that of plain CHARM++. We notice that when there is more trailing computation, we gain less benefit from exploiting the single-node parallelism in the computation due to the overhead associated with each parallelization.

System	#Phy Nodes	w/o *CkLoop*	w/ *CkLoop*	Speedup (%)
DHFR	64	2.01	1.80	11.11
	128	1.76	1.56	12.82
Apoa1	64	3.11	3.02	2.98
	128	2.32	2.16	7.41
1M-atom STMV	512	6.31	5.94	6.23
	1024	4.69	4.47	4.92

Table 5.11: Performance (ms/step) of Exploiting Intra-node Parallelism (PME every timestep) in NAMD with 7 PEs per SMP Node on Titan

Finally, after applying this optimization to the PME part of NAMD, we obtain a decent performance improvement on simulating three typical molecular systems as shown in table 5.11 when the PME becomes performance scaling bottleneck. As mentioned before, to strike a balance between computation and communication, we have chosen to place at most one PME object per SMP node. As a result, the total PME computation will be fixed on each node beyond a certain scale, implying there is a theoretical upper limit in the reduction of PME computation time we can achieve per node.

6 Investigation of Multi-node Runtime Performance

After addressing performance issues and exploring the corresponding optimization techniques for the multithreaded parallel language runtime system on a single SMP node (i.e., all control flows sharing the same memory address space), this thesis continues to investigate performance issues of the runtime arising from runs on multiple SMP nodes. It is well known that the communication performance in parallel applications is critical to the overall performance as all high-end massively parallel machines are distributed-memory, and compute nodes are connected by high-performance network interconnects for data exchange. Therefore, this chapter primarily focuses on studying the "inter-node communication" performance in the multithreaded runtime. For simplicity, we generally refer to "inter-node communication" as "communication" in this chapter unless it requires explicit description. In sum, we considers following communication-related issues in different aspects of a multithreaded parallel programming language runtime in this chapter:

- *What choices are available for the multithreaded runtime to handle computation and communication, and which one is preferred in practice?* This issue is motivated by the fact that there is freedom in the runtime to assign computation work and communication work to different threads because they are in the same memory address space. See section 6.1 for details.

- *How to optimize the usage of MPI as the communication substrate for the message-driven language runtime?* As mentioned in chapter 2, key differences lie in the message passing between MPI and the message-driven CHARM++. We identified that such differences lead to worse performance in the deliverable communication latency when using MPI for message-driven runtime system. See section 6.2 for details.

- *What are the communication performance issues in the multithreaded runtime?* And how to address them? In general, using a performance analysis and visualization tool facilitates the detection of communication performance problems. So, first we will check how the design of this multithreaded runtime will affect the usage of the tool associated with the parallel language. We extend such a tool *Projections* [40] associated with CHARM++ to support the SMP mode better as detailed in section 6.3.1. Then we will analyze the performance issues in section 6.3.2 and propose optimization techniques to address those issues in section 6.3.3.

- *How can we leverage the capability provided by a multithreaded runtime in applications to achieve better performance?* We will examine two aspects in section 6.4.2. Firstly, we will exploit the language construct to improve the performance of applications with minimal changes. Secondly, we will investigate how to further improve the asynchronous collective communication performance especially, when it is on the performance critical path. The asynchronous reduction is examined as an example.

The organization of this chapter follows the order of the issues listed above. A portion of this work has been published in paper [24].

6.1 Design Exploration in Handling Communication Among Threads

Since CHARM++ PEs on a SMP node (PE is implemented with a thread in this case) share the same memory address space, we have the freedom in assigning the computation work and the communication work to different threads. In this section, we will discuss the choices of handling communication and computation in the multithreaded runtime, and analyze the impact of those choices on the performance of the runtime system.

6.1.1 Options in Assigning Communication to Threads

Clearly, there are two types of work–computation and communication in the runtime that threads on a SMP node need to perform. In other words, there exists two types of threads on a node: one performs computation work and the other carries out communication work. However, it is not necessary those two types of threads perform work exclusively. Formally, suppose one SMP node has $M = M_1 + M_2$ (where $M_1 > 0$ and $M_2 > 0$) threads, we have three options in computation and communication work assignment:

1. **All M threads perform computation and communication alternatively:** This option has the least difference from the non-SMP mode since each thread has the exact same work flow as each OS process does. The biggest difference lies in the thread-safety requirement of the lower-level communication library used for message passing. In non-SMP node, the library does not need to be multithread-safe because there is only one control flow accessing the library. In contrast, in SMP node, M threads will possibly access the communication library at the same time, thus requiring multithreaded support. If the library is not multithread-safe, then mechanisms of atomic operation, such as using locks for protection, may have to be applied to every function call to the communication library in the runtime to ensure its integrity.

2. **M_1 threads perform computation, but the communication work is separated between M_1 and M_2 threads:** Different from the first option, we first limit the computation only to M_1 threads instead of all M threads. Secondly, we separate the communication work into two parts assigned to M_1 threads and M_2 threads. M_1 threads involve the message sending operations, while M_2 threads involve polling incoming messages along with the message receiving operations. After the message is received, it will be forwarded to one of the M_1 threads where the associated entry method (i.e., computation) is triggered. Obviously, this option also requires communication operations to be thread-safe. However, depending on the implementation of the lower-level communication library used in the runtime system, it is possible that the message sending part has no conflicting data accesses with the message receiving part. As a result, the thread-safety management on function calls to the library could be limited to M_1 and M_2 threads respectively.

3. **M_1 threads only perform computation while M_2 threads only perform communication:** This option is similar to the second one, however, the biggest difference lies in that the computation and communication are totally separated into two different sets of threads. As a result, all communication operations are funneled through M_2 threads. In other words,

when sending network messages from M_1 threads, the data have to be first forwarded to M_2 threads and then get sent out. In terms of receive incoming network messages, it only takes place on M_2 threads. The message then gets forwarded to one of the M_1 threads to trigger the associated computation. Similarly, the message-passing operations are required to be thread-safe except the case when M_2 is 1.

6.1.2 Analysis of Different Options for Communication Assignment

With the three different options described in the previous section, we are going to analyze the pros and cons among those three options in the following aspects:

- **Performing the computation and communication work in separate threads:** Firstly, as mentioned in section 4.2, letting every thread perform communication operations (including polling the network) results in a high overhead for intra-node communication. From this perspective, it is better to separate the computation and communication work into different threads.

Secondly, the communication work performed by the CHARM++ PE involves almost no floating-point operations, but mainly operations on the memory space internally managed by the lower-communication library. The characteristics of such work are very different from the computation work performed by the CHARM++ PE which may involve floating-point-intensive computation or memory-intensive operations that touch a much larger memory space allocated by the application. As a result, alternating these two different types of work on a PE in the non-SMP mode or in the first design option of the SMP mode could lead to cache interference in each type of the work. Demonstrated by the following experiment, we conclude that separating the computation work from the communication operations by the lower-level communication library gives better cache performance because cache pollution is prevented from the communication work for the computation work and vice versa.

We have written a synthetic benchmark that every PE has a chare object, in which half of them (referred as Type A) will loop over a computation-intensive entry method with random accesses to a memory region of 24KB private to the PE, and the other half (referred as Type B) perform similar computation but without random accesses to the 24KB memory. We ran

(a) L1/L2 Data Cache Misses for Type A Objects

	comp/comm separated	comp/comm not separated	reduction percentage
L1 DCM	749	833	11.21%
L2 DCM	322	400	24.22%

(b) L1/L2 Data Cache Misses for Type B Objects

	comp/comm separated	comp/comm not separated	reduction percentage
L1 DCM	125	203	62.40%
L2 DCM	97	150	54.64%

Table 6.1: Comparing L1 and L2 Data Cache Misses for Separating the Computation and Communication Work

the benchmark in two modes on a two-socket Intel Xeon E5520 node (8 physical cores in total). The first mode involves mixing function calls to the lower-level communication library (MPI_Iprobe specifically acting for the polling incoming network messages) with the computation (i.e., representing the control flow of a PE in the non-SMP mode). The second mode, in contrast, runs without any communication operations (i.e., free of MPI_Iprobe representing the control flow of a PE that only performs the computation work mentioned in the third design option). We collected the the average number of of L1 and L2 data cache misses of different types of chare objects in each iteration in table 6.1. We can see that, for both types of objects, the number of L1 and L2 cache misses is reduced in the case where the communication and the computation work is separated. Comparing the reduction percentage in both table 6.1(a) and table 6.1(b), we can find that more reduction in cache misses is witnessed when the computation work incurs fewer cache misses as illustrated by type B objects. When the computation work causes more cache misses, the cache interference between the lower-level communication library and the computation work is less obvious, implying the cache benefit of separating the two different types of work is reduced as shown by type A objects.

Finally, considering alternating between computation and communication within one thread, the execution of computation may prevent a prompt service to an incoming network message if it arrives in the middle of the computation work. If the message happens to be on the performance critical path, the delay in processing this message may trigger a ripple effect so that the final performance of the application is degraded. Therefore, having separate threads dedicated for communication work could improve the responsiveness of programs to network messages. Taking advantage of this design option could help asynchronous communication, which will be presented later in this thesis. However, as described later, separating the computation work and the communication work has its own disadvantages in two aspects: one is the loss of computation resources if it is a one-to-one mapping between the thread and the underlying physical core, and the other is the overload of work on the threads dedicated to communication.

- **Limiting the communication work in a subset of threads:** all three options require thread-safe operations on the lower-level communication library except the case of single communication thread mentioned in the third option. Therefore, the way to ensure the atomicity of those operations affects the communication performance of the multithreaded runtime. For the communication libraries we have experience with, such as DCMF for BlueGene/P [41], uGNI for Cray [42], ibverbs for Infiniband [43], LAPI [44] and MPI, the thread-safety of function calls to those libraries is either not guaranteed or not efficiently implemented, such as the approach to using a global lock on every call internally. Consequently, based on our experience above, the first option potentially creates many spots in the runtime for synchronization contention among M threads, particularly in polling incoming network messages. In comparison, in the third option, the contention is limited to M_2 threads, reduced from M threads. As for the second option, the contention on communication library calls may still involve M threads if a global lock is used. But if separate locks are used for sending and receiving operations, then the contention could be limited to M_1 and M_2 threads respectively. From the performance perspective of such contention, limiting the communication work in a subset of threads instead of performing the communication on all M threads is a better design option.

Furthermore, historically, communication libraries have not generally been designed for use in a multithreaded parallel runtime. Generally, only one communication context is allowed to be launched per OS process, and communication only happens between two communication contexts. This means that the sending and receiving messages by the lower-level communication library is handled in the unit of a SMP node, not in the unit of a thread on a SMP node. Therefore, any thread in the set of M_2 threads that polls for incoming network messages could receive a message that is supposed to trigger computation on another thread in the set of M threads in the first option or one of M_1 threads in the second and third options. By forwarding the received message, a producer-consumer relationship is formed among the threads in a SMP node. If $M_1 > 1$ and $M_2 > 1$, such relationship turns to be a multi-producer-multi-consumer model which generally requires using locks in implementation, thus introducing another source of contention in the runtime. Therefore, we conclude that the number of threads that check the network progress (i.e., polling for incoming messages) should be very limited. Clearly, the smallest value of M_2 is 1.

It is worth noting that the overall computation capability per SMP node is reduced because a subset of physical cores are totally dedicated to the communication work in the second and third option. Such loss could be calculated as $M_2/M * 100\%$, because it is a one-to-one mapping between threads and logical CPUs. Therefore, the performance in SMP mode is potentially worse than that of non-SMP mode because of this loss in computation capability. When it happens, the scenario can be analytically expressed as:

$$\begin{aligned} T_M^{nonsmp} < T_{M_1}^{smp} &\Rightarrow T_1/(M \cdot E_M^{nonsmp}) < T_1/M_1 \cdot E_{M_1}^{smp}) \\ &\Rightarrow M \cdot E_M^{nonsmp} > M_1 \cdot E_{M_1}^{smp} \\ &\Rightarrow E_M^{nonsmp}/E_{M_1}^{smp} > M_1/M \\ &\Rightarrow E_M^{nonsmp}/E_{M_1}^{smp} > 1 - M_2/M \end{aligned}$$

where E_i stands for the parallel efficiency which is calculated as $E_i = T_1/(i \cdot T_i)$ and T_i stands for the execution time when i processors are used.

Based on the above expression, we can conclude that when the ratio of the non-SMP and the SMP parallel efficiencies is larger than $1 - M_2/M$, then the non-SMP mode will always outperform the SMP mode. Clearly, the condition for this to happen will be more difficult to satisfy if setting M_2 to a smaller value relative to M. Therefore, setting $M_2 = 1$ will lead to the least possibility that the performance of non-SMP mode will be always better. Practically, the parallel efficiency of a program tends to decrease when the program scales up. In particular, for real-life scientific and engineering parallel applications, the parallel efficiency is rarely higher than 50% for large-scale runs. So it is likely the performance of SMP mode surpasses that of non-SMP mode even at the expense of losing some cores dedicated for communication. The fact of the actual application parallel efficiency also entails the feasibility (in terms of achieving better performance) of partitioning a physical multicore node into more logical SMP nodes, in which case multiple threads across different SMP nodes are dedicated to communication per physical node. This implies that we can have different sizes of a SMP node for a particular multicore platform. We evaluated application performances with different choices of SMP node size in section 8.3. In short, indicated by the aforementioned theoretical analysis, as long as the parallel efficiency of SMP mode over that of non-SMP mode is larger than the percentage of cores dedicated to computation per physical nodes, the performance in SMP mode could be better even though we have lost more cores for communication.

- **Distributing different types of communication work into different threads:** the second option differs from the other two options in that the communication work is further divided into two parts which are assigned separately to two different sets of threads. One set of threads handle message sending operations, while the other set perform message receiving operations which includes the polling for incoming network messages. Comparing with the first option, this separation may mitigate the contention in ensuring thread-safety of the lower-level communication library if two parts of the communication have no conflicting writing accesses. Speaking of the third option, in which all the communication work goes through M_2 communication threads, although the contention could be further reduced, M_2 threads may not make a full utilization of the network bandwidth (i.e., communication by M_2 threads is not enough to saturate the network). Every communication operation is associated with certain cost, so it is possible M_2 threads may not be fast enough to process the communication request in time, thus under-utilizing the network bandwidth. Note that M_2 is better to be set very small so as to reduce the contention and the loss of computation capability according to the analysis described before. Therefore, the problem of not fully utilizing the network bandwidth could turn to be more severe.

Assuming $M_2 = 1$, we use a benchmark written in CHARM++ to illustrate the under-utilization of the network bandwidth. In this benchmark, we conduct two sets of iterative communication. The communication pattern of this benchmark is showcased in figure 6.1. In every iteration of the first one (referred as “serial” in figure 6.1), one core of node A will send out n messages to n different nodes respectively, and those n messages will be acknowledged for that single core to finish one iteration. However, in every iteration of the second set(referred as “parallel” in figure 6.1), n messages will be sent simultaneously from every core of node A to n different nodes, and the messages will also be acknowledged back to each sender to finish one iteration. From the perspective of node A, all n messages are serialized at its network port. n is usually set to be equal to the number of cores on a physical node. Note: n is set to 4 in the case illustrated by figure 6.1.

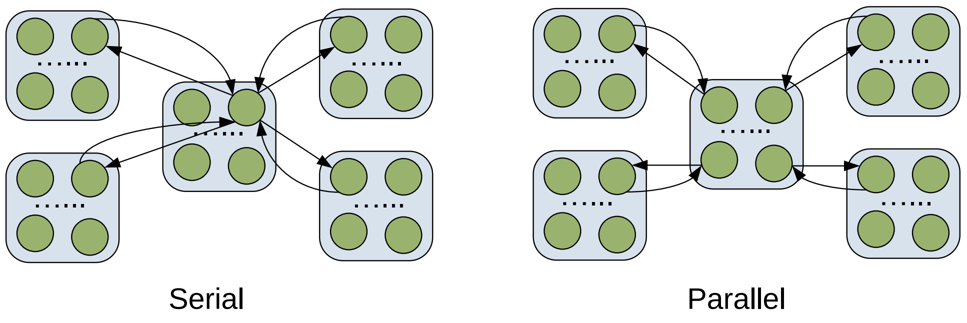

Figure 6.1: The Communication Pattern of the Benchmark Evaluating the Aggregated Sending and Distributed Sending

Figure 6.2 plots the ratio between the iteration time of the first set and the second set across different sizes of messages on Kraken, Hopper and Taub with $n = 12$. Clearly, the iteration time of the second set is better because more network bandwidth is utilized by more simultaneous sending from node A. However, on all three platforms, we can see the benefits of distributed sending decrease with the increase of message size because the serialization at the network card begins to dominate the performance, and the network bandwidth begins to be saturated.

The above experiment shows the potential benefit of distributing communication work to different threads so that the network bandwidth could be better utilized. However, such benefit could be offset by the cost of ensuring the thread-safety of those distributed communication

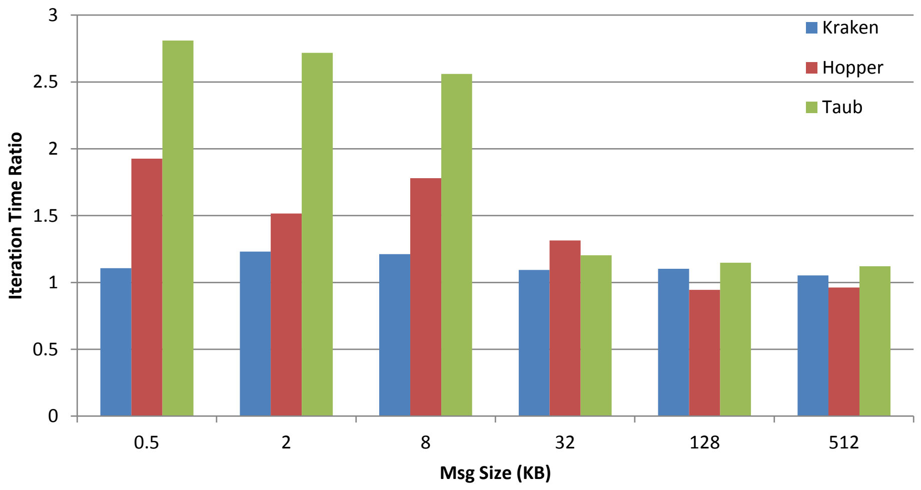

Figure 6.2: Performance Comparison between Aggregated Sending and Distributed Sending

operations (i.e., more contention caused by threads) in some cases. Still assuming $M_2 = 1$, our experiment using MPI as the communication substrate in NAMD illustrates this point. The MPI standard specifies four levels of increasing thread safety [45] as:

1) MPI_THREAD_SINGLE; 2) MPI_THREAD_FUNNELED;

3) MPI_THREAD_SERIALIZED; 4) MPI_THREAD_MULTIPLE.

The maximum level supports concurrent MPI function calls while the other three essentially only allow one thread to call MPI functions at a time. Therefore, when the maximum thread-safety is supported, we could choose any options for implementation while the third option is the only choice when the lower thread-safety is supported. In the experiment, we measured the performance of NAMD with the standard benchmark input Apoa1 molecule system on 192 cores of Kraken and Hopper (on both machines, one CHARM++ SMP node contains 12 cores). On these two Cray machines, the installed MPI library could be configured to support the maximum thread safety while the default configuration supports a lower level of thread safety. Table 6.2 shows the relative performance result for three runs, shown in the last three rows respectively. All performance data (measured as the time spent per timestep) have been normalized to the performance using the default MPI library that has a lower level of thread safety. Comparing the second row and the third one in table 6.2, distributing the communication work into two sets of threads (i.e., the second design option in assigning communication

Configuration	Kraken	Hopper
MPI not in thread-safe mode w/o communication distribution	1.00	1.00
MPI in thread-safe mode w/ communication distribution	1.18	1.17
MPI in thread-safe mode w/o communication distribution	1.02	1.01

Table 6.2: Performance Evaluation of Distributing the Communication Work Using MPI as the Communication Substrate

work to threads presented in section 6.1.1) degrades the performance by about 15% on both parallel machines. In order to understand whether the performance degradation is caused by the changes in the MPI library as it was configured to support the maximum thread safety, we compared the performance with and without communication distribution under the same MPI library. Demonstrated by the third and fourth row of table 6.2, we see the performance with the communication work separated is still worse. According to the man page of MPI on these two Cray machines, the MPI implementation that supports the maximum thread safety could cause performance issues because certain global lock has been used to protect MPI function calls. Consequently, the contention on the global lock is much higher in the case of distributing the communication work to two sets of threads, which we believe caused the worse performance of NAMD in this experiment.

Therefore, there is a performance trade-off in whether to distribute the communication work to different threads. With the distribution, the network bandwidth could be better utilized but the cost of contention in ensuring the thread-safety of communication operations may be too high to lead to an overall better performance. In this experiment, we only studied using MPI as the communication substrate to conclude with such a tradeoff. However, using other lower-level communication libraries, like DCMF or uGNI, to build the CHARM++'s communication subsystem will expose more fine-grained communication control, thus we will have better opportunities to reduce the granularity of critical sections for communication operations. The resulted reduction of communication contention may render a different conclusion that having the communication work distributed is always a better design option.

Analysis Summary: in short, considering the contention overhead in ensuring the thread-safety of communication operations, the cache pollution resulted from the alternation of the communication and computation work and the responsiveness to network messages, we regard the first option not to be a good choice compared with the other two options which separate the communication and computation work to different threads. Because of the performance trade-off in whether to distribute different types of the communication work to different threads, we remain neutral in the preference on the second and third option. Specifically, when it comes to sending network messages, they can either be sent directly from a set of threads that also handle the computation work (i.e., the M_1 threads in the second option) or from the subset of threads that only perform the communication work (i.e., the M_2 threads in the third option in which case all communication operations are funneled through those M_2 threads).

6.1.3 Summary of SMP Mode Implementation in CHARM++

Based on the analysis in the previous section 6.1.2, the implementation of the multithreaded mode (i.e., the SMP mode) in CHARM++ follows the second and the third options in which the computation work and the communication work is separated into two sets of threads. To reduce the contention overhead in ensuring the thread-safety of communication operations and the loss of computing power per SMP node, M_2 is set to 1 in CHARM++'s implementation. To compensate for the possibility of under-utilization of network bandwidth on a physical node, multiple SMP nodes could be launched per physical nodes if necessary.

In a nutshell, for the implementation of the CHARM++ SMP mode (i.e., the multithreaded implementation of CHARM++ runtime system), if a logical SMP node is allocated with $N + 1$ cores, then $N + 1$ threads are spawned per SMP node, where N threads are for the computation work (called *worker threads*), and the remaining one thread is dedicated to the communication work (called *communication thread*). Table 6.3 lists the major differences between the non-SMP mode and the SMP mode.

Because the underlying communication library could only deliver a message to a SMP node, not to a specific thread on that node, in the CHARM++ runtime, a special field is required in the header of every message to indicate on which thread the incoming message should trigger the associated computation. The value of this special field is set during the send operation to the *rank* of the destination thread. The *rank* of a thread is used to

non-SMP mode	SMP mode
• An **OS process** represents a CHARM++ PE, and is mapped to a logical CPU (a core or a hardware thread). • **Disjoint** virtual memory address space for all PEs on a physical node. • **Alternative** computation and network communication per PE.	• A **thread** represents a CHARM++ PE, and is mapped to a logical CPU (a core or a hardware thread). • **Single** virtual memory address space for all PEs on a SMP node. • **Separated** computation and network communication work among threads. • If the SMP node has N cores, N-1 of them host worker threads, the left one is dedicated to the communication thread. • Worker threads only perform the computation work, but may also send network messages. • The communication thread only performs the communication work.

Table 6.3: Summary of Implementation Differences between the non-SMP Mode and the SMP Mode in CHARM++

index all the threads in a SMP node. Since the SMP mode is designed to be transparent to application developers, the CHARM++ runtime system assigns parallel objects, such as chares that encapsulate the data and the control flow, only to the N worker threads on each node but not to the communication thread. Additionally, the communication thread is generally not exposed as a ranked PE even in CHARM++ library codes. The *rank* of *worker* threads is then valued from 0 to $N-1$, the *communication* thread is ranked as N. If we claim M SMP nodes in the application, then $M \cdot (N+1)$ threads in total are invoked, meaning $M \cdot (N+1)$ logical CPUs are needed for this application due to the one-to-one mapping between threads and logical CPUs. However, the number of PEs in the application will be reported as $M \cdot N$ in CHARM++.

The work flow of a worker thread and a communication thread in SMP mode is illustrated in figure 6.3. Comparing the flow in non-SMP mode shown in figure 2.1, we can see the flow of a PE (i.e., an OS process in non-SMP mode) is basically divided into two parts: one for the worker thread, and the other for the communication thread.

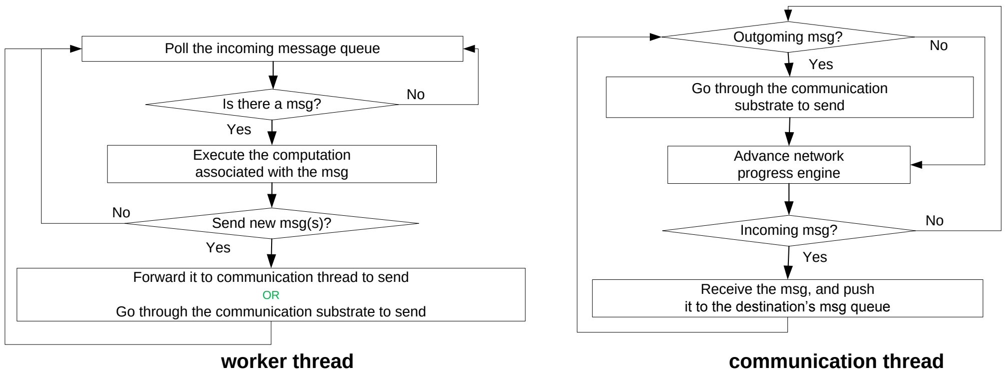
Poll the incoming message queue
Is there a msg?
No
Yes
Execute the computation associated with the msg
Send new msg(s)?
No
Yes
Forward it to communication thread to send
OR
Go through the communication substrate to send
worker thread
Outgoing msg?
No
Yes
Go through the communication substrate to send
Advance network progress engine
Incoming msg?
No
Yes
Receive the msg, and push it to the destination's msg queue
communication thread

Figure 6.3: The Flow of a Worker/Communication Thread in SMP Mode

As a language runtime may be built on different communication substrates, the second and third design options described in section 6.1.1 are both implemented in CHARM++ depending on the specific communication substrate used. For example, for DCMF on BlueGene/P, it is safe to send messages from each thread and one physical node only has 4 cores, as a result, the default implementation follows the second design option in order to utilize the network bandwidth better. In case of MPI, as described before, we implemented both options, but with a default to the third one where there is only one communication thread per SMP node responsible for all the network communication because this requires the least thread safety support from MPI. It guarantees the runtime to be most portable across different platforms as every MPI implementation at least functions correctly if all MPI function calls are made from one thread. The implementation of the second design option could be enabled by a user-specified runtime parameter if the actual MPI implementation installed on the parallel machine supports the maximum level of thread safety.

6.2 Optimizing the Usage of MPI as the Communication Substrate for the Message-driven Runtime

The communication component in the parallel runtime is indispensable for its responsibility of message passing between networked nodes. Generally, multiple high-performance lower-level communication libraries are available on massively parallel machines, among which MPI is the most deployed communication library. This section focuses on the usage of MPI in the context of the message-driven parallel language runtime system. First, we describe the benefits of using MPI as the communication substrate. Then we discuss the issues in using MPI for the message-driven runtime and present corresponding techniques to address them. This section will shed light on implementations of other new parallel languages when using MPI as the communication substrate, particularly for those that also adopt a message-driven execution model.

6.2.1 Pros and Cons of Using MPI as the Communication Substrate

MPI, the *de facto* standard for message passing interface, is the most widely used high-performance communication library for large-scale parallel machines. According to our experience, we have found the following benefits of using MPI as the communication substrate for a parallel language runtime system.

• **The best portability across different platforms:** MPI has been installed on almost every massively parallel machine. Therefore, the communication module of a parallel language runtime that is built on MPI is the most portable one, regardless of the underlying high-performance network hardware used on the machine. Our experience with Infiniband "ibverbs" demonstrated the issue of portability. We initially developed the communication module using "ibverbs" on the Mellanox Infiniband platform, and thought it would also work on the other two Infiniband platforms. But it turned out to be not the case because the three platforms have different implementations of the "ibverbs" library which in turn affect the usage of "ibverbs" API. Therefore, using MPI provides better portability, enabling the rapid deployment of the parallel language, and applications that are implemented in that language, on almost every parallel machine. It should be noticed that there are new initiatives such as the Common Communication Interface (CCI) [46] to substitute for MPI as a portable high-performance communication stack in the HPC community. But it may take years for such work to mature and be widely accepted.

• **Great for rapid prototype of communication subsystem of the language runtime system:** it takes less effort to use MPI as the communication substrate for the runtime than using other communication libraries due to two reasons. First, MPI has a well defined programming interface, specification and documentation that are generally accessible online. Secondly, it enjoys much wider community support than any other high-performance communication library. Hence, it is possible for a rapid prototype of any new parallel language using MPI to be used almost everywhere. Developers of the new parallel language could then focus more on the part in language design which is known to affect the programming productivity most.

- **A potential candidate as a common runtime stack for interoperability among different languages:** if acting as the communication substrate for different parallel languages, MPI is able to provide a common software stack for interoperability among those parallel languages. Interoperability has become increasingly important as we are heading to the exascale computing era in the next decade. First of all, with parallel machines growing significantly in size, scientific and engineering applications that can fully utilize those machines will become more complex consisting of different modules, each of which could be programmed in different parallel languages by different developers across different geographical locations. Thus, the interoperability among those modules is a must. Furthermore, considering that the majority of existing production-level applications running on those massively parallel machines are written with MPI, the adoption of new parallel programming languages is heavily dependent on how to convert those applications with the new language. Admittedly, it is a daunting task to rewrite from scratch those legacy applications in a new language. However, the interoperability of the new language with MPI provides an evolutionary path for the adoption of the new language. Simply speaking, for an existing MPI program, its low-performance components could be gradually replaced with high-performance components re-written in new parallel languages. In short, there is an increasing drive in the direction of interoperability in the HPC community [47, 48] in which MPI plays an important role.

However, because of the overhead from the MPI software stack, in terms of performance on absolute message latency, MPI may result in lower performance than some other lower level communication libraries [49], such as machine vendor provided ones like DCMF from IBM for BlueGene/P [41], uGNI from Cray for Cray XE series [42, 50], etc. In fact, MPI libraries on those platforms are always built on top of those lower level communication libraries. Furthermore, a language may have a different execution model from the MPI. Such differences may introduce undesirable usage of MPI, thus leading to a further performance loss. In the case of using MPI as the communication substrate for the message-driven CHARM++ runtime, CHARM++ messages are always "unexpected" while MPI favors "expected" messages as mentioned in section 2.2. This "unexpected" characteristic of using MPI gives rise to major communication performance issues.

In summary, the following problems have been found to prevent the communication component of the message-driven runtime from achieving high performance when using MPI:

- **Potential insufficient amount of unexpected receiving buffers:** the application could potentially fail in the middle of the execution if the internal unexpected receiving buffers of MPI become insufficient. Without proper error messages, it would be very bewildering for application developers to figure out the root cause of this running failure.

- **The extra copy overhead for "unexpected" small-size messages:** MPI uses an eager protocol [51] for small-size messages, in which case an extra copy is required to transfer the message from the internal unexpected receiving buffer to the user buffer if the message is unexpected. However, for high-performance network connections that are capable of RDMA [52], such extra copy cannot be avoided regardless the type of message (expected or unexpected). In those networks, the memory allocated for messages has to be registered first before messages can be sent or received because network RDMA is utilized to transfer the message. However, memory registeration is more expensive than a memory copy for small-size messages. Consequently, small-size messages are always put into internal pre-registered buffers first before being copied to the user buffers. In other words, "unexpected" small-size messages on those networks exert no more overheads than those "expected" ones as both require the message copy from the library internal buffer to the user buffer.

However, for message-driven runtimes, as is the case with CHARM++, the memory buffer of the received message is not allocated by the user application, but provided by the runtime. So using other lower-level communication libraries such as uGNI for Cray Gemini networks [53] which MPI is built upon, the message copy mentioned above can be avoided. Simply speaking, the internal memory buffer for "unexpected" messages managed by MPI could be directly used by the user-level applications in CHARM++, thus saving a memory copy.

- **The extra synchronization overhead for "unexpected" large-size messages:** MPI uses a rendezvous protocol [54] for large-size messages, in which case the actual message payload will not be transferred from the sender side until the sender becomes aware that a corresponding buffer on the receiver side is available. To hide the extra one-way latency caused by this sender-receiver synchronization, it is better to allocate the memory buffer and notify the sender as early as possible. If such a large-size message is "expected", then it is easy to decide when to perform the memory buffer allocation and notify the sender (a.k.a., posting a receive buffer in MPI). In contrast, when the message is "unexpected", there is no way to tell the appropriate time for posting the receiving buffer. Generally, in the message-driven runtime, such post is performed right at the time when the receiver is notified with the next incoming message. Thus, the synchronization overhead has to be counted in full to the overall message latency because it could not be hidden effectively as the "expected" large-size messages could by posting the receiving buffer earlier.

In short, there are both advantages and disadvantages in using MPI as the underlying communication library for the parallel language runtime. However, thanks to the advantages of using MPI, it is still worth optimizing its performance as the communication substrate for a language runtime system. In this thesis, we have taken the implementation of the communication subsystem in the message-driven CHARM++ runtime on top of MPI as a case study to investigate how MPI could be better used. Note that we have only used the two-sided operations in MPI to construct the communication in CHARM++, not the one-sided operations due to their restricted usage described in section 2.2.

6.2.2 Techniques to Overcome Disadvantages in Using MPI as the Communication Substrate

To overcome the disadvantages of using MPI as the communication substrate for a message-driven runtime, the following approaches could be taken to improve the performance:

1. **Overlapping communication and computation:** this is a common approach to optimizing the message-passing parallel programs to hide communication latencies. Adopting this optimization idea during implementing message-passing programs will help alleviate the problem of the higher message latency caused by using MPI as the communication substrate than directly using lower-level communication libraries. CHARM++ programs, in general, automatically enjoys this optimization because of the over-decomposition and asynchronous communication. First, when one object is idle waiting for message arrival on a PE, another object on the same PE could exploit this idle time by doing useful work. Furthermore, the one-sided asynchronous communication allows the execution of computation when the communication is happening at the same time.

2. **Hybrid receiving schemes**: when receiving the large-size messages, due to the required synchronization between the sender and receiver, it is better to use the non-blocking receive call MPI_Irecv instead of the blocking call MPI_Recv. The time spent on waiting for the completion of receiving the large-size message, as a result, could be utilized to perform other useful work. It is particularly important in the case of the multithreaded runtime that has every network communication funneled through a dedicated communication thread. The blocking MPI_Recv call for a large-size message may cause a long stall on the communication thread preventing the communication thread from serving other work promptly, such as sending out messages. In short, we optimize the receiving part with a hybrid scheme that uses the non-blocking MPI_Irecv call for large-size messages while still using the blocking one for small-size messages.

We evaluate this hybrid receiving scheme via a simultaneous multi-pingpong benchmark on four physical nodes (128 cores in total) of JYC machine. In this benchmark, PEs are divided into multiple groups and each PE p is assigned a rank as $p\%groupSize$. We benchmark the execution time taken to complete a step in which every PE pingpongs with corresponding PEs of the same rank in every other groups. In the experiment, we set the $groupSize$ to be 8 in the non-SMP mode, and same as the SMP node size 7 in the SMP mode considering the dedicated communication thread of each SMP node occupies a core exclusively. Figure 6.4 shows the percentage of speedup of this hybrid receiving scheme over the default one both in non-SMP mode and SMP mode.

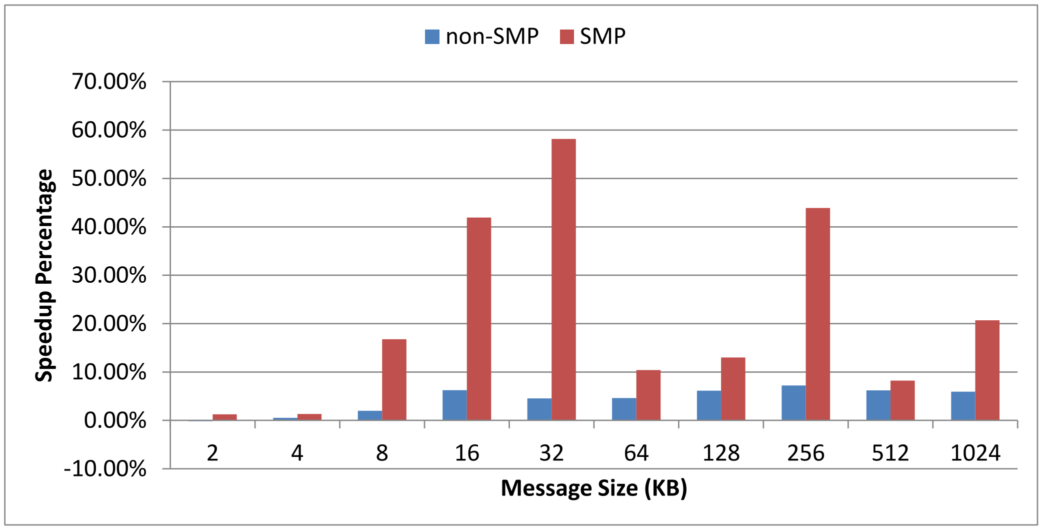

Figure 6.4: Performance of the Hybrid Receiving Scheme on a Multi-Pingpong Benchmark

First, since we do the non-blocking receiving only on large-size messages as 8KB in this case, almost identical benchmark performances are achieved for message sizes of 2KB and 4KB in both modes. The slight differences are due to the system noise [55, 56, 57]. Secondly, based on figure 6.4, we can see that the hybrid receiving scheme is more beneficial in the SMP mode than that in the non-SMP mode for message sizes larger than or equal to 8KB. Given that all network communication is funneled through the communication thread in the SMP mode, and the benchmark only has network communication in this case, the cost of a blocking receiving call amplified by the synchronization overhead of large-size messages is much more than that in the non-SMP mode. Once the communication thread is blocked, it cannot respond to other incoming messages promptly.

3. **Pre-posting of receiving buffers:** to address the issues caused by "unexpected" messages when using MPI as the communication substrate in the message-driven CHARM++

runtime, or any other similar runtime systems such as HPX [58], SWARM [59], we pre-post receiving buffers to make messages “expected” for MPI.

First, we proposed a static approach as posting a certain number of receiving buffers at the beginning of the program on each node. If an “unexpected” message is matched with a pre-posted buffer, a same pre-posted buffer is posted again. The number of buffers and the range of message sizes to be posted are made statically configurable via command-line options so that those values could be tailored to each different applications.

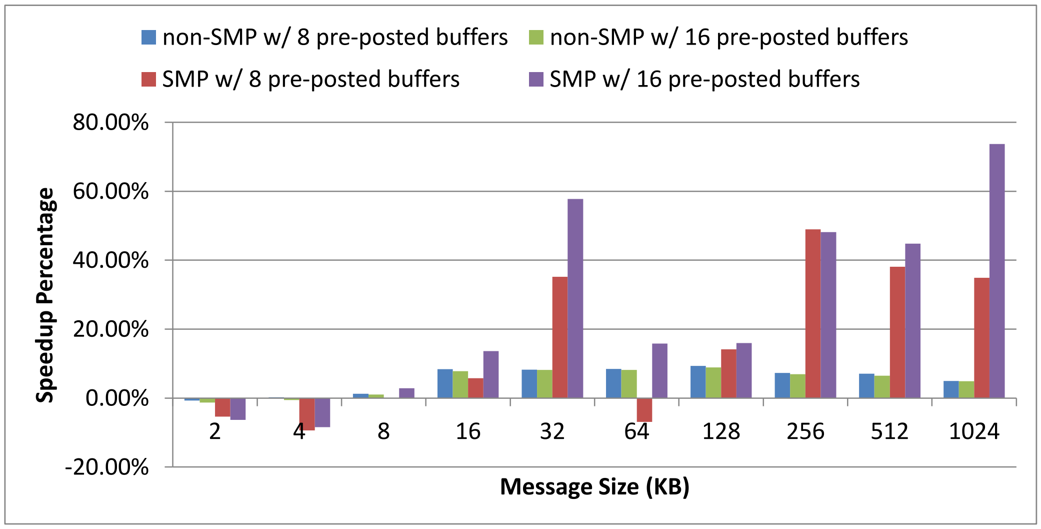

Figure 6.5: Performance of the Hybrid Receiving Scheme on a Multi-Pingpong Benchmark

We evaluated this static approach using the same simultaneous multi-pingpong benchmark mentioned above. We also performed the test on four physical nodes of JYC both in the SMP mode and in the non-SMP mode with the same parameter settings. Since we know the message size in prior, we adjusted the size of a receiving buffer to the amount that the message just fits in. The performance speedup over the default scheme of two runs with different number of receiving buffers is shown in figure 6.5. Based on this figure, we can find that the pre-posting scheme does not lead to performance improvement when the message size is small (i.e., less than 8KB) and the communication uses the eager protocol. In this scenario, we think the overhead of managing those pre-posted buffers offsets the benefit from saving the additional message copy (i.e., from the internal “unexpected” message buffer to the user memory). Such overhead is amplified by the funneled communication in the SMP mode. For large-size messages, first, we see this scheme improves the performance generally thanks to the save of synchronization incurred by the rendezvous protocol on the message. Secondly, it also achieves more performance improvement in SMP mode due to the same funneled communication. Finally, adding more pre-posted receiving buffers leads to more benefits in this benchmark in general.

Although the multi-pingpong benchmark shows the benefits of this static pre-posting scheme, there are several concerns over this static scheme in a parallel language runtime as detailed in the following:

(a) How many buffers should be pre-posted? Since those pre-posted buffers have to be checked from time to time in order to test whether a matched "unexpected" message is there, a large polling overhead will be incurred if we pre-post many buffers.

(b) What is the size of the buffers to be posted? If pre-posted buffers are never used by the runtime, then this scheme is not useful at all and still cause additional polling overhead. If pre-posted buffers are much larger than the "unexpected" message size, then extra memory space is wasted. Although the above two parameters (the number of buffers and buffer size) could be configured as execution parameters by users, it requires a deep understanding of the application to make them appropriate.

(c) The statically pre-posted receiving buffers do not work well in dynamic applications due to three reasons. First, "unexpected" messages may have different ranges of message sizes on different nodes. As a result, the pre-posted receiving buffers may be appropriate for some nodes, but not for others. Secondly, the size of"unexpected" messages may be distributed across several ranges across the overall execution on a node. Therefore, the initial range of pre-posted buffer sizes may not be appropriate later even on the same node. Finally, a dynamic application may have different communication phases. Consequently, the pre-posted receiving buffers may work well in some phases, but not in others. This challenge is more pronounced for CHARM++–a general parallel language runtime which may experience "unexpected" messages of various sizes on every node.

We addressed the above concerns via a more sophisticated dynamic and adaptive scheme to pre-post receiving buffers. This scheme is taking advantage of the fact that HPC applications are usually iterative, hence, exhibiting persistent communication patterns as the near past is reflected by the near future. In this scheme, every time interval, we do a histogram of message that are received during the past interval on a node over a very wide range of message sizes. To determine the default values on a platform, we perform pingpong tests to obtain the minimal and the maximal message sizes that pre-posting a receiving buffer will provide benefits. Currently, the bucket size is set to 1KB as a default. Note that all those parameters regarding message histogram could be overwritten by users at the start of program.

Based on the histogram of message sizes over a period of time, we pre-post receiving buffers to capture those most frequent message sizes that "unexpected" messages have. We have two issues here: how do we handle those pre-posted receiving buffers that are no longer needed according to the message size histogram? And how often do we rebuild the pre-posted receiving buffers? Regarding the first issue, we first need to cancel the request put on the receiving buffer, and then free the buffer. Unfortunately, MPI has not provided a function that can tell whether a request is in the process of being fulfilled, implying the runtime could not automatically and safely cancel a request. Unless MPI provides such a function call in the future, a full dynamic and adaptive pre-posting scheme would be impossible. Therefore, the current adaptive mechanism requires the user program to provide a safe point so that there are no ongoing messages that are received into those buffers. This user-specified safe point, however, also determines the frequency of rebuilding pre-posted receiving buffers.

6.3 Tuning Communication Performance

In this section, we analyze the communication performance issues we have identified in the study and present runtime techniques employed to resolve those issues. To begin with the investigation, it is necessary to have a performance analysis tool to help the understanding of the performance issues associated with the multithreaded runtime in the multi-node case. Since we take CHARM++ runtime as the experimental platform in this thesis, we extend the associated performance analysis tool–"Projections" [60, 61] so as to serve the multithreaded runtime better. Then, we focus on analyzing the communication thread as it is responsible for the network communication so that we can understand how it will affect application performances. In the end, we describe the optimization techniques that are used in the runtime system to improve the communication performance.

6.3.1 The Extension of Performance Tracing Framework and Visualization Tool

To understand performance issues in a message-passing parallel application, a performance analysis tool is generally required to help developers identify problems. With the multithreaded runtime, as there are dedicated communication threads in the preferred design, the communication path will be changed in applications. Therefore, it is necessary to extend the performance analysis tool to reflect the change. This section is devoted to this part of work of my thesis. We will first introduce the performance analysis tool–"Projections" [40, 62] associated with CHARM++ programs, and then describe the extension of this tool for the SMP mode of CHARM++ as well as a simple case study.

Introduction to Projections: The Projections consists of two components. The first one is a tracing module embedded in the CHARM++ runtime system, which is responsible for generating performance tracing files for every PE. As the runtime is aware of when a message is sent, when it is received and when it is scheduled for triggering the associated computation work, at those execution points, the runtime then automatically records performance-related data, such as timestamps for the beginning and the end of a piece of work, into a trace file on every PE if tracing is enabled by the user. Therefore, CHARM++ programs could enjoy this automatic way of obtaining tracing files without changing source codes. To reduce the amount of tracing data generated for large-scale runs, this tracing module also provides APIs to manually turn on and off the tracing capability as well as options of tracing only a subset of PEs. The reduction on the amount of collected data entails better scalability of the performance analysis tool and less perturbation on the application performance.

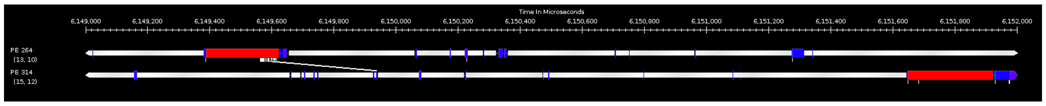

Figure 6.6: A Snapshot of Timeline View of Projections

The second component of Projections is a Java-based GUI visualization and analysis tool. Based on the input of the tracing files, this component provides a set of functionalities to help understand and analyze the performance, such as the timeline view of events [61], the communication volume of each PE and system noise analysis [63] etc.. For example, figure 6.6 shows a snapshot of timeline view in Projections. In the figure, bars of different colors represents different computation work on a PE. The white portion stands for idle time of the PE. The line starting from PE 264 to PE 314 indicates the path and the latency of a message. By tracking the path and latency of messages, we can understand the flow of the program and figure out the performance critical path.

The Extension to Projections for the multithreaded runtime: it is known that message latency is an important performance factor. A message with an unexpected longer latency may cause a cascading effect that reduces the performance of the whole application. In the multithreaded runtime that is preferred in design as summarized in section 6.1.2, the communication between two SMP nodes involves the communication thread. For example, in the default implementation of CHARM++ SMP mode on top of MPI, all communication is funneled through the communication thread where a point-to-point message has two more intermediate steps besides the step of network transfer for the message. One such step is on the sender side as the sender pushes the message to the communication thread, and another one is on the receiver side as the communication thread pushes the received message to the destination processor. Without changes specifically for the multithreaded runtime, Projections would not display each of these two additional steps but just one step combining all three steps. The message shown in figure 6.6 demonstrates such an example. The inter-node message from PE 264 to PE 326 is displayed as a single-step communication, and it takes more than 300us to be delivered in SMP mode! Such a high latency is indeed intriguing as such a message is expected to take at most tens of microseconds. If the two intermediate communication steps were displayed, we would have a clearer idea of the root cause of this high latency.

To address this issue, the two components of Projections are extended respectively. In the tracing module, additional information regarding the SMP mode, such as the number of SMP nodes etc., is recorded. Various events that execute on the communication threads have to be traced as well. For example, in the MPI-build SMP mode of CHARM++, when the message is sent by MPI_Isend on the communication thread, a tracing event is created containing the information that will link the message with the originating worker thread (the message is first sent out from the worker thread, and then is forwarded to the communication thread). Correspondingly, when the message is received on the communication thread, another tracing event is created containing the information that will link the message with the destination worker thread. Furthermore, to understand other costs on the communication thread, particularly the MPI function calls, we also add trace events that record the execution time for those functions. However, as invocations of those functions are quite frequent, we only record those "abnormal" ones defined as their execution time exceeding a certain pre-defined threshold. In this way, we can limit the number of tracing events for a better scalability but still keep track of important performance factors.

With the additional information of SMP mode and the trace files of communication threads, we extend the GUI-based visualization and analysis component of Projections with more features to help better understand the performance issues in the multithreaded runtime. Just to list a couple of them here: first, the label of a PE will be displayed with an additional SMP node identifier so that a message between two PEs can be easily decided whether it is a inter-node one. Secondly, in the timeline view, when the timeline of a PE is requested to be displayed, its associated communication thread will be automatically loaded. This is useful to check if there any activities on the communication thread during the same period when the worker thread is idle. Additionally, when displaying an inter-node message, all the three steps will be displayed at the same time instead of displaying each step on demand, which shows a clearer communication path and the latency of each step of this message.

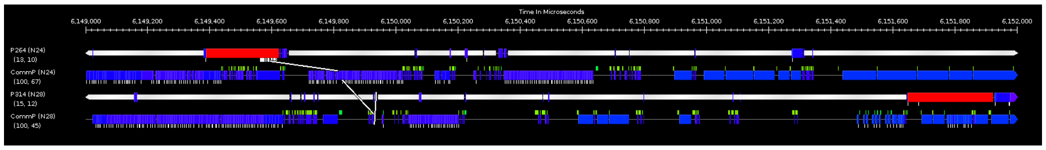

Figure 6.7: An Enhanced Timeline View of Figure 6.6

Figure 6.7 shows the same inter-node message as displayed in figure 6.6 but in the new timeline view after the aforementioned extension work in Projections. From the figure, we know PE 264 is on SMP node 24 (as shown "N24" in the bracket) and PE 314 belongs to SMP node 28. Therefore, the message is clearly an inter-node message that involves communication threads. Furthermore, the two intermediate steps are clearly illustrated: one is from PE 264 to its communication thread (as shown "CommP (N24)") and the other is from the communication thread of SMP node 28 to PE 314. Observing these three steps, we can easily figure out that there is a delay about 200us out of the 300us overall message latency on the sending communication thread. During this time 200us period, as displayed by many blue bars on the communication threads, we know that the communication thread is overwhelmed with other work including sending many outgoing messages, before it can send out this particular network message. Therefore, this will drive us to investigate more on the activities of the communication thread which is the topic of section 6.3.2.

6.3.2 Performance Analysis of the Communication Thread

As stated in section 6.1.2, the preferred design of a multithreaded runtime is to have dedicated communication threads to be responsible for network messages. Therefore, the communication performance will be closely affected by the issues on the communication thread. In this section, in the context of the SMP mode of MPI-based CHARM++ runtime, we will analyze the strategy of alternating different communication operations performed by the communication thread and the situation when the communication thread has been overloaded with too much work. The analysis lays a foundation of the optimization techniques that are presented in the following sections.

In the SMP mode of MPI-based CHARM++ runtime, all network communication is funneled through the dedicated communication thread because the maximum thread safety of MPI (i.e., MPI_THREAD_MULTIPLE) is not generally supported in default. With this setting, the communication thread is alternating among three types of work as sending outgoing messages, freeing delivered message buffers and probing network then receiving incoming messages. By default, we applied the "best-effort" strategy that the communication thread will not switch to another type of work until there is no more work of the current type. For example, if there are N outgoing messages in the queue, the communication thread will send all those N messages, and then switch to the other two types of work.

We have identified the following performance issues on the communication thread, with all of them related to message latencies:

- **Dilemma in "best-effort" strategy for alternating sending and receiving messages:** from the perspective of a sender, outgoing messages should be sent out as soon as possible in order to trigger the computation on receivers. However, standing on the side of a receiver, the incoming messages should be polled and received as early as possible in order to start the computation or new communication. Since the communication thread plays a role as both sender and receiver, the thread is facing a dilemma as which type of work is more important: the message sending or the message receiving? Using the average waiting time of a message to be served (either to be sent out to other nodes, or to be forwarded to worker threads) as a metric will analytically illustrates this dilemma. The following conditions are assumed:

– there are M outgoing messages and N incoming messages [1] on the communication thread at time T.

– the communication thread will serve m outgoing messages as a unit at a time and n incoming messages as a unit at a time where $1 \leq m \leq M, 1 \leq n \leq N$.

– let the cost of serving an outgoing message be O_s and the cost of serving an incoming message be O_r. In addition, O_w stands for the service switch cost of communication thread; We assume those costs are constant.

[1]Strictly, in MPI-based CHARM++, the variable N is not known because MPI_Iprobe is used to poll the network.

– we only consider the case when $M = m * C, N = n * C$, which means the communication thread need to serve C times for M outgoing messages and N incoming messages, respectively. This implies that C as an integer stands for the number of service switches. If $M = m * C1, N = n * C2$, without generality, suppose $C1 > C2$, then C is equal to $C2$. It is clear for the remaining $m * (C1 - C2)$ messages, they should be served at one time (i.e., using the "best-effort" strategy) to save the service switch cost, thus achieving the least amount of total waiting time. Therefore, we only consider the case when outgoing and incoming messages are served with the same amount of times as C.

If at this particular time T, the communication thread happens to serve outgoing messages first, then based on the above conditions, the beginning time T_k^S of serving the kth (where $1 \leq k \leq C$) unit of m outgoing messages is calculated as summing the beginning time T_{k-1}^R of serving the $(k-1)$th unit of n incoming messages, the service switch cost O_w and the total time that the communication thread spends on receiving n incoming messages. The equation 6.1 simply represents such calculation.

$$T_k^S = T_{k-1}^R + (O_w + nO_r), k \in [2, C] \text{ and } T_1^S = T \tag{6.1}$$

Similarly, we could derive the beginning time of every n incoming messages as equation 6.2.

$$T_k^R = T_k^S + (O_w + mO_s), k \in [1, C] \tag{6.2}$$

Replacing T_{k-1}^R in equation 6.1 by equation 6.2, we can derive the expression 6.3 for calculating the beginning time of serving one unit of outgoing messages.

$$\begin{aligned} T_k^S &= T_{k-1}^S + (2O_w + nO_r + mO_s), k \in [2, C] \\ \Longrightarrow T_k^S &= T_1^S + (2O_w + nO_r + mO_s) \cdot (k-1), k \in [2, C] \\ &= T + (2O_w + nO_r + mO_s) \cdot (k-1), k \in [1, C] \qquad (6.3) \end{aligned}$$

Following the same procedure, we can derive the beginning time of receiving one unit of incoming messages as expressed by equation 6.4.

$$\begin{aligned}
T_1^R &= T_1^S + (O_w + mO_s) = T + (O_w + mO_s) \\
T_k^R &= T_{k-1}^R + (2O_w + nO_r + mO_s), k \in [2, C] \\
\Longrightarrow T_k^R &= T_1^R + (2O_w + nO_r + mO_s) \cdot (k-1), k \in [2, C] \\
&= T + (O_w + mO_s) \\
&\quad +(2O_w + nO_r + mO_s) \cdot (k-1), k \in [1, C] \qquad (6.4)
\end{aligned}$$

Based on equations 6.3 and 6.4, we could derive the total relative waiting time of every message in each serving unit.

$$\begin{aligned}
WT_k^S &= \sum_{i=1}^{m}((T_k^S - T) + (i-1)O_s) \\
&= m \cdot (T_k^S - T) + \frac{m(m-1)}{2} \cdot O_s, k \in [1, C] \qquad (6.5)
\end{aligned}$$

$$\begin{aligned}
WT_k^R &= \sum_{i=1}^{n}((T_k^R - T) + (i-1)O_r) \\
&= n \cdot (T_k^R - T) + \frac{n(n-1)}{2} \cdot O_r, k \in [1, C] \qquad (6.6)
\end{aligned}$$

Therefore, the total waiting time of all outgoing messages is derived as equation 6.7 and that of all incoming messages as equation 6.8

$$\begin{aligned}
T_{wait}^S = \sum_{j=1}^{C} WT_j^S &= m(2O_w + nO_r + mO_s) \cdot \frac{C(C-1)}{2} \\
&\quad + \frac{Cm(m-1)O_S}{2} \qquad (6.7)
\end{aligned}$$

$$\begin{aligned}
T_{wait}^R = \sum_{j=1}^{C} WT_j^R &= n(2O_w + nO_r + mO_s) \cdot \frac{C(C-1)}{2} \\
&\quad + \frac{Cn(n-1)O_R}{2} + Cn(O_w + mO_s) \qquad (6.8)
\end{aligned}$$

Thus, replacing m with M/C, and n with N/C, the total waiting time of all messages to be served by the communication thread is showed by equation 6.9.

$$\begin{aligned}
T_{wait} &= (M+N)\cdot\frac{C-1}{2}\cdot(2O_w+nO_r+mO_s)+N(O_w+mO_s) \\
&\quad +\frac{M(m-1)O_s}{2}+\frac{N(n-1)O_r}{2} \\
&= \frac{1}{2}(M^2+MN+mN-M)O_s+\frac{1}{2}(N^2+MN-nM-N)O_r \\
&\quad +(\frac{M^2}{m}+\frac{N^2}{n}-M)O_w \\
&= (M+N)O_w\cdot C+\frac{1}{2}MN(O_s-O_r)\cdot\frac{1}{C} \\
&\quad +\Big(\frac{1}{2}(M+N-1)(MO_s+NO_r)-MO_w\Big) \qquad (6.9)
\end{aligned}$$

It is clear that the expression 6.10 determines the minimal average waiting time of the total $M+N$ messages as other terms are constant.

$$(M+N)O_w\cdot C+\frac{1}{2}MN(O_s-O_r)\cdot\frac{1}{C} \qquad (6.10)$$

Now, if at the particular time T, the communication thread happens to serve incoming messages first, then the expression that determines the minimal average waiting time of all messages is symmetric to expression 6.10 as shown by expression 6.11

$$(M+N)O_w\cdot C+\frac{1}{2}MN(O_r-O_s)\cdot\frac{1}{C} \qquad (6.11)$$

Observing expression 6.10 and 6.11, it is clear that two expressions have contradictory conditions to achieve the minimal value, reflecting the aforementioned dilemma. In MPI-based CHARM++ runtime, O_r is usually larger than O_s, then expression 6.10 will become minimal when $C=1$, which is exactly the "best-effort" strategy in the case of message sending served first. However, if message receiving is served first, expression 6.11 will become minimal when $C=\left\lceil\sqrt{MN(O_r-O_s)/(2(M+N)O_w)}\right\rceil$ indicating the "best-effort" strategy may not be best in terms of the average waiting time of a message. Additionally, the above analytical model is simplified in that cost variables O_w,O_s and O_r have been assumed to be constant, which is not the case in reality. Because of that, it is possible that "best-effort" strategy is not optimal even if outgoing messages are served first from a particular moment T.

Analyzing the actual performance traces of applications, we found performance issues that also reflect the dilemma of the "best-effort" strategy. Some message latencies are much longer than usual because the communication thread is not able to promptly serve the message. This happens because either sending the outgoing message is delayed by continuously receiving multiple messages or the forwarding of the incoming message to destination worker thread is stalled by continuously sending messages. For example, such a message is illustrated in figure 6.7 where the message is not sent out because the communication thread is busy with other work. If this message lies in the performance critical path or its stretched latency could not be hidden by the computation on the destination worker thread, a degradation in the overall application performance will be incurred. This example empirically indicates that the "best-effort" strategy may not be best in all situations.

- **Overload in communication thread:** since all network messages have to be funneled through the communication thread, if there is too much inter-node communication, then the communication thread will become overloaded. This becomes more conspicuous when a burst of network messages occur in the application. For example, on large scale runs on JaguarPF in SMP mode, we noticed sometimes MPI_Iprobe (which tests if there is an incoming network message) abnormally took 12 milliseconds, thus stalling the execution of the communication thread. By studying the communication activities around the prolonged MPI_Iprobe function call, we identified that the burst of outgoing messages on every node around that time was the root cause. Consequently, the communication thread became overloaded. In particular, with the "best-effort" strategy in sending messages, during that period, every node would also experience a burst of incoming messages. As some internal expensive operation is associated with the receipt of a message on the network card of JaguarPF, a burst of incoming messages resulted in the stall reflected by the much prolonged MPI_Iprobe call.

6.3.3 Techniques to Improve Runtime Communication Performance

Corresponding to the performance issues regarding the communication thread as mentioned in section 6.3.2, in this section, we present the general techniques we have developed to attack those issues as follows:

- **"Restrained-effort" strategy for alternating work:** as suggested by the analysis on the simplified modeling of "best-effort" strategy, sometimes, it is better to restrain the effort on finishing one type of work so that the overall communication responsiveness is improved. In other words, we dynamically exert a cap on each type of communication tasks to control how many operations the communication thread could perform at a time. We name this cap-based strategy as "restrained-effort". Specifically, during the process of sending messages, if the communication thread has detected that it has sent out a number of messages exceeding the pre-defined threshold, or it has accumulated a certain number of messages that require to be released (i.e., freeing the message memory buffer), then the communication thread will stop the message sending work and change to the other work. Similarly, when polling the network progress engine to receive network messages, the communication thread will stop doing this work if it has performed more than the number of receiving messages allowed by the cap.

This simple "restrained-effort" strategy also helps to handle the burst of messages, as the cap on how many messages to be sent at a time prevents the flooding of network messages. It differs from credits-based flow control schemes such as those described in [64] in that it only involves the sender side without requiring feedback from receiving sides, and it is simpler but yet effective for large scale runs. With the 100M-atom simulation using NAMD on 4480 nodes (i.e., 53,760 cores), we observed a performance improvement of 12.3% after applying this "restrained-effort" strategy.

- **Node-aware communication to reduce network messages:** since the communication thread is responsible for all network communication, in order to reduce its load characterized by the time spent on sending outgoing messages and receiving incoming messages, the number of inter-node messages should be decreased as much as possible. With fewer inter-node messages, the communication thread will be more prompt to serve messages. Additionally, the network message is more expensive than the intra-node one as the latter one does not need to go through the network card. Therefore, reducing the number of network message also lowers the communication cost.

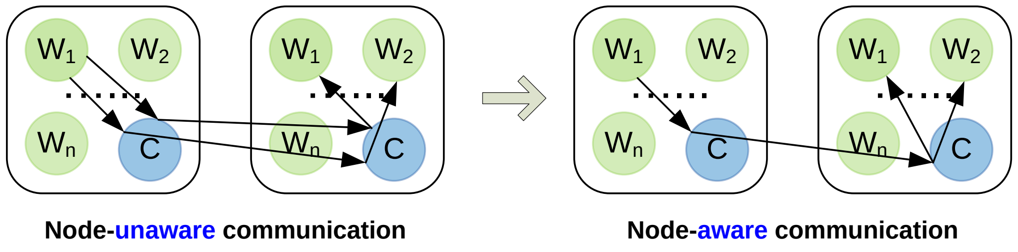

Figure 6.8: Differences between Node-unaware Communication and Node-aware Communication

There are different ways to reduce the number of network messages, such as using a different algorithm for the application that has less communication. In this work for runtime system, we apply a straightforward yet effective “node-aware” communication optimization technique, the idea of which is illustrated by figure 6.8. In this approach, we will remove duplicated messages which refer to those that are sent from the same node, but will be received by the same destination node. For example, in figure 6.8, PE $w1$ on one SMP node will send two same messages to two PEs $w1$ and $w2$ on another SMP node. Without node-aware communication optimization, two inter-node messages will be sent by the communication thread on the sender side. In contrast, applying the node-aware communication optimization, the communication thread on the sender side will just transfer one inter-node message while on the destination node, the communication thread will make an extra copy of the message and forward two copies of the message to the two destination PEs, respectively.

In the SMP mode of the CHARM++ runtime, we apply this optimization technique to communication idioms that exhibit the property of having “duplicated” messages, such as the multicast and broadcast etc. Take the broadcast communication as an example, the communication is optimized by broadcasting the message among the communication thread of every SMP node and then the communication thread will forward the message to every PE on each SMP node. Table 6.4 shows the number of inter-node and intra-node messages of different broadcast schemes. The first column shows the total number of SMP nodes involved in the broadcast. Since we dedicate one core to the communication thread and the numbers of physical cores on a multicore chip currently are usually 8, 12 and 16, we set the number of PEs per SMP node as 7, 11 and 15 as shown in the second column. The remaining columns show the different number of inter-node and intra-node messages in each broadcast scheme. In the "binomial" tree scheme, the parent of a PE p is calculated as $p\&(p-1)$, i.e., removing the first least significant bit of 1 in the binary representation of p. In the "simple" spanning tree scheme, if the branch factor (abbreviated as "bf") is C, then the parent of a PE p is calculated as $\lfloor p/C \rfloor = (p - p\%C)/C$. In the "node-aware" scheme, we do not specify the actual broadcast algorithm because the number of inter-node messages will not change regardless of the broadcast algorithm as explained later.

#nodes	#ppn	binomial		simple w/ bf=2		simple w/ bf=4		node-aware	
		inter	intra	inter	intra	inter	intra	inter	intra
128	7	303	592	889	6	889	6	127	768
	11	343	1064	1397	10	1397	10	127	1280
	15	375	1544	1905	14	1905	14	127	1792
256	7	607	1184	1785	6	1785	6	255	1536
	11	687	2128	2805	10	2805	10	255	2560
	15	751	3088	3825	14	3825	14	255	3584
512	7	1215	2368	3677	6	3577	6	511	3072
	11	1375	4256	5621	10	5621	10	511	5120
	15	1503	6176	7665	14	7665	14	511	7168

Table 6.4: Comparison among Different Broadcast Schemes with regard to Inter-node and Intra-node Messages

Observing table 6.4, first we can find the node-aware scheme entails the least number of inter-node messages comparing all the broadcast schemes. Such reduction in the inter-node messages will lead to a better performance. In addition, the number of inter-node messages remains unchanged regardless of the number of cores per node. This is because each SMP node will just receive one message from a different SMP node except the node that initiates the broadcast. Therefore, the algorithm of broadcast will not affect the number of network messages. Secondly, the number of intra-node messages remains same even if the branch factor of the simple spanning tree changes. Actually, it is equal to the number of PEs per node subtracted by one. Suppose a PE $p, p \in [n*ppn, (n+1)*ppn-1]$ where $n \in [0, N-1]$,

N stands for the number of total SMP nodes and ppn is the number of PEs per SMP node. Its children PE will be $p_c = C * p + i$ where $i \in [1, C]$, C is the branch factor. If this message from p to p_c is intra-node, then p_c must also satisfy $p_c \in [n * ppn, (n + 1) * ppn - 1]$. Therefore, we have the following deduction:

$$\begin{aligned} & p_c \le (n+1) * ppn - 1 && \Rightarrow C * p + i \le (n+1) * ppn - 1 \\ & \Rightarrow C * n * ppn + i \le (n+1) * ppn - 1 && \Rightarrow C * n < n + 1 \end{aligned}$$

Note C must be at least 2, in order to satisfy the above condition, n has to be 0. This means only PEs in the root SMP node of the simple spanning tree will receive an intra-node message, thus the total number of intra-node messages will be $ppn - 1$ irrespective of the branch factor. Correspondingly, the number of inter-node messages is $(N - 1) * ppn$ as the total number of messages is $N * ppn - 1$. Since the number of inter-node messages of the "node-aware" scheme is $N - 1$, the "node-aware" scheme will be always better by ppn times. The data in table 6.4 validates this analytical relationship between the "simple" spanning tree scheme and the "node-aware" scheme. Finally, the binomial tree scheme has much greater number of intra-node messages than the simple spanning tree scheme. Considering the total number of messages remains same as one less than the total number of PEs (as each PE will finally receive a message except the root PE), the binomial scheme will be better than the simple spanning tree one, but will be worse than that of node-aware one.

In the startup phase of NAMD startup, the performance is greatly improved after using the node-aware broadcast. In a run on 17,920 nodes (215,040 cores) of Jaguar, the startup phases of NAMD involve a series of broadcast operations ranging from 4KB to 65KB. In non-SMP mode, it takes 76.3 ms to finish. In contrast, the time is reduced to 20.2 ms on average, a speedup of 2.78 times, in SMP mode.

This node-aware communication optimization technique could also be applied to the application itself to exploit the application-level knowledge so that the spanning tree could be better constructed. It will be detailed in the following section 6.4.1.

6.4 Application-level Techniques to Leverage Multithreaded Runtime

In addition to exploring optimizations that could help the runtime performance, we also examine what we can do in applications to take advantage of the new capabilities provided by the multithreaded runtime. In this section, we describe such techniques in two aspects: one is to utilize a language construct that is designed for the multithreaded runtime; the other is to expose the dedicated communication thread to developers. Then by exploiting the thread, for example, we can improve the responsiveness of asynchronous non-blocking collective communication.

6.4.1 Utilizing *NodeGroup* Construct to Improve Performance

For large-scale parallel applications, the best performance would not generally be achievable without iterative optimizations from the application itself to its underlying language runtime system. With this multithreaded runtime system, its new features such as the shared-memory address space on a SMP node could be exposed via language constructs while the same programming model is maintained and the application should be executed correctly with or without the multithreaded runtime. Therefore, effectively using those language constructs in the application codes is an important way to further optimize overall program performance.

In CHARM++, *NodeGroup* is a language construct representing a collection of parallel objects with each SMP node having just one object indexed by the rank of the SMP node. When the message associated with a *NodeGroup* object is received on a SMP node, it will be pushed into a node-level queue that is shared by all PEs on the node. Any idle PEs on the SMP node could process the message and perform the associated computation at that time. In contrast, the message receiving of other types of objects in CHARM++ is only processed by a certain PE designated by a field in the message header. So, different from the handling of OpenMP-like intra-node parallelism described in chapter 5 which targets only at the parallelization within the computation, this *NodeGroup* object could also be utilized to parallelize receiving messages that are serialized on a PE.

Using *NodeGroup* is portable across non-SMP mode and SMP mode because the message scheduling and computation triggering mechanism are not changed, and the non-SMP mode can be viewed as a special instance of SMP mode as a SMP node just consists of one PE. However, the computation needs to be re-entrant because different entry methods of this *NodeGroup* object can be simultaneously executed by different PEs on a node. This can lead to performance degradation.

In the following texts, we describe two use cases of *NodeGroup* in NAMD that help to improve the overall performance. First of all, during the PME calculation, a PME object will receive tens of messages each containing some FFT transposed data that are deposited to non-overlapping memory chunks of the PME object. Since the PME object is not of *NodeGroup* type, those tens of messages will be processed in serial on the PE that hosts the PME object. Through the Projection tool, in large-scale runs when the PME calculation is on the performance critical path, we observed that the processing of those tens of messages took relatively long time and there are idle PEs during the same time. As a result, it is applicable to utilize *NodeGroup* object to parallelize the message receiving process. We create a *NodeGroup* object associated with the PME objects. When a PME object is created on a PE, it will register itself with the local *NodeGroup* object into a hashtable using its object index as the key on the SMP node. When the transposed message re-routed to the *NodeGroup* object is received, the index of the PME object will be extracted and the associated memory deposit will be executed on the idle PE that grabs the message from the shared node-level queue that is dedicated to *NodeGroup* objects' message. Combining this optimization with the one described in chapter 5 that has been applied to PME computation has shown benefits to NAMD's overall performance.

The second use case of utilizing *NodeGroup* is related with optimizing a set of simultaneous multicasts and reductions in NAMD. At the beginning of each timestep of NAMD, each PE of a subset of all PEs will multicast atoms' information including coordinates, velocity etc., to multiple destination PEs (a.k.a. "Patch" objects to "Proxy" objects communication in NAMD) roughly at the same time. After the force calculation, the results will be reduced from "Proxy" objects to "Patch" objects. Obviously, the "node-aware" communication optimization mentioned in section 6.3.3 could be applied to each multicast and reduction in order to reduce the communication thread load. However, since multiple multicasts or reductions are happening simultaneously, the application-oblivious way of constructing spanning tree in the runtime system could result in unbalanced load among all the PEs that participate in multicasts. Taking the multicast as an example, a PE could be selected multiple times as an intermediate PE of the spanning tree, which will perform extra work as forwarding multicast messages down the tree and to PEs on the same SMP node that are also the destinations of the mutlicast. Consequently, this intermediate PE becomes overloaded in serializing the distribution of multiple multicast messages, which in turn delays the start of simulation computation. Furthermore, there is no global barrier after this communication phase implying that the multicast message may be received in the middle of computation which is triggered by the receipt of an earlier multicast message. In such scenario, the multicast message will be blocked for forwarding until the computation finishes. To attack those two issues, we construct node-aware spanning trees at global synchronization points in NAMD such as the time for load balancing for those multicasts (reductions use the same trees), and we take advantage of *NodeGroup*'s capability of processing messages on any idle PEs to avoid message blocking. Specifically, when constructing the spanning tree, we maintain a counter on each PE that records the number of times this PE has acted as an intermediate tree node of the node-aware spanning tree. When picking up the intermediate tree node, we will select the PE that has the lowest value of this counter. The way of utilizing *NodeGroup* to process messages is similar to the one described in the first use case.

Table 6.5 shows the NAMD performance before and after utilizing *NodeGroup* objects to perform the multicasts and reductions. The results are obtained on the JYC machine running NAMD with three representative molecule systems–DHFR, Apoa1 and 1M-atom STMV with their default simulation parameters.

Figure 6.9 plots the performance speedup based on the results in table 6.5. It is clear that NAMD achieves better performance generally after utilizing the *NodeGroup* for multi-

#cores	#SMP nodes	DHFR		Apoa1		1M-atom STMV	
		orig.	opt.	orig.	opt.	orig.	opt.
256	32	3.02	3.11	9.22	8.67	94.83	95.05
512	64	1.79	1.61	4.93	4.59	48.15	48.17
1024	128	1.42	1.33	2.96	2.66	25.43	24.86

Table 6.5: NAMD Performance (ms/step) before and after Utilizing *NodeGroup* Construct

casts and reductions. Comparing the results of 1M-atom STMV with the other two systems, we find its performance gain is generally less because 1M-atom STMV is a larger molecule system, and it leads to more computation per PE. In NAMD, more computation per PE usually means the message latency is more effectively hidden. As this optimization targets at reducing the message latency, it becomes less beneficial for larger molecule systems than it becomes for smaller ones for runs on the same number of cores. The performance degradation for DHFR on 32 SMP nodes is not clear at this point, requiring further investigation.

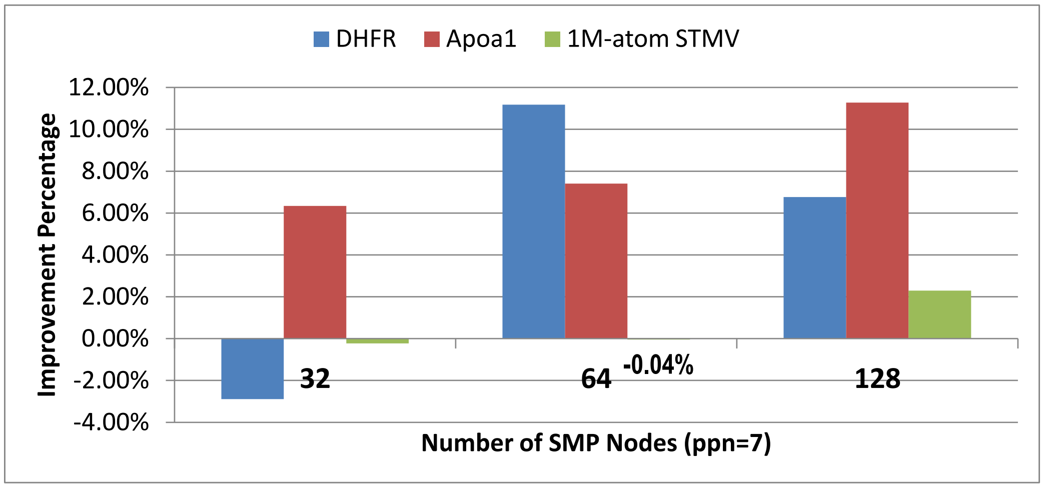

Figure 6.9: Performance Speedup of Utilizing *NodeGroup* Construct in NAMD

6.4.2 Exploiting the Dedicated Communication Thread

In the preferred multithreaded runtime design, we dedicate one logical CPU to communication as continuously polling the network message (i.e., driving the network progress engine) for multiple reasons as analyzed in section 6.1.2. If exposing this dedicated communication thread inside the runtime system to application developers, how can developers take advantage of it to improve the application performance further or implement new features that are difficult to be incorporated into worker threads? This section is dedicated to answering the question by by examining use cases we develop in the context of SMP mode of CHARM++.

Considering that exposing the communication thread to user programs provides additional computing power, but the thread is totally responsible for driving the progress of communication, which is critical to the communication performance, the main idea to exploit the dedicated communication thread is to *offload some work from worker threads to the communication thread* and *the work should not degrade the responsiveness of the communication thread to network messages*.

In the SMP mode of CHARM++, if a *NodeGroup*'s message is attributed with "immediate", then when this message arrives on a remote SMP node, instead of being enqueued into a worker thread and processed by that worker thread later, it will be enqueued into a special dedicated message queue, and only the communication thread polls the queue and executes the associated work. As a result, "immediate" messages provide a way of offloading the work to the communication thread. However, local "immediate" messages involve memory allocation and deallocation for messages themselves as well as message packing and unpacking, so to avoid such overhead, we develop simpler APIs that reuse the execution mechanism of "immediate" messages but recycle fixed-size notification messages that are used to trigger the computation on the communication thread. The message is made fixed-size thanks to the shared-memory address space shared among PEs on a SMP node where the message only contains a function pointer variable in addition to the general message header of constant size. To ensure the execution compatibility of the same codes in non-SMP mode of CHARM++, a PE will also poll the special dedicated message queue and process the "immediate" message.

With the above mechanisms, we describe two use cases that exploit more benefits from the multithreaded runtime in applications from the perspective of utilizing the dedicated communication thread. In particular, we illustrate the optimization that can improve the responsiveness of asynchronous collective communication using the reduction idiom as a representative.

First, the dedicated communication thread is exploited to **overlap GPU and CPU computation**. NAMD has been optimized for hybrid clusters installed with NVidia GPUs [65, 66, 67]. Specifically, the non-bonded computation of NAMD is offloaded to GPU using CUDA [68] while the bonded and PME computation are still performed on CPUs. Although CUDA kernel function calls are non-blocking, the PE (i.e., the worker thread) that makes the subsequent memory copy from the GPU to the CPU as retrieving the result will be blocked until the kernel finishes. Consequently, the worker thread remains idle during this time. It would be better if these CPU time were utilized for non-bonded and PME computation. Although we can make the memory copy later so that other type of computation could be executed on the CPU, the presence of multiple CUDA kernel calls each of which requires a memory copy at the end of the kernel to transfer the computed results and the restriction on concurrent CUDA kernel calls much complicate the coordination between the timing of making these memory calls and that of performing other types of computation. In contrast, exploiting the dedicated communication thread will greatly simplify the job. Considering that NAMD enters the computation phase at this particular time, and there are no more communication happening until the end of all computation of this timestep, we offload all the submissions of CUDA kernels and the device-to-host memory copies to the local communication thread while not affecting the communication performance. As a result, the bonded and PME computation on CPUs are overlapped with the non-bonded computation on the GPU without bothering worker threads to coordinate the device-to-host memory copies and other types of computation.

Secondly, taking advantage of the dedicated communication thread can **improve the responsiveness of asynchronous non-blocking collective communication**. In CHARM++ applications, the collective communication is asynchronous as well as non-blocking. When the collective completes, a function callback will be triggered to perform the computation on the collected data. Generally, the collective communication is usually performed via a spanning tree so that it consists of a number of intermediate steps, each of which requires processing messages and performing the collective operation on the data it has collected so far. Therefore, those intermediate steps of such collective communication may well overlap with other computation happening on every PE. If the message of an intermediate step arrives on a PE in the middle of computation, then the message will not be responded until the computation completes. Such a delay of one intermediate step is likely to cause a cascaded effect on the following intermediate steps, leading to a significant delay of the final completion of the collective communication. If the collective communication is on the performance critical path of the application, then the application performance will be degraded accordingly.

To attack this issue, the communication thread could be utilized to help process the intermediate message so that the intermediate step of the collective overlaps with the computation on the PE on which the step is supposed to be processed. As a result, the asynchronous non-blocking collective communication will have no delay and better responsiveness. However, such optimization has to satisfy the following conditions:

- The collective operation associated with the intermediate message should be small enough so as not to block the network progress engine for a long time. Otherwise, the overall communication performance would suffer.

- The associated collective operation should be thread-safe because the object touched by the operation is now possibly accessed by two threads: one is the worker thread to which the object is distributed, and the other is the communication thread.

Ideally, the language runtime system should automatically select a collective communication that will be applicable to utilize the communication thread for offloading intermediate steps and will benefit from the optimization. But the above conditions make it difficult for the runtime to do this automatically, especially to satisfy the requirement of thread-safety. Therefore, application developers who have more knowledge about the application and its performance issues have to determine whether this optimization could be applied or not.

We use a synthetic iterative benchmark that involves an asynchronous collective reduction to demonstrate the performance benefit of exploiting the dedicated communication thread. In each iteration of the benchmark, an asynchronous collective reduction is performed at the beginning and the reduction callback is supposed to be triggered on PE 0.

Immediately after the reduction, each PE will consecutively perform two computation kernels. We will benchmark the total time from the beginning of the iteration to the end of this asynchronous collective reduction in this environment where the collective communication overlaps with computation.

Figure 6.10 shows the timeline of a step in this benchmark obtained from the performance visualization and analysis tool “Projections”. The figure includes the activities of two full SMP nodes, each node consisting of 4 PEs. Every line represents a thread, and bars of different colors on each line represent different activities. As we extended the “Projections” tool described in section 6.3.1, every fifth line shows the activity of the communication thread. Note that the bars on the communication thread line refer to the normal message sending and receiving activities.

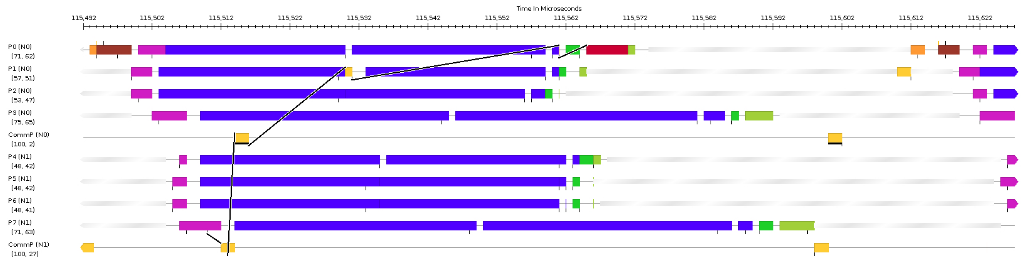

Figure 6.10: The Timeline before Exploiting the Dedicated Communication Thread

We have also applied the “node-aware” communication optimization mentioned in section 6.3.3 to this collective communication. Basically, the reduction is first finished within a SMP node and the intra-node reduction is handled by a “NodeGroup” object introduced in section 6.4.1. After the completion of this part, the “NodeGroup” object will send the partially reduced result to the parent “NodeGroup” object on another SMP node. Finally, the reduction callback will be triggered on PE 0 as represented by the red bar on the first line in figure 6.10. The solid lines depicts the communication path of the final contribution step of the reduction and the path to trigger the reduction callback. It is clear that the reduction callback happened after the two computations represented by blue bars on PE 0. Note that the reduction calculation of the final contribution step on PE 1 (shown as the small yellow bar) finishes after the beginning of the second computation on PE 0. As a result, the execution of the reduction callback on PE 0 is delayed until the completion of the second computation. We also notice that the message that triggers the final reduction calculation arrives much earlier on the communication thread. However, the calculation, which is handled by a "NodeGroup" object and which could be executed on any worker threads on the node, is not executed until the earliest completion of the first computation kernel on every worker thread.

After exploiting the communication thread to execute the final reduction calculation, i.e., the one represented by the small yellow bar on PE 1 in figure 6.10, the message that triggers the reduction callback is sent much earlier than the second computation on PE 0 as in the middle of the first computation. It is shown by the dotted line in figure 6.11.

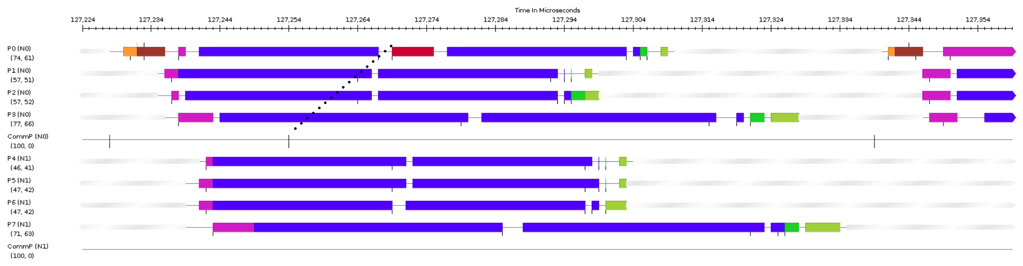

Figure 6.11: The Timeline after Exploiting the Dedicated Communication Thread

Because the "immediate" message as used to enable this optimization are not traced in the tracing framework of "Projections" on the communication thread, we are not able to visualize the final reduction calculation as same as the yellow bar in figure 6.10, and trace back to the point that triggers the reduction callback on PE 0. However, we know such point is very close the one when the final partial remote reduction message is received on the communication thread. With the mark of the receipt of this message on the communication thread, we artificially added the dotted line in figure 6.11 to represent the communication path. The tracing of "immediate" message will be left in the future work.

Thanks to the earlier processing of the final reduction calculation on the communication thread, the average benchmark time is reduced from 67.39us to 36.95us in this case. Clearly, exploiting the dedicated communication thread of the multithreaded runtime has improved the response time of the asynchronous collective reduction.

7 Multicore-aware Dynamic Load Balancing

Nowadays, large-scale scientific and engineering applications become increasingly dynamic and complicated consisting of several different physics modules. Those applications, during the execution, usually experience load imbalance leading to serious performance issues. Thus, adaptive dynamic load balancing techniques are applied to attack those challenging issues [69, 70, 71]. However, to the best of my knowledge, those techniques could become ineffective in some cases as they have not considered the architectural characteristics of multicore chips. In addition, the multithreaded runtime exhibits new properties that could be taken into account to better the load balancing strategy. So, in this chapter, we first motivate the development of multicore-aware load balancing strategies in more details. After describing the extension we make to the existing load balancing framework in CHARM++, we present a set of new load balancing strategies and demonstrate them to be useful.

7.1 Motivation

Many high-performance parallel scientific and engineering applications require many iterative steps for simulating the evolution of the system or for refining the result until reaching acceptable error boundaries. As a result, those programs show a “persistent” property in computation and communication such that the near future reflects the near past [72]. To take advantage of this property, measurement-based strategies have been used to attack load imbalance issues, such as ones that are used in CHARM++ applications [73, 74, 23]. The key idea behind the measurement as to characterize the load of a task is to use the CPU execution time as the metric for the computation load, and use a certain not-practically-measured cost per message as the metric for the communication load.

However, with the multithreaded runtime on multicore chips, those two metrics may not accurately represent the load of a task because of the following factors:

- **Lower cost for intra-node messages:** in the multithreaded runtime, the intra-node communication is performed via the message pointer thanks to the shared memory address space among PEs on a SMP node. In contrast, the inter-node communication has to go through the network stack, involving memory copy of messages. As a result, the cost of an intra-node message is much lower than that of an inter-node one. Furthermore, even in the traditional runtime mode, say the non-SMP mode of CHARM++, the lower-level communication library such as MPI may also have exploited the shared-memory of a physical node to reduce the latency of communication within a physical node. Therefore, the cost of communication between a pair of cross-physical-node PEs is more expensive. In short, on one hand, for load balancing strategies that consider the communication cost, we can not simply assume flat cost per message and characterize the communication load based on the volume of messages. We have to take the cost difference between the intra-node and inter-node messages into account. On the other hand, because of such different cost in different types of messages, a more effective load balancing strategy should always consider the balance in communication on multicore-based parallel machines.

- **Shared physical resources:** on multicore chips, multiple cores share physical hardware resources such as the memory controller (equivalently as sharing memory bandwidth), L2 or L3 cache, or even function units. For example, in the AMD Interlagos processor used by BlueWaters, every two physical cores will share the same floating-point function units. Additionally, simultaneous multi-threading (SMT) becomes popular again on current multicore processors. For example, on BlueGene/Q, the latest generation from IBM BlueGene supercomputers' family, every core can be configured to have up to four hardware threads. In this case, many more physical resources are shared among those hardware threads (i.e., logical CPUs from the perspective of OS). Because of this sharing on multicore platforms, the contention could occur on the underlying resources and affect the application performance [75]. This implies the execution time of one piece of work could change noticeably from one PE to another depending on the computation that is happening on neighboring PEs. Consequently, just using the execution time to characterize the computation load is not adequate enough to predict the resource contention on the PE this computation is to going to be migrated to. We have to record additional information as well which could reflect the degree of resource contention. For example, we could record the number of L2 cache misses to reflect the load on L2 cache; we could also record the number of floating-point operations to reflect the load on the floating-point functional units. Correspondingly, the new additional information should be taken into account in load balancing strategies.

- **Asymmetry among PEs introduced by runtime system design:** in the preferred design of the multithreaded runtime system, we have separated the computation and communication into different threads. In particular, a logical CPU is dedicated to the communication thread. Considering the worker threads that share physical resources with the communication thread, those worker threads will have more computing resources available to them, but will have more contention in memory-related resources than other worker threads of the same SMP node because the communication thread involves very few computation (more accurately, there is almost no floating-point operations on it except when it is exploited in the application level to offload some ceratin amount of work) but a significant amount of memory-related operations such as memory allocation, memory copy etc. Consequently, there is a certain degree of asymmetry among worker threads in the multithreaded runtime. Such asymmetry could be serious enough to distort the perceived execution time (i.e., the computation load) in practice.

To the best of my knowledge, existing measurement-based load balancing strategies hardly consider the above factors. Given that those factors affect the quality of load balance, we are motivated to develop new and more effective load balancing strategies that incorporate the above factors in the case of running a multithreaded runtime system on multicore-based parallel machines.

7.2 Background of Load Balancing Framework in CHARM++

CHARM++ has a built-in automatic measurement-based load balancing framework, enabled by the fact there are a large number of parallel objects (i.e., chares or elments of a chare array) typically available to map to existing PEs, and the runtime system can migrate those objects at certain synchronized points. Based on this framework, we investigate how to make load balancing strategies more effective on multicore-based clusters by considering the motivating factors described in the previous section.

In CHARM++'s load balancing framework, to track the computation and communication load of parallel objects, every PE has a load balancing database that accumulates the CPU execution time spent on each parallel migratable object and records its communication information. Specifically, as the computation of an object is scheduled by runtime system via the message, its beginning and end points are known to the runtime. The runtime could then obtain the execution time of the computation by subtracting the two timestamps. In addition, when a message is sent from one object, the runtime knows the destination object the message is supposed to be received by and the message size.

When the application reaches the load balancing point, either manually specified by the codes in the application or automatically specified by the load balancing period, a centralized or distributed load balancing strategy will be invoked to balance the load across all PEs based on the collected load information. A centralized strategy will combine the load balancing databases of all PEs into a single global database, while a distributed strategy will divide all PEs into multiple groups, each of which has a root PE that will gather the load balancing database from all PEs in the group. When writing a new load balancing strategy for CHARM++, developers only need to implement the specific algorithm that operates on the load balancing database to make load balanced across the PEs whose load information is included in the database. Thus, it is convenient for researchers to experiment and evaluate load balancing strategies in CHARM++.

After the load balancing, an object could be migrated from one PE to another, facilitated by the packing/unpacking framework, a.k.a "pup" in CHARM++ [39]. This framework is designed to describe the in-memory layout of an object, and can be extended to provide services to any operations that require a traversal of the object state. The "pup" framework has been used in other situations such as checkpointing for fault tolerance [76, 77], out-of-core execution in the emulation part of BigSim [78].

7.3 Extension to the Existing Load Balancing Framework

In order to record the information that could characterize the degree of contention on the shared physical resources, we have to extend the existing load balancing framework in CHARM++ in the following aspects:

To begin with, we need to update the existing data structure of the load balancing database so that it is flexible enough to store user-specified information of arbitrary size. Note that different shared physical resources require different information to characterize them, and different load balancing strategies targeted at reducing the contention of the same shared physical resources may also require storing different information in the database. As a result, we could not simply add a fixed number fields to the database data structure. To address this issue in flexibility, we add a field representing a re-sizeable memory buffer into the definition of the database data structure and develop APIs that developers could use to register the user-specified information with the database and retrieve its value from the database. With this extension, the database could not only serve the load balancing strategies that are developed in this thesis, but also serve other drastically different strategies, such as those that target at power efficiency [79, 80].

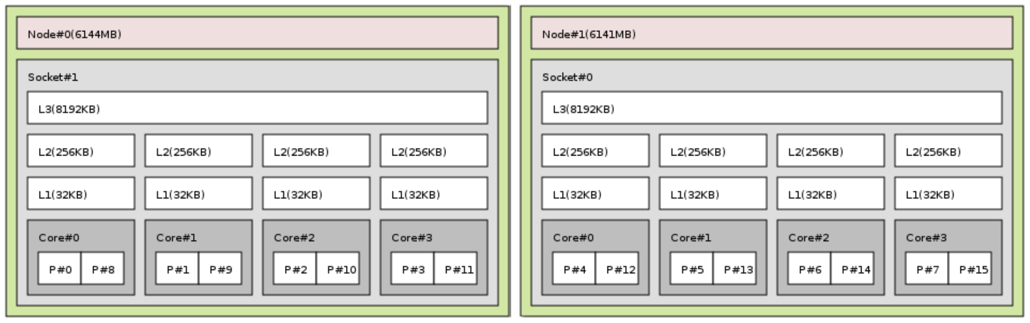

Figure 7.1: The CPU Topology of Intel Xeon E5520 Visualized by HWLOC

Secondly, in order to identify which physical resources are shared among PEs, the runtime has to know the CPU topology of the physical node and the affinity mappings from the CHARM++ PEs to the logical CPUs. Based on the HWLOC [81] library, a cross-platform CPU topology detection library, we develop a lightweight library that could be used to query in runtime whether two logical CPUs are sharing the cache, or whether they are two hardware threads running on the same core etc.. The HWLOC library also provides a tool to visualize the CPU topology of a multicore processor exemplified by figure 7.1, in which two logical CPUs share a physical core and four physical cores in a socket share the 8MB L3 cache. As the CPU affinity is automatically set by the CHARM++ runtime or explicitly set by users, we keep this affinity information in memory for later queries in the mapping of CHARM++ PE to the logical CPU. Combining these two pieces of information, we are able to know whether two CHARM++ PEs are sharing physical resources or not and what type of resources they are sharing.

Finally, we are facing the problem in what extra information is needed to be recorded in the load balancing database in order to characterize the degree of contention in the shared physical resources. As modern processors are equipped with hardware performance counters, those counters can be used to reflect the amount of usage of certain physical resources by the computation. Therefore, based on the usage information, one could infer whether performing a piece of computation simultaneously with some computation creates more contention on a certain type of physical resources than that with some other computation. To retrieve the value of hardware performance counters, we use the widely-used cross-platform library Performance Application Programming Interface(PAPI) [82]. By reading the names of PAPI counters from an input file on PE 0 and then broadcast the PAPI information, we allow flexibility in recording different PAPI counter values for different load balancing strategies in the CHARM++ runtime.

In summary, extending the load balancing framework with the ability of recording information of arbitrary size, querying the type of physical resources two PEs are sharing and obtaining the values used to characterize the degree of resource contention lays the foundation to develop and evaluate multicore-aware load balancing strategies which are described in the following sections.

7.4 Awareness of Message Latency Difference in SMP-mode

In the SMP mode of CHARM++, the memory pointer to a message is the entity that is transferred from one PE to another for message passing on a SMP node thanks to the shared memory address space on the node. In contrast, the inter-node message involves a memory copy of the whole message payload. As a result, the intra-node message incurs less cost than that of inter-node messages. This section thus is devoted to the load balancing strategy that considers the communication cost difference.

Intuitively, given two load balancing strategies that result in the same computation load distribution for a parallel program, then it is likely the strategy that causes less inter-node communication leads to a better performance for the application because the overall object-level communication pattern and volume are exactly same in the two load balancing strategies.

Following this intuition, we develop a load balancing strategy that adjusts the mapping of objects based on the communication cost after the computation load is first balanced. As the strategy is designed to improve the quality of the load balance of the application based on the previous load balancing strategy, we categorize such a strategy as "refinement"-based. The key behind the strategy is the cost function for the communication associated with an object as expressed in the following:

$$cost(c, p) = cost_l(c, node(p)) + \alpha \times cost_r(c, node(p)) \tag{7.1}$$

In expression 7.1, $cost(c, p)$ is the total communication cost of a migratable CHARM++ object c if it is mapped to PE p. It consists of two parts: $cost_l(c, node(p))$ represents the communication cost of this object c with other objects on PEs of node $node(p)$ that PE p belongs to (i.e., the cost of intra-node communication), and $cost_r(c, node(p)$ represents the inter-node communication cost with objects on every other nodes. The coefficient α controls the weight inter-node communication cost has over the intra-node communication. In the actual implementation, we use the number of intra-node messages and inter-node messages this object c incurs as $cost_l(c, node(p))$ and $cost_r(c, node(p)$ respectively. As for α, we set it to the ratio of average latency of inter-node messages over that of intra-node messages. In the future, we will make α more accurate and adaptive by considering the topology of all SMP nodes inspired by works [83, 84] and the actual message sizes that the application uses during the execution.

Based on this cost function, we implement this "refinement"-based strategy with a greedy heuristic for adjusting the objects' mapping in order to reduce the total communication cost as shown by algorithm 1. The algorithm is greedy in nature because we start the adjustment from the heaviest communicating object and then tries to exchange it with an object on a remote node that will make this object has the least communicated cost. Given

Algorithm 1: SMP-node-aware Communication Refinement Strategy

```
Input: Load information of migratable objects of all PEs
Output: PE map of Objects to be migrated
initialize $objSet$ with all migratable objects;
while $!objSet.empty()$ do
    retrieve and delete the heaviest object $candidate$ in terms of the communication
    cost function from $objSet$ find the remote SMP node $rnode$ that will make the
    communication cost of $candidate$ least among the remote nodes that $candidate$
    communicates with;
    find the list of objects $objList$ that do not communicate with $candidate$ on
    remote node $rnode$;
    sort list $objList$ decreasingly according to the number of messages that each
    object has with object $candidate$;
    foreach object obj in $objList$ do
        // Exchanging with objects on the same SMP node
           does not help to reduce the total communication
           cost
        if exchanging candidate with obj makes the computation load of PEs
        $candidate.pe$ and $obj.pe$ still within the range of balanced load then
            migrate $candidate$ to $obj.pe$;
            migrate $obj$ to $candidate.pe$;
            update the communication cost of all objects that have communicated
            with $candidate$ and $obj$;
            remove $obj$ from $objSet$;
            break;
```

n objects and M SMP nodes, the algorithm presents a time complexity of $\mathcal{O}(n^2 \log n)/M$ as $\mathcal{O}(n \log n)/M$ represents the cost of sorting the list of objects $objList$ in each iteration considering the average number of objects per SMP node is n/M. We could achieve an amortized $\mathcal{O}(\log n)$ time in removing an arbitrary object from $objSet$ while still be able to retrieve the heaviest object from $objSet$ in $\mathcal{O}(1)$ with a data structure that combines "hashtable" and "max heap". Specifically, during the initialization of $objSet$, we will populate both the "hastable" and the "max heap". When retrieving the heaviest object from $objSet$, we pop the object from the "max heap" continuously until the object exists in the "hashtable". When removing an object from $objSet$, we just delete the object from the "hashtable" and leave the "max heap" untouched.

In addition to the greedy algorithm proposed above, we can use existing graph partition libraries such as METIS [85] or SCOTCH [86] to partition the object communication graph into a number of parts same with the number of SMP nodes by considering the object load and the communication cost at the same time. Afterwards, within a SMP node, we can again use any existing load balancing strategies to balance the computation load without considering the communication cost. Such a two-level load balancing approach has already been investigated in work [74, 87] and in the context of fault tolerance [88] but for different purposes as to reduce the memory consumption of the load balancing strategy and the message logging mechanisms for fault tolerance, respectively. In the future, we plan to port those work to serve this SMP-node-aware communication load balancing strategy and compare them with the strategy described in this section.

7.5 Awareness of Shared-Resource Contention Among PEs

There is a distinctive architectural difference between multicore chips and the traditional uni-processor chips in that several physical cores could share certain physical resources, such as the last level cache. As motivated in section 7.1, the load balancing strategy could be more effective by considering the contention as a natural consequence of resource sharing. In this section, we present a new load balancing strategy to show our approach to considering the contention factor. In practice, there is contention for multiple types of resources. In this work, we focus on considering the contention on the shared last level cache in load balancing.

As for the contention in the last level cache, it is straightforward that putting objects that perform computation-intensive work with those that perform memory-intensive work together on different PEs, but on the same die, could be very beneficial because the resource required by each objects is complementary to each other implying less contention in each type of resources including the shared cache. This is key idea behind the new load balancing strategy we develop.

Another important aspect of the new strategy is how we represent the computation load of a parallel object in the load balancing database when incorporating the additional metrics that will indicate the degree of contention for a certain resource that is targeted to reduce the contention. We think it very difficult to find a general composite single value based on multiple metrics to accurately represent the computation load. Therefore, we keep those metrics separately, but with different priorities. The execution time is still the first-order metric for the computation load of a parallel object. The other metrics such as the amount of cache misses are considered to be second-order. Based on this cost model, we develop a group-based greedy refinement load balancing strategy that will adjust the mapping of objects based on the second-order metrics after the load is balanced according to the first-order metric. Specifically, as for the new load balancing strategy we develop that considers the contention on the shared last level cache, we refine the object mapping according to the amount of cache misses that each object incurs after objects are first balanced according to the CPU time an object consumes.

It is worth mentioning that this strategy could be applied into the runtime both in the multithreaded mode (i.e., the SMP mode) and in the traditional default mode (i.e., the non-SMP mode) because this load balancer strategy attempts to address the physical resource contention on multicore platforms without any references to the features of the multithreaded runtime.

The new load balancing strategy consists of three steps: the first two steps are shown by algorithm 2 and the third one as the main part is illustrated by algorithm 3. First, the strategy is performed based on groups of PEs. PEs in each group share the same physical resource targeted for contention reduction while each group has its own dedicated such resource. For example, considering the last level cache (L3) illustrated in figure 7.1, PE 0 to

PE 3 are in the same group while PE 4 to PE 7 are in another group. Therefore, the first step shown in algorithm 2 is to divide all PEs into such groups utilizing our lightweight library mentioned in section 7.3 that provides the functionality of querying the topology information of CHARM++ PEs during the execution. The load of each group is also accumulated at the same time.

Algorithm 2: Group-based Greedy Refinement Load Balancing Strategy-Part1

```
Input: Load information of migratable objects of all PEs
Output: PE map of Objects to be migrated
/* Step 1: Divide all PEs into groups of PEs that share
   the target physical resource                              */
foreach PE $p$ in all PEs do
    $grpId$ = getGroupId($p$);
    Update the load of GroupLoad[$grpId$] with the load of PELoad[$p$];
/* Step 2: Set target load                                   */
// first-order metric generally refers to the CPU
   execution time
Sort groups decreasingly according to the load by the first-order metric;
foreach group $grp$ in all PE groups $allGrps$ do
    // second-order metric generally refers to the one
       that characterizes the shared resource whose
       contention is targeted to be reduced, such as the
       number of L2 cache misses
    Sort all objects in $grp$ increasingly according to the load by the second-order
    metic;
Obtain and relax the $maxPELoad$ and $minPELoad$ in terms of first-order metric;
// Calculate the $targetMaxLoad$ and $targetMinLoad$ in terms
   of second-order metric
Obtain the average load $avgLoad$ of all groups in terms of second-order metric;
$targetMaxLoad$ = $avgLoad$*(1+threshold);
$targetMinLoad$ = $avgLoad$*(1-threshold);
```

In the next step of the new strategy (i.e., step 2 shown in algorithm 2), we will set the target load for each metric. Because this strategy is to refine the object mapping in terms of second-order metrics, we will relax the range of the load allowed in terms of the first-order metrics. Currently, we calculate the range of target load based on the average load across all groups with a relaxation percentage defined by a statically pre-defined parameter.

Finally, as the third step of this new strategy shown in algorithm 3, we takes a greedy approach to adjusting the load among different groups. Basically, we continuously try to exchange a pair of objects, each from a more loaded group and a less loaded group in terms of second-order metrics, respectively as long as the exchange will not create a load imbalance according to the relaxed range of load in terms of the first-order metric. In addition, in the more loaded group, we select the candidate object for exchange in the order of decreasing load (i.e., select the object that incurs the most load first). In contrast, in the less loaded group, we select the candidate object for exchange in the order of increasing load (i.e., select the object that incurs the least load first). Given n migratable objects handled by the strategy, this step presents a time complexity of $\mathcal{O}(n^2)$ in the worst scenario in which every pair of objects is tried. In general, if we make a successful exchange, the two objects will not be considered in later exchanging attempts. Therefore, in the best scenario, each object will be exchanged with one another in $\mathcal{O}(n)$ time.

We demonstrate the benefits of this group-based greedy refinement load balancing strategy via a synthetic program. In the program, every parallel object iteratively performs some certain type of computation. Half of those objects will perform more memory-intensive computation and they will be mapped to the first half of total PEs. In contrast, the remaining half of PEs will host the other half of the objects that perform, however, less memory-intensive computation. We performed the test on a single node that is illustrated by figure 7.1 with 8 CHARM++ PEs. We simply created 8 objects so that each PE has one such object. The fist 4 objects performing the memory-intensive are mapped to PE 0 to PE 3, while the remaining 4 objects are assigned to PE 4 to PE 7 at the beginning of the execution.

As we intend to reduce the contention on the last level cache–L3 cache in the experimental platform, we specified the PAPI event as PAPI_L3_DCA (L3 data cache accesses) to be recorded in the load balancing database because the amount of cache access reflects the degree of contention in using cache. On this particular experimental node, the number of L3 data cache accesses is same with the number of L2 data cache misses. In other words, we could also specify PAPI_L2_DCM as the event for record.

Table 7.1 shows the average number of L3 data cache accesses and the execution time of each object per iteration of the synthetic program without any load balancing. In the

Algorithm 3: Group-based Greedy Refinement Load Balancing Strategy-Part2

```
/* Step 3: Refine the load of by trying to exchange
   between a pair of objects                                */
Initialize a deque $toExchangePairs$;
// Make the most loaded object in the most loaded group
   and the least loaded object in the least loaded group
   a pair, and push it into the deque
Push a pair ($allGrps[0].objs[numObjs-1]$,$allGrps[numGrps-1].objs[0]$) to
$toExchangePairs$;
while $!toExchangePairs.empty()$ do
    $objPair$ = $toExchangPairs.pop()$;
    // Note: the object here is ranked by the
       second-order metric
    $heavyObj$ = $objPair.first()$;
    $lightObj$ = $objPair.second()$;
    if GroupLoad[$heavyObj.grp$] $\in [targetMinLoad, targetMaxLoad]$ then
        continue;
    if GroupLoad[$lightObj.grp$] $\in [targetMinLoad, targetMaxLoad]$ then
        continue;
    if the PELoad[$heavyObj.pe$] and PELoad[$lightObj.pe$] are still
    $\in [minPELoad, maxPELoad]$ after exchanging $heavyObj$ and $lightObj$
    then
        if Exchanging $heavyObj$ and $lightObj$ balances the load in terms of
        second-order metric for $allGrps[heavyObj.grp]$ and
        $allGrps[heavyObj.grp]$ then
            Exchange $heavyObj$ and $lightObj$;
            Update PELoad[$heavyObj.pe$] and PELoad[$lightObj.pe$];
            Update GroupLoad[$heavyObj.grp$] and GroupLoad[$lightObj.grp$];
    else
        $newHeavyObj$ = /*Get the next immediate lighter object of $heavyObj$ in
        group$heavyObj.grp$*/;
        $newLightObj$ = /*Get the next immediate heavier object of $lightObj$ in
        group$lightObj.grp$*/;
        $toExchangPairs.push(newHeavyObj, lightObj)$;
        $toExchangPairs.push(heavyObj, newLightObj)$;
```

table, all PEs are divided into two groups as each group PEs share the same L3 cache. The "L3_DCA" columns show the number of L3 data cache accesses. Based on the "PE" column and "obj ID" column in each group, we come to know the object mapping to PEs. Clearly, each PE in group 0 has much more L3 data cache accesses than that of group does. Therefore, there is more contention in accessing the shared L3 cache among PEs in group 0 than that in group 1. By examining the execution time in table 7.1, the execution time of objects in group 0 is longer than that in group 1, indicating a load imbalance in the program. Since each PE just owns one object, we would think the load balancing will not be useful at all in this case to reduce the overall iteration time which depends on the maximum execution time of all objects.

PE Group 0				PE Group 1			
PE	L3_DCA	obj ID	time (ms)	PE	L3_DCA	obj ID	time (ms)
0	235,571	0	4.54	4	194	4	3.19
1	237,809	1	4.50	5	181	5	3.19
2	236,222	2	4.50	6	187	6	3.19
3	235,608	3	4.54	7	175	7	3.19
total	945,209	max	4.54	total	736	max	3.19

Table 7.1: The Number of L3 Data Cache Accesses and the Execution Time without Any Load Balancing

However, from a different perspective as considering the contention in L3 cache, we find the load in terms of the number of L3 data cache accesses in each group of PEs is quite imbalanced as the total L3_DCA in group 0 is about 1300X more than that in group 1! After applying the load balancing strategy described the in this section, according to table 7.2 that shows the performance result, object 0 which migrated from PE 0 to PE 6 exchanged with object 6 which migrated from PE 6 to PE 0 and object 2 exchanged with object 4 as well. Such migration follows the greedy algorithm 3 mentioned above as object 0 and object 2 are the top 2 objects in group 0 that incur most L3 data cache accesses while object 6 and object 4 are the last 2 objects in group 1 that have least L3 cache accesses. As a result of such migration, the total L3_DCA of two groups now become roughly the same. With the reduction in the contention in L3 cache, the execution time of the first half objects (i.e., objects from 0 to 3) per iteration is accordingly reduced by by 11.8%, from 4.54ms to 4.06ms. In addition, objects 4 to 7 experienced a slight increase in L3_DCA as each of those objects now in a PE group that have much higher number of L3 cache accesses than that before the load balancing.

Figure 7.2 shows the timeline of this synthetic program before and after applying the load balancing strategy described in this section where the blue bars represent the work

PE Group 0				PE Group 1			
PE	L3_DCA	obj ID	time (ms)	PE	L3_DCA	obj ID	time (ms)
0	241	6	3.20	4	235,138	2	4.06
1	238,015	1	4.06	5	231	5	3.20
2	226	4	3.20	6	235,233	0	4.05
3	240,024	3	4.04	7	230	7	3.20
total	478,506	max	4.06	total	470,832	max	4.06

Table 7.2: The Number of L3 Data Cache Accesses and the Execution Time after Cache-contention-aware Load Balancing

on objects that are more memory-intensive while the yellow bars represent the work on objects that are less memory-intensive. The middle part of the figure shows the load balancing period. It is clear from the timeline that the iteration time is reduced after the cache-contention-aware load balancing, demonstrating the benefits of taking shared-resource contention into account.

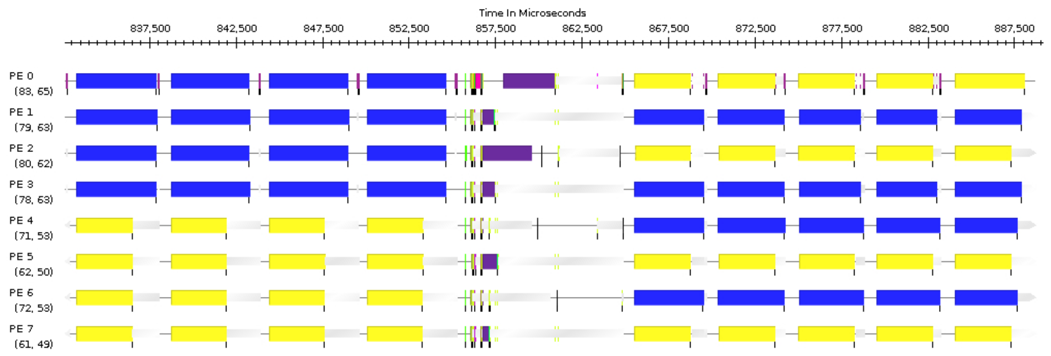

Figure 7.2: The Timeline before and after Cache-contention-aware Load Balancing

7.6 Awareness of Asymmetric PEs Introduced by SMP Design

As mentioned in section 7.1, the worker threads co-located with the dedicated communication thread in the preferred multithreaded runtime design have more computation resources than other worker threads. This section investigates its impact on the application performance and how a load balancing strategy could address this asymmetry of PEs introduced by the multithreaded runtime design.

Considering the communication thread rarely uses the floating-point function units, we

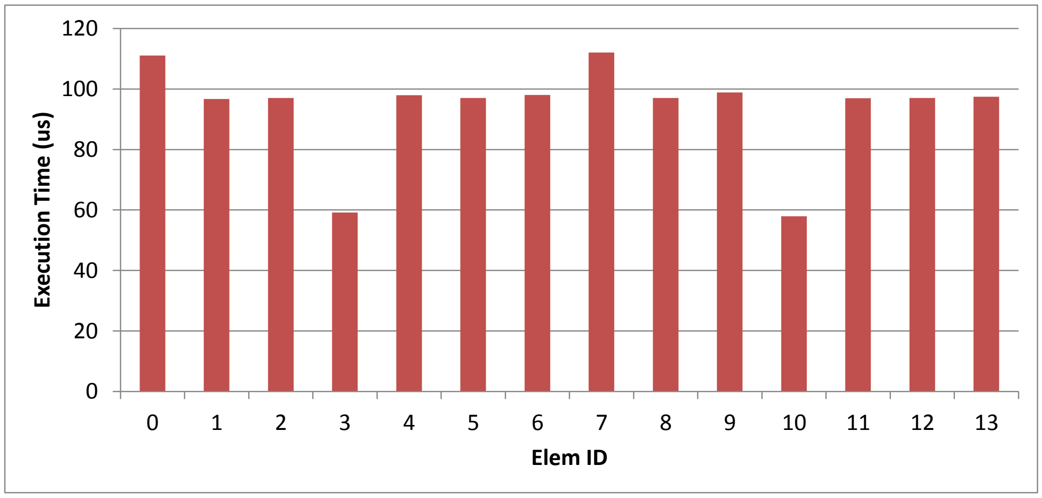

Figure 7.3: The Initial Average Execution Time of Objects

implement a synthetic program in which each element iteratively performs a pure floating-point computation kernel without any memory accesses so that the asymmetry of PEs could be more pronounced for the sake of this study. We performed the experiment on a physical node consisting of two-socket Intel Xeon E5520 quad-core chips. With two hardware threads on each physical core, there are 16 logical CPUs in total. We lauched two SMP nodes, each composed of 7 PEs (i.e., 7 worker threads and 1 dedicated communication thread), on this node, and created 14 objects hence each PE having one object. Figure 7.3 shows the average execution time of each object in microseconds. Clearly, object 0 and object 7 have the highest execution time while object 3 and object 10 have the lowest execution time. The remaining objects have the roughly same execution time.

If we refer to the amount of floating-point operations each object performs, which are obtained from hardware performance counters via PAPI library in figure 7.4, it is interesting to see the the amount of operations do not correspond well with the execution time of each object as shown in figure 7.3. Specifically, only object 0 and object 7 performed more floating-point operations than others. Given the computation is pure of floating-point operations, it is understandable that object 0 and object 7 takes longer time to complete computation. Comparing object 3 and object 10 with remaining objects such as object 2 etc., those two objects have performed almost the exact same amount of floating-point operations with others but their execution time is about 40% lower than others according to

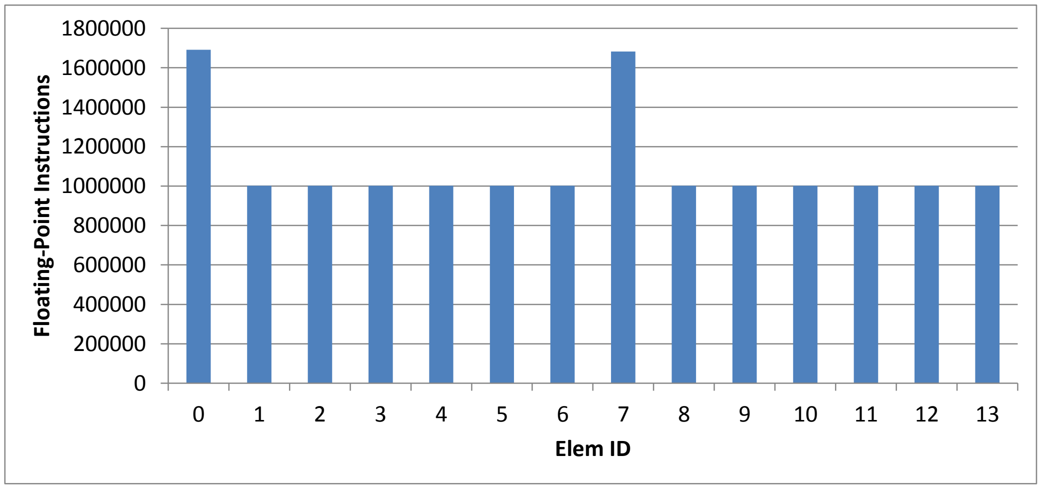

Figure 7.4: The Amount of Floating-point Operations of Each Objects

figure 7.3.

By examining the mapping of objects to logical CPUs and the topology of all logical CPUs illustrated by figure 7.5, we identified that object 3 and object 10 stay on PEs that are co-located with the communication thread (represented by the bold “C” alphabet in the figure) of each SMP node on the same physical core, respectively. Consequently, the mismatch between the execution time and the amount of floating-point operations happens due to the more floating-point function units object 3 and object 10 have access to than other objects. In summary, PEs do become asymmetric in terms of the amount of computation resources they have access to in the multithreaded runtime with a dedicated communication thread, and such asymmetry impacts the application performance to some extent.

We introduce the following strategy that makes the load more balanced across all PEs in the multithreaded runtime on multicore platforms by considering the above asymmetric-PE factor. The key observation here is that the PEs co-located with the communication thread have more computation resources, thus appearing faster than others. Therefore, when being assigned computation load, those artificially faster PEs should take more computation load. In other words, the computation load of each PE should be relative to the speed of the PE. Following this idea, we associate each PE with a factor β to represent the its speed. Then the load of one object o_i on PE p_j becomes $\beta_{p_j} \cdot load(o_i)$ where $load(o_i)$ is the CPU time consumed by this object o_i collected in the load balancing database. When this

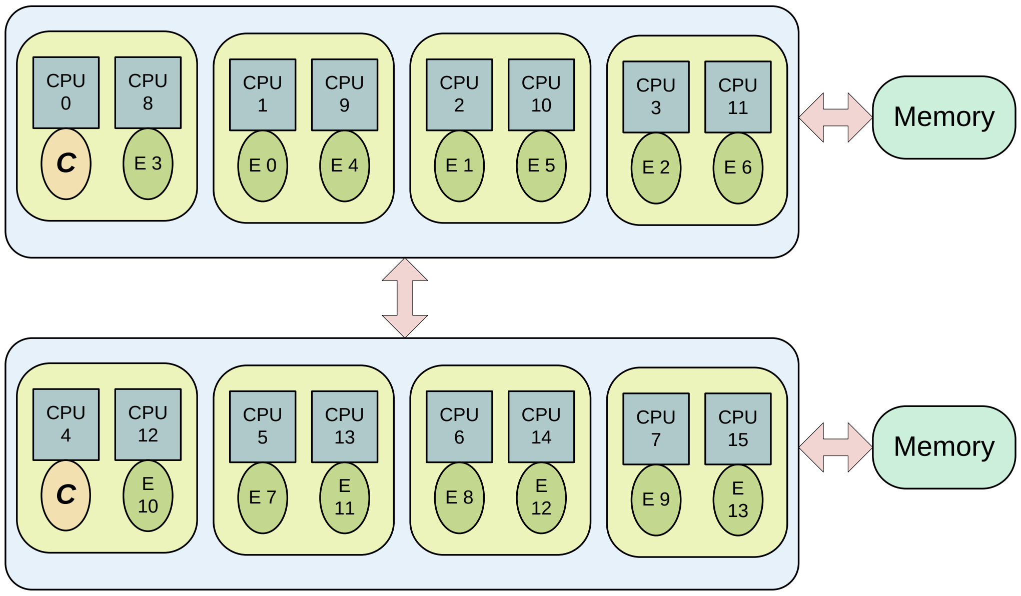

Figure 7.5: The Mapping of Objects to Logical CPUs on a Node

object migrates from PE p_j to PE p_k, the increase on the total load of PE p_k is calculated as $\beta_{p_j}/\beta_{p_k} \cdot load(o_i)$. With these cost functions, the load balancing then can reuses any existing load balancing strategies to handle the asymmetric PEs introduced by the design of the multithreaded runtime on multicore platforms, such as those in CHARM++ [39, 89].

Currently, the speed factor β of a PE is obtained in two steps. First, our lightweight library that query whether two logical CPUs share a certain type of physical resource, as part of our extension to the existing CHARM++ load balancing framework described in section 7.3, is used to differentiate PEs whether they share physical computation resources, say the floating-point function units, with the dedicated communication thread. Afterwards, we set the speed factor of all PEs that are detected not sharing computation resources with the communication thread to be 1.0. Additionally, we run a computation intensive kernel simultaneously on two different PEs as one shares the resource with the communication thread and the other does not. The ratio of the kernel execution time is used to represent the speed factor. Specifically, suppose PE p_i takes t_i to finish the kernel and PE p_j takes t_j to finish, and PE p_i shares the resource with the communication thread, then the speed factor β_{p_i} is calculated as t_j/t_i. The speed factor will then be recorded into the load balancing database.

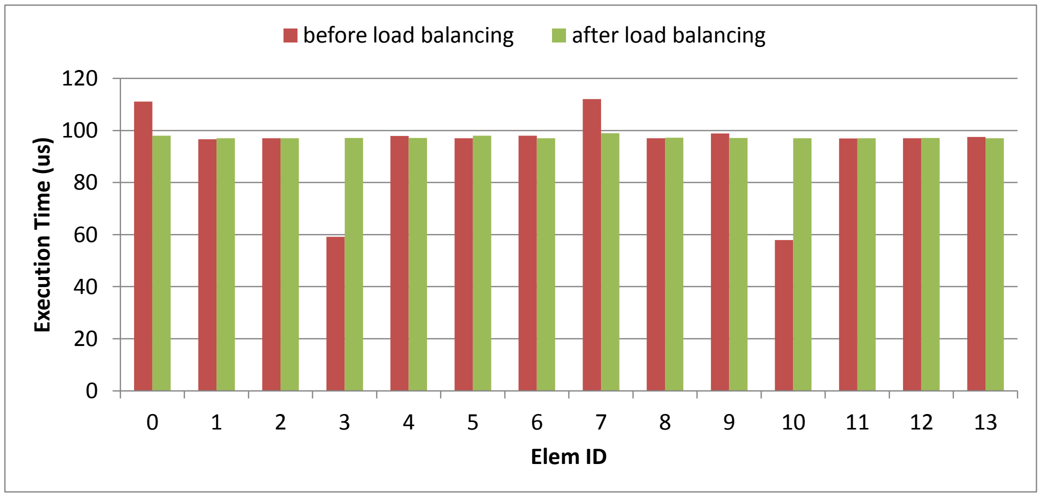

Figure 7.6: The Average Execution Time of Objects after Asymmetric-PE-aware Load Balancing

As a reference implementation of the asymmetric-PE-aware load balancing strategy, we integrated those changes into an existing load balancing strategy–"GreedyLB" [39] in CHARM++ runtime, and modified it slightly to favor migrating objects to PEs that have a larger speed factor first. Figure 7.6 shows the execution time of the objects in the synthetic program before and after the load balancing. Obviously, the execution time across all objects become roughly same after the balancing. This results from the fact that, after the balancing, object 0 and object 7 that have more floating-point operations are moved to the PEs co-located with the communication thread (i.e., CPU 8 and CPU 12 in figure 7.5) while object 3 and object 10 that are originally on those PEs are moved to others. Therefore, the expected longer execution time is reduced by the availability of more computation resources for object 0 and object 7.

8 Performance Evaluation on Scientific Applications

After the systematic performance study of the multithreaded runtime and the development of corresponding techniques to address various performance issues, in this chapter, we will show the performance results of CHARM++ scientific applications, including important production-level scientific codes running in the SMP mode of CHARM++ runtime. We compare the performance with that in the non-SMP mode to demonstrate the benefits of a multithreaded language runtime on multicore-based massively parallel machines, and to understand the conditions when it is more advantageous. In particular, we have intensively performed tests on the widely used molecule dynamics simulation code–NAMD with multiple representative molecule system inputs listed in table 8.1 on multiple parallel machines.

Molecule Name	#Atoms	Cutoff($\dot{A}$)	Simulation Box($\dot{A}$)	Default PME Frequency
DHFR	23,558	9	62x62x62	2
Apoa1	92,224	12	108x108x77	4
1M-atom STMV	1,066,628	12	216x216x216	4
100M-atom STMV	106,662,800	12	1084x1084x867	4

Table 8.1: Parameters of Representative Molecule Benchmarks Used by NAMD

8.1 Evaluation of Stencil Computation

We first present the performance results of stencil computation program–Jacobi2D on JYC as shown in table 8.2. In the experiments, the matrix size of Jacobi2D is set to be 8192×8192 and the program is decomposed into 32×32 objects, each holding a block of size 256×256. We ran the program both in the non-SMP mode and in the SMP mode. We set every SMP node to use 8 cores, i.e., the PEs per Node (ppn) is 7 with one core dedicated to the communication thread.

Comparing the performance between the non-SMP and the SMP (the third column vs.

#nodes	#cores	#SMP Nodes	non-SMP	SMP	speedup	SMP*	speedup*
1	16	2	81.13	82.50	-1.65%	80.80	0.41%
1	32	4	42.75	42.83	-0.19%	41.69	2.54%
2	64	8	22.07	22.25	-0.80%	21.35	3.38%
4	128	16	11.45	11.35	0.85%	11.24	1.84%
8	256	32	5.91	6.31	-6.40%	6.06	-2.51%
16	512	64	2.75	3.05	-9.99%	2.71	1.44%

Table 8.2: Jacobi2D Performance (ms/step) Comparison on JYC

the fourth column in table 8.2), we find that the SMP mode performs slightly slower on small core counts and moderately slower on large core counts (i.e., the 256-core and 512-core runs) despite the lower cost of intra-node communication (occupying roughly half of the total communication in this program). The slow down can be attributed to the following two reasons:

First, we have spared one core every SMP node for communication, therefore, we have lost about 10% computation power in the SMP mode. Although we do not lose any computation resources in terms of floating-point (FP) operations as every two cores share the same set of FP function units in the AMD Interlagos processor, the resources may not be fully utilized by just one worker thread (i.e., one CHARM++ PE).

Secondly, with a total of 1024 objects, every PE has the same amount of objects to perform computation in the non-SMP mode. However, in the SMP mode as we have 7 PEs per node, the load becomes slightly imbalanced because one or more PEs will have one more object than the others. As a result, the computation performed by those extra objects delayed the step completion. When scaling up, since each PE will have fewer objects, the extra object makes the load imbalance more pronouncing. For example, on the 512-core run, on every SMP node, two PEs have 3 objects while the remaining five PEs have 2 objects each. This factor is the major cause for the more performance degradation on runs with larger core counts.

As presented in chapter 5, we can mitigate such load imbalance via the conditional exploitation of the single-node parallelism. The last two columns of table 8.2 show the effectiveness of this approach that the SMP mode performs better than the non-SMP mode does except the 256-core run which, however, is about 4% improvement over the default

SMP run.

8.2 Evaluation of NAMD

We now present the results of NAMD performance on multiple parallel platforms, including JYC, JaguarPF, Intrepid and Titan. In particular, we focused on the performance of simulating the 100M-atom STMV molecule system because it is very close to the molecule system that NSF appointed to run on the sustainable petascale machine–BlueWater[1] as one of the machine acceptance benchmarks.

Table 8.3 shows the best achievable performance of DHFR, Apoa1 and 1M-atom STMV in MPI-based CHARM++ runtime with default simulation parameters on JYC. As for the smallest molecule system DHFR, the performance in SMP mode is better than that in non-SMP mode in all three scaling runs. However, regarding the larger systems Apoa1 and 1M-atom STMV, the SMP performs worse than the non-SMP does. We also notice that with the increase of cores, the performance gap between the non-SMP mode and the SMP one becomes decreasingly smaller. We think such performance behavior is a mixture of three factors.

#nodes	#cores	DHFR		Apoa1		1M-atom STMV	
		non-SMP	SMP	non-SMP	SMP	non-SMP	SMP
8	256	4.33	2.87	7.91	8.67	86.33	92.89
16	512	2.48	1.61	4.51	4.59	44.77	48.17
32	1024	1.45	1.29	2.56	2.6	23.33	24.86

Table 8.3: Performance (ms/step) of NAMD on JYC

First, when the average number of atoms per PE becomes smaller, there is less computation that could effectively hide the communication latency. Therefore, NAMD becomes more communication sensitive. We think the communication performance becomes better in the SMP mode under such circumstances thanks to the lower cost of intra-node communication and the optimizations both in the runtime and in NAMD itself that reduce the number of network messages. Secondly, the separation of the computation and communication into worker threads and the communication thread respectively in the SMP mode

[1]http://www.ncsa.illinois.edu/BlueWaters

improves the quality of load balancing in NAMD as the worker threads are less perturbed by the communication noise. Finally, we dedicate one core for the communication thread without doing any computation in the SMP mode. Even the floating-point computation resources remain the same in our experiment on JYC (every two cores of Interlagos share the same set of floating-point function units), a quarter of the resources are not fully utilized as they are only touched by one worker thread that shares the resource with the communication thread, while the remaining three quarters of the resources are touched by two worker threads simultaneously. Such loss of the computing power in the SMP mode negates the performance of the floating-point intensive scientific and engineering applications.

We evaluated the performance of the 100M-atom STMV both in the non-SMP mode and in the SMP mode on JaguarPF. Table 8.4 shows the performance results with three physics configurations as PME, cutoff with barrier and cutoff without barrier, each of which corresponds to a different scientific simulation. Each node of JaguarPF has 12 physical cores, therefore, we set ppn to be 11, i.e., each SMP node contains 11 worker threads and 1 communication thread.

#cores	PME		Cutoff w/ barrier		Cutoff w/o barrier	
	non-SMP	SMP	non-SMP	SMP	non-SMP	SMP
1680	1295.5	1344.0	1097.3	1118.5	*	*
6720	351.15	345.84	329.51	294.19	319.53	281.59
53760	60.34	54.25	68.67	44.21	43.49	36.84
107520	39.58	36.49	49.10	28.74	25.07	18.62
224076	45.52	26.28	38.25	16.84	14.58	9.00

Table 8.4: Performance (ms/step) of 100M-atom STMV Simulation on JaguarPF

Figure 8.1 plots the performance speedup based on the results obtained on the 1680-core run, demonstrating a better scaling in SMP mode.

Based on the performance data in table 8.4, performance in non-SMP mode is better than that in SMP mode on smaller number of cores (i.e., 1680 cores). As the machine utilization is relatively high on small-scale runs for this very large molecule system simulation, the loss of one core of each node to the dedicated communication thread without doing any computation does hurt the performance. However, after the number of cores exceeds a threshold (6720 cores in this case), the performance in SMP mode becomes better in all

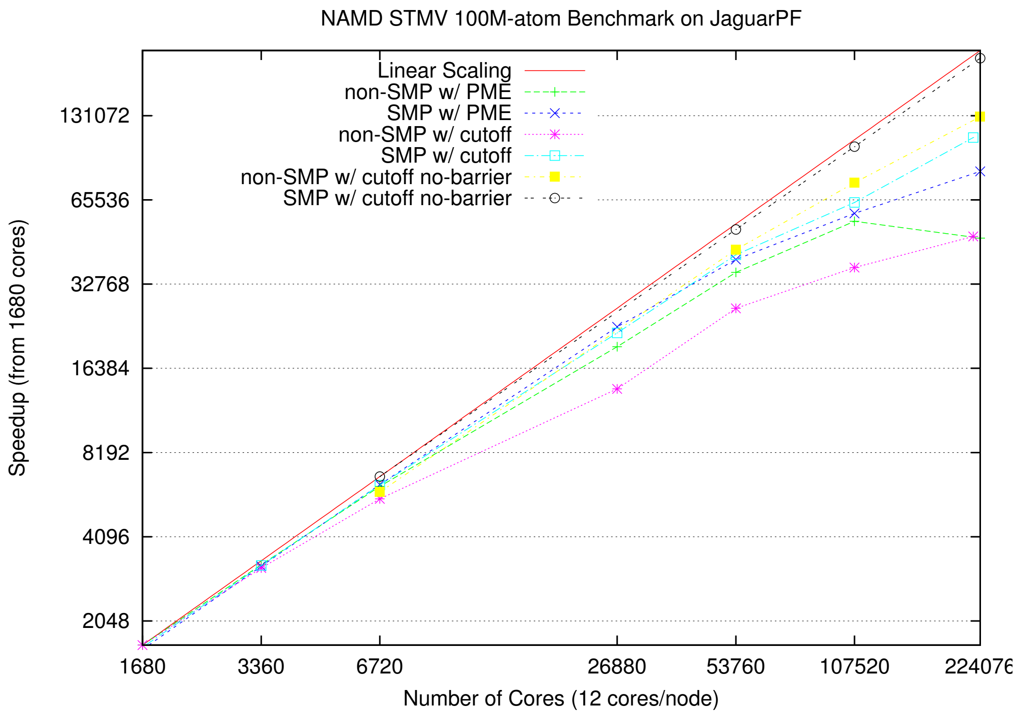

Figure 8.1: The Speedup of 100M-atom STMV Simulation on JaguarPF

three configurations. With further scaling in the number of cores, as illustrated in figure 8.1, NAMD achieves an increasing performance improvement of SMP mode over non-SMP mode. For example, with PME set, NAMD performs worse from 107,520 cores to 215,040 cores in non-SMP mode, while it still scales in SMP mode. For the full JaguarPF run of 224,076 cores, NAMD is almost twice as fast as in SMP mode than in non-SMP mode thanks to the various benefits of the multithreaded runtime such as the significant reduction in memory footprint via the shared memory address space, and the lower communication cost within a SMP node etc.

On Intrepid, the 100M-atom STMV simulation can just use one core out of four cores per physical node in the non-SMP mode because its memory footprint per OS process is so large that it barely fit in the 2GB of physical memory that is available on each Intrepid node. However, in SMP mode, as memory is shared across all cores, the simulation is able to use all cores on a physical node and scales almost perfectly to 65536 cores based on the 2048-core performance in all three configurations as shown in figure 8.2.

The Titan machine is upgraded from JaguarPF with the latest Cray Gemini intercon-

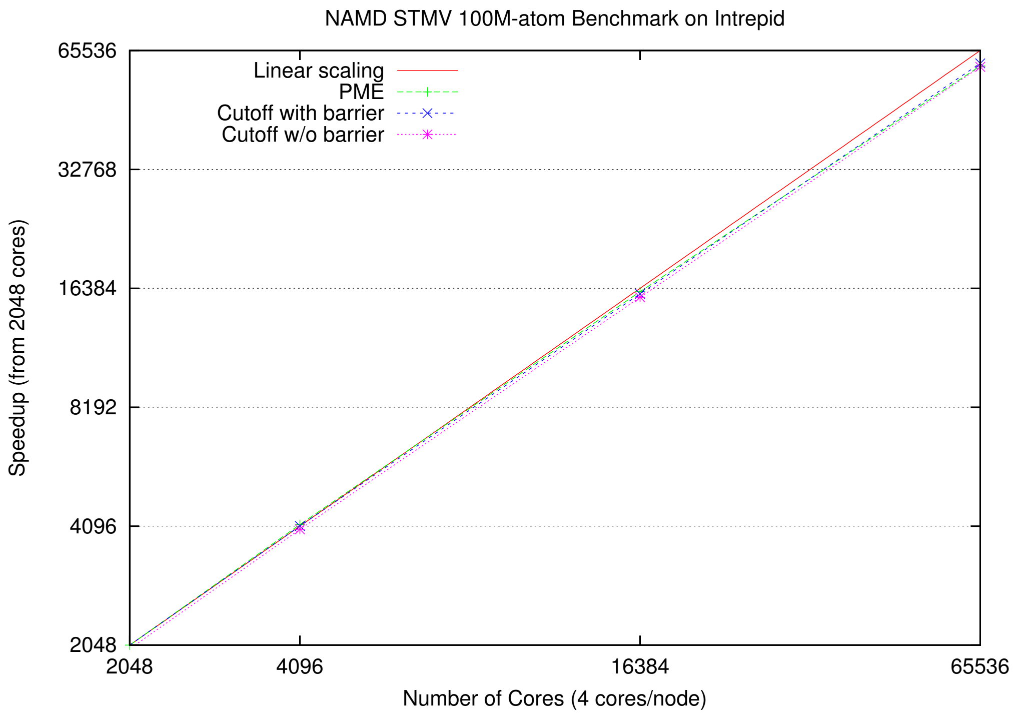

Figure 8.2: The Speedup of 100M-atom STMV Simulation on Intrepid

#cores	DHFR (ms/step)			1M-atom STMV (ms/step)		
	MPI	uGNI	Speedup	MPI	uGNI	Speedup
16	16.60	15.70	5.73%	1484.00	1483.00	0.07%
64	4.71	4.55	3.52%	395.00	393.50	0.38%
128	2.85	2.50	14.00%	200.00	198.00	1.01%
512	1.36	1.07	27.10%	52.00	51.60	0.78%
1024	1.11	0.80	38.75%	29.80	27.50	8.36%
2048	1.00	0.60	66.67%	15.20	14.80	2.70%
4096	-	-	-	10.5	8.32	26.20%
8192	-	-	-	8.76	6.05	44.79%

Table 8.5: NAMD Performance (PME Every 4 Steps) Under Different Communication Substrate

nect [50] and AMD Interlagos processors. The lower-level communication library uGNI on this interconnect has been utilized as the communication substrate in CHARM++ [49]. With two molecule benchmarks DHFR and 1M-atom STMV, table 8.5 compares the performances of NAMD in SMP mode between uGNI-based CHARM++ and MPI-based one. Based on the results in the table, we can see that NAMD always achieves better performance with uGNI-based CHARM++ runtime. We find that the performance gap generally becomes wider with the increase of cores. As NAMD becomes increasingly sensitive to the communication cost when it scales up, the overhead of MPI over uGNI turns to be more pronouncing, thus degrading the overall NAMD performance. In addition, we observe that, for a run on the same number of cores, the smaller molecule system–DHFR achieves more speedup from the change in the communication substrates than the larger molecule system–1M-atom STMV does. This is because the larger molecule system incurs more computation work which in turn more tolerates the communication latency.

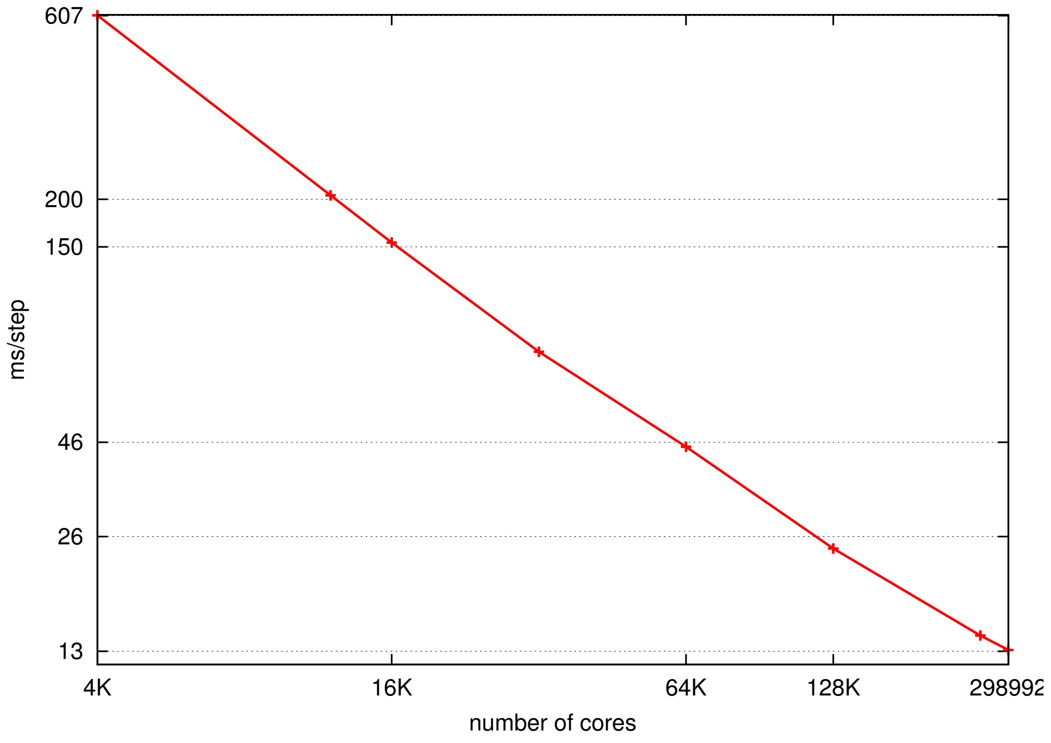

Figure 8.3: The Performance (ms/step) of 100M-atom STMV Simulation (PME Every 4 Steps) on Titan

Figure 8.3 shows the performance of the 100M-atom STMV on Titan in the SMP mode of uGNI-based CHARM++ runtime. It is clear that NAMD achieves a very good scalability up to the full Titan machine with 298992 cores. It is worth mentioning that the performance of the full-machine scale is a significant improvement over that of JaguarPF.

8.3 Miscellaneous Evaluation

- **Performance impact from the size of SMP node:** Noticing the increasing number of cores per physical node on the latest massively parallel machines, we evaluate the application performance with different SMP node sizes (i.e., different ppn values). Table 8.6 shows the performance results on JYC of Jacobi2D and NAMD with three molecule systems in two cases of SMP node size. The first column of the table indicates the number of physical nodes. Each physical node will have 4 SMP nodes in case of $ppn = 7$, and it will have 2 SMP nodes when $ppn = 15$.

#nodes	Jacobi2D		DHFR		Apoa1		1M-atom STMV	
	ppn=7	ppn=15	ppn=7	ppn=15	ppn=7	ppn=15	ppn=7	ppn=15
8	6.06	6.32	2.87	3.58	9.22	9.66	94.83	92.89
16	2.71	3.33	1.79	1.98	4.93	5.39	48.15	48.66
32	1.32	1.51	1.42	1.73	2.96	3.49	25.43	25.45

Table 8.6: Performance (ms/step) Comparison of Different SMP Node Size on JYC

According to the results in the table, the performance in case of $ppn = 7$ is always better than that in case of $ppn = 15$ except the 1M-atom STMV simulation on 8 physical nodes. This behavior could be attributed to two factors. First, as mentioned in section 6.1.2, if the SMP node size is larger, the only one dedicated communication thread per SMP node may more easily become overloaded thus decreasing the overall performance despite that the loss of computation resources is smaller. Secondly, due to the first-touch memory allocation policy, the memory buffers for network messages are always allocated in the memory local to the communication thread. Note that each node of JYC has two AMD Interlagos processors and each of them consists of two dies connected via HyperTransport interconnection links, forming a Non-Uniform Memory Architecture (NUMA). In the case $ppn = 15$, suppose the communication thread stays on the first die, then all PEs on the second die, in order to access their received network messages, will always have to access the remote memory attached to the first die. In contrast, in the case $ppn = 7$, PEs have no such remote memory accesses. Since it is much quicker to access local memory than it is to access memory attached to the other die in NUMA, the performance of $ppn = 15$ becomes worse. However, if the run is more dominated by computation, then the loss of computation resources due to the dedicated communication thread will affect the performance more as the case of 1M-atom STMV simulation where the performance of $ppn = 7$ is worse than that of $ppn = 15$. In general, suggested by the performance results in table 8.6, at least one SMP node should be launched per NUMA node due to the first-touch memory allocation policy on a physical node.

- **Fewer cache misses in the SMP mode:**As described in section 6.1.2, the separation of computation and communication work into different threads produces better cache performance. We observed this benefit from NAMD runs on JaguarPF in which we collected information regarding the number of misses of L1 cache and L2 cache per timestep of NAMD as shown in table 8.7. The corresponding percentage of cache performance improvement in terms of the amount of cache misses is plotted in figure 8.4.

Molecule System	#Nodes	L1 Data Cache Misses		L2 Data Cache Misses	
		non-SMP	SMP	non-SMP	SMP
Apoa1	32	1.26E+05	6.75E+04	1.14E+04	8.61E+03
100M-atom STMV	4480	8.23E+05	5.02E+05	3.89E+04	3.32E+04
100M-atom STMV	8960	4.54E+05	2.55E+05	2.21E+04	1.86E+04

Table 8.7: Comparison of L1 and L2 Data Cache Misses per Timestep of NAMD between SMP Mode and non-SMP Mode

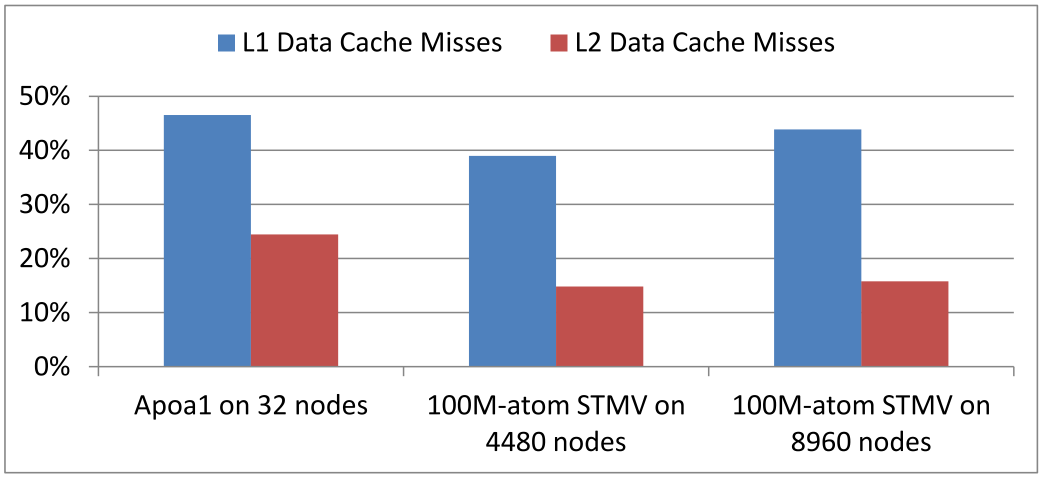

Figure 8.4: The Percentage of Cache Performance Improvement

Clearly, the number of cache misses is reduced in SMP mode in all cases as demonstrated by figure 8.4. The decrease in L2 cache misses is mainly due to the overall memory footprint reduction in SMP mode from sharing certain data structures enabled by the shared memory address space. The less memory footprint also contributes to the fewer L1 cache misses. Observing the figure 8.4, we find that the SMP mode saves more L1 cache misses than it does for L2 cache misses. This is because no network communication is performed on worker threads of the multithreaded runtime, and the L1 cache is thus prevented from being polluted by the underlying communication library as MPI in this case. Such separation also prevents the communication noise from perturbing the load information recorded for the load balancing in NAMD.

8.4 Summary

Based on the performance evaluation in this chapter, many factors determine that whether applications running in the multithreaded runtime mode achieve better performance than that in the traditional runtime mode in which one OS process is launched on every core.

As we dedicate a core to the communication thread on every SMP node in the multithreaded runtime, such loss in computing power is the major cause for the worse performance in the SMP mode. Additionally, the funneled communication via the communication thread may cause worse performance if there are many network messages.

However, we also witnessed the benefits of the multithreaded runtime that lead to better application performances. The lower latency of intra-node communication is one of them. The dedicated communication thread as separated from the worker threads results in better cache performance, more responsiveness in communication when it is not overloaded. In addition, it is very beneficial to have memory consumption reduced via the shared memory address space in the SMP mode as clearly demonstrated by the 100M-atom STMV simulation in NAMD.

In particular, the multithreaded runtime enables better scalability for applications than the traditional non-SMP one does on very large-scale runs. It is clearly exemplified by the better scalability that NAMD achieves. With the 100M-atom STMV simulation, NAMD scales up to 224,076 cores on JaguarPF and up to 298,992 cores on Titan respectively in the SMP mode.

9 Related Work

Since the mass production of multicore chips in the market, much work has been done to exploit the multicore system or optimize for it. A large portion of these work, such as presented in [90, 9, 91, 92, 10, 93], has been focused on user-level applications. This work, however, takes a different perspective as it tries to exploit the multicore system in the level of parallel language runtime, one layer down from the application in the software stack. The performance benefits gained in runtime will be automatically enjoyed by its associated applications.

Some application-level work particularly focused on the performance on a single multicore node such as optimizing lattice Boltzmann computation [90], fast multipole method [94]. We may share a similar engineering process of optimizing the runtime on a single node with such work. But because the language runtime is supposed to serve many different applications, the development of its general optimization techniques becomes more complex and is likely to require more effort.

For the effort on optimizing the multi-node performance of applications, using the hybrid programming method, such as MPI+OpenMP, is a popular approach illustrated in work [9, 91, 93, 95]. This approach requires application developers to modify source codes or restructure the whole program to achieve a better performance. In comparison, since our approach applies the hybrid programming in the language runtime, application developers do not have to write shared-memory specific codes, or restructure the whole program to do optimizations that a runtime can automatically perform such as intra-node communication optimizations, sharing data structures to save memory etc. Note that a good language abstraction, like the one provided by CHARM++, may be required for this to happen automatically in the runtime.

As for MPI runtimes, work has been devoted to optimizing message latencies within a node to utilize the shared physical memory of a multicore node. MPI tasks are OS processes according to the MPI standard. Therefore, the message passing within a node replies on inter-process memory transfer techniques. Some of these, like nemesis used in mpich [2], the hybrid approach developed in mvapich [96], rely on posix-shared memory between communicating tasks by first copying the sending buffer to the shared memory, and then a second copy is performed from the shared memory to the receiving buffer. OS-kernel assisted methods reduce the number of message copies down to one, as described in [3, 4, 96, 97], by directing copy the buffer from the source to the destination with the assistance of a OS kernel module performing physical-address memory copy. This approach incurs additional overhead in OS kernel involvement and multiple intermediate control steps. In contrast, in a multithreaded runtime described in this work, the message passing within a node is naturally achieved by one single copy as "tasks" are now sharing a single virtual memory address space. In the case of CHARM++, because its nature of message driven execution, the copy of the whole message is not required at all as the destination only needs to learn the pointer to the message. In addition, because of the single memory address space, sharing some data among "tasks" may be easily exploited.

To eliminate all extraneous memory copies imposed by UNIX-based shared memory strategies, some MPI runtime systems use SMARTMAP [98, 99, 5] which implements fixed offset virtual memory address and allows OS processes on a multicore chip to directly access peer's memory without kernel assistance. Although this approach achieves the same capability as delivering a message via its pointer with the multithreaded runtime, its utilization is limited to certain platforms because SMARTMAP requires architecture-specific support to be implemented. In contrast, the thread-based approach is more portable, say using pthread which is standardized and available almost on all UNIX-based platforms. We are also aware that on the platform where SMARTMAP is supported, by taking advantage of this technique, the traditional runtime execution mode as one OS process per core can also realize the benefits of a multithreaded runtime mentioned in chapter 3. It is worth applying this technique into the CHARM++ runtime, and then comparing its performance with that of the multithreaded mode. We also think the analysis conducted for the design space exploration of the multithreaded runtime system in section 6.1.2 can still be useful in this case.

Using a thread to represent a MPI task instead of a OS process was proposed as long as a decade ago by H.Tang *et al.* [100], and such idea has been re-visited by some recent work as shown in [101, 102, 103]. They agree on that communication within a node could be better performed if the source and destination are in the same memory address space achieved by using threads. This work shares this same idea, and implements it in the asynchronous message-driven CHARM++ runtime. Furthermore, we have examined the inter-node communication performance issues in depth with the multithreaded runtime. In contrast, the aforementioned work is lack of the discussion of issues in inter-node communication. H.Tang *et al.* [100] foresees there are going to be potential problems in the communication in their own multithreaded MPI runtime, but did not go further at that time as they did not encounter them in their experiments. Work in [101, 102, 103], on the other hand, just rely on the standard MPI implementation to provide inter-node communication, which has performance issues as we point out when using MPI as the communication substrate for a runtime.

There is also work on improving MPI performance in case that MPI functions are used in a multithreaded environment by P.Balaji *et al.* [104]. The purpose of such work is not to make the MPI runtime multithreaded, thus differing from the motivation of this work. But the work will be complementary to our efforts on improving the communication performance of the multithreaded runtime if MPI is used as the communication substrate. Furthermore, their findings such as reducing the granularity of critical sections is key to achieve high performance align with ours when dealing with performance issues of the multithreaded runtime on a single node.

As there are increasingly number of cores per node, the effect of NUMA on application's performance is more pronouncing. Work [105, 106] analyzed its impact on MPI and its applications, and developed techniques to solve performance issues. Although this work has not touched NUMA issues, we are very well aware of their effects on the multithreaded runtime described here. For example, since the communication thread allocates the memory for an incoming network message, and then only delivers the pointer of this message to the destination worker thread. If the worker thread is not on the same NUMA node with the communication thread, it could be potentially better to copy the message to the destination worker thread to save memory accesses of much longer latency as a result of NUMA. Therefore, we generally take one NUMA node as a SMP node in the multithreaded runtime in case the NUMA factor is significant. Furthermore, since only one communication thread is generally insufficient to serve the communication of tens of worker threads, restricting a SMP node to be one NUMA node helps avoid such overloading problem.

There is also optimization work on the scheduling inside operating systems [107, 33, 108] for multicore system. Such work is similar to the new load balancing strategy described in section 7.5 in that the contention of shared resources among cores on a multicore chip needs to be taken into account. However, techniques used by these work are not appropriate for load balancing in the context of the multithreaded runtime described in this work, especially in the case of CHARM++. First, the granularity of migration unit is very different. The scheduling in OS deals with processes or threads, whereas the load balancer in the runtime deals with more fine-grained CHARM++ parallel objects. Secondly, the factors considered in OS scheduling are not enough for load balancer to achieve good results primarily due to different purposes between a OS scheduler and a load balancer in parallel language runtime. The load balancing strategy presented in section 7.6 that considers the asymmetric PE introduced by the multithreaded runtime design bears similar ideas with the *SyMMer* [109] process mapping library that also considers the asymmetry in the effective capability of the different cores resulted from the interaction between multicore hardware components and the system software. However, *SyMMer* focuses more on the effect of such asymmetry on the communication performance while this work considers it more for balancing the computation load.

Plenty of studies have been performed on considering the communication cost to improve the quality of load balancing. The network topology-aware load balancing strategy [73] proposes *hop-bytes* as a communication performance metric, defined as the total number of bytes exchanged between processors weighted by the distance between them, to be considered in the heuristic of the load balancing strategy. Particularly targeted at 3D mesh and torus network architectures, work [110] also used *hop-bytes* to evaluate the load balancing algorithms in the molecular dynamics application. Those techniques have been demonstrated very useful to improve the application performance. In comparison, the load balancing work in this thesis considers the communication latency difference caused by the multithreaded runtime as the cost of a message within a SMP node (i.e., represented by an OS process) is lower than that a message across SMP modes. Therefore, combining all these work as considering all different levels of communication hierarchy could lead to more effective load balancing strategies.

Work-stealing [111, 112, 71] is another well-known approach to distributing computational tasks among a set of processors. The key idea of this technique is that if a processor becomes idle without any computation work, it will then "steal" computation work from other processors. This thesis also utilizes this idea to exploit single-node parallelism as described in chapter 5. The work-stealing is also proposed in [113] to improve the performance of parallel loops by ensuring multicore machines sharing the same cache work on the data that are close in memory which reduces the total number of cache misses. The work on load balancing strategies in this thesis also considers the shared cache of multicore chips, but mainly targeting at reducing its contention in a broader context of shared physical resource of multicore platforms.

10 Conclusion and Future Work

Multicore chips have become the standard building blocks for large scale massively parallel machines, however, it is difficult make an effective use of those chips as the popular hybrid programming approach requires application developers deal with various shared-memory program problems, and the approach does not necessarily improve the performance. Instead of optimizing the application codes, this thesis takes a different perspective in utilizing multicore chips as focusing on optimizing the parallel language runtime system by parallelizing the runtime with threads to utilize the multicore chips. As a result, the parallel application could almost automatically enjoy the benefits of multicore chips and performance improvement with minimal burden on developers to change source codes. Motivated by the initial inefficient implementation of this approach in CHARM++ runtime and based on such system, this thesis investigates various design and performance issues in the message-driven multithreaded runtime and develops corresponding optimization techniques. In addition, this thesis uses production-level scientific applications to evaluate the multithreaded runtime and demonstrate its effectiveness, making the work of more practical usage.

In chapter 3, we examine the benefits of a multithreaded language runtime system including lower latency of intra-node communication, reduction of overall memory footprint and parallel job startup time, more intuitive way of sharing certain data structures and transparency to application developers thanks to the unchanged programming model.

Those benefits could not be reflected unless we achieve a high-performance implementation of the multithreaded runtime. In the context of the initial implementation of mulithreaded CHARM++ runtime, we first investigate issues that affect the single-node performance, particularly the intra-node communication in chapter 4. We identify multiple performance critical factors such as the contention arising from making runtime multi-thread safe, the cache false sharing and CPU affinity etc. We also develop corresponding optimization techniques such as effectively using atomic operations, using thread private variables etc. to improve the single-node performance. After those optimizations, we improve a communication-intensive benchmark by an average of about 14 times than that running with the initial multithreaded CHARM++ implementation, and about 5 times on average than that running with the default non-multithreaded CHARM++. We also obtain better performances in production-level applications–NAMD and ChaNGa with the multithreaded runtime. In the future, we will study how to take NUMA effects and the CPU topology of a multicore into account to improve the intra-node communication further.

Furthermore, we address the problem of how to integrating fine-grained single-node parallelism into the multithreaded runtime in chapter 5. Motivated by the overhead of directly using OpenMP, we develop a low-overhead library *CkLoop* that resembles OpenMP, but leveraging existing threads in the runtime. We demonstrate the benefits of this library in a Jacobi program and NAMD. This part of work suggests the necessity of a common software stack that various parallel language runtime systems can use to coordinate with each other. In the future, we will use language directives to represent the program transformation needed to use this library, and then employ source-to-source compilers such as Rose [114, 115] to automatically generate the code transformation.

As the parallel language runtime targeted in this thesis is supposed to run on distributed-memory parallel machines, this thesis also focuses on examining performance issues in the component of the multithreaded runtime that is responsible for network communication in chapter 6. We first explore the design space of assigning the computation and communication work to different threads, and conclude it with a preferred design that has a dedicated communication thread per SMP node after analyzing the pros and cons of different design options. Furthermore, we address issues of using MPI as the communication substrate to better serve the message-driven execution model. With the extension of performance tracing and visualization framework, we study the performance problems with the dedicated communication thread and propose optimization techniques as the node-aware communication and "restrained-effort" strategy in alternating different work to alleviate those problems. In the end, to make most out of the capability of the multithreaded runtime, we describe methods that could be applied into the application codes for better performance such as exploiting the dedicated communication thread to improve the responsiveness of the asynchronous collective communication. In the future, we will explore latest low-level communication libraries such as PAMI [116] that support multiple communication endpoints in a single memory address space for the multithreaded language runtime. Additionally, it is worth investigating an automatic scheme to set the best parameters, such as the number of PEs per SMP node and the number of SMP nodes per physical node etc., of the multithreaded runtime

Motivated by the lack of considering differences between the multithreaded runtime and the traditional non-multithraded runtime as well as architectural characteristics of multicore chips in load balancing strategies, in chapter 7, we develop and demonstrate new and more-effective load balancing strategies based on the built-in load balancing framework in CHARM++ with extensions in recording values of hardware performance counters and querying APIs for the CPU topology. In the future, we will explore in constructing a synthetic metric that could represent the work load more accurately, and apply the insights gained from making load balancing more effective on multicore platforms into the task scheduling of the multithreaded runtime.

We finally evaluate the performance of scientific applications running with the multithreaded runtime on a couple of massively parallel machines in chapter 8. In case of the 100M-atom simulation using NAMD, we achieve a higher utilization rate by three times with the multithreaded runtime than that with the default non-multithreaded one on BlueGene/P. NAMD also achieves much better scalability up to 224,076 cores on JaguarPF and up to 298,992 cores on Titan respectively with this multithreaded runtime.

In short, in spite of the great amount of engineering work spent on tuning the performance of the multithreaded runtime, it is worth the efforts because the language runtime is such a important software layer for applications that affect their performances significantly. The performance issues identified in this thesis and corresponding optimization techniques will shed light on the implementation of other new high-performance parallel language runtimes for multicore-based massively parallel machines.

References

[1] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard - Version 2.2*, 2009.

[2] Darius Buntinas, Guillaume Mercier, and William Gropp. Design and evaluation of nemesis, a scalable, low-latency, message-passing communication subsystem. In *CCGRID '06: Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid*, pages 521–530, Washington, DC, USA, 2006.

[3] Hyun-Wook Jin and Dhabaleswar K. Panda. Limic: Support for high-performance mpi intra-node communication on linux cluster. In *Proceedings of the 2005 International Conference on Parallel Processing*, ICPP '05, pages 184–191, 2005.

[4] Hyun-Wook Jin, Sayantan Sur, Lei Chai, and Dhabaleswar K. Panda. Lightweight kernel-level primitives for high-performance mpi intra-node communication over multi-core systems. In *Proceedings of the 2007 IEEE International Conference on Cluster Computing*, CLUSTER '07, pages 446–451, Washington, DC, USA, 2007. IEEE Computer Society.

[5] Ron Brightwell. Exploiting direct access shared memory for MPI on multi-core processors. *Int. J. High Perform. Comput. Appl.*, 24(1):69–77, 2010.

[6] Leonardo Dagum and Ramesh Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Computational Science & Engineering*, 5(1), January-March 1998.

[7] David R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[8] James Reinders. *Intel threading building blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007.

[9] Lorna Smith and Mark Bull. Development of mixed mode MPI / OpenMP applications. *Scientific Programming*, 9(2-3/2001):83–98, 2001.

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

[11] NERSC. Using openmp effectively on hopper. `http://www.nersc.gov/users/computational-systems/hopper/performance-and-optimization/using-openmp-effectively-on-hopper/`.

[12] Thomas Sterling Tarek El-Ghazawi, William Carlson and Katherine Yelick. *UPC: Distributed Shared Memory Programming*. John Wiley & Sons, Inc., Hoboken, NJ, USA, 2005.

[13] G.R. Gao, T. Sterling, R. Stevens, M. Hereld, and W. Zhu. ParalleX: A study of a new parallel computation model. In *IEEE International Parallel and Distributed Processing Symposium, 2007*, pages 1 –6, march 2007.

[14] Aparna Chandramowlishwaran, Kathleen Knobe, and Richard Vuduc. Performance evaluation of concurrent collections on high-performance multicore computing systems. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2010.

[15] Laxmikant V. Kale and Gengbin Zheng. Charm++ and AMPI: Adaptive Runtime Strategies via Migratable Objects. In M. Parashar, editor, *Advanced Computational Infrastructures for Parallel and Distributed Applications*, pages 265–282. Wiley-Interscience, 2009.

[16] Stéphane Zuckerman, Joshua Suetterlein, Rob Knauerhase, and Guang R. Gao. Using a ”codelet” program execution model for exascale machines: position paper. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, EXADAPT '11, pages 64–69, 2011.

[17] James C. Phillips, Rosemary Braun, Wei Wang, James Gumbart, Emad Tajkhorshid, Elizabeth Villa, Christophe Chipot, Robert D. Skeel, Laxmikant Kalé, and Klaus Schulten. Scalable molecular dynamics with NAMD. *Journal of Computational Chemistry*, 26(16):1781–1802, 2005.

[18] L.V. Kale and S. Krishnan. Charm++: A portable concurrent object oriented system based on C++. In *Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications*, September 1993.

[19] Avneesh Pant, Hassan Jafri, and Volodymyr Kindratenko. Phoenix: A runtime environment for high performance computing on chip multiprocessors. *Parallel, Distributed, and Network-Based Processing, Euromicro Conference on*, 0:119–126, 2009.

[20] Dan Bonachea and Jason Duell. Problems with using mpi 1.1 and 2.0 as compilation targets for parallel language implementations. *Int. J. High Perform. Comput. Netw.*, 1:91–99, August 2004.

[21] Marc Snir. A note on n-body computations with cutoffs. *Theory of Computing Systems*, 37:295–318, 2004. 10.1007/s00224-003-1071-0.

[22] Kevin J. Bowers, Ron O. Dror, and David E. Shaw. Zonal methods for the parallel execution of range-limited n-body simulations. *J. Comput. Phys.*, 221(1):303–329, January 2007.

[23] Pritish Jetley, Filippo Gioachin, Celso Mendes, Laxmikant V. Kale, and Thomas R. Quinn. Massively parallel cosmological simulations with ChaNGa. In *IPDPS*, 2008.

[24] Chao Mei and Laxmikant V. Kalé et al. Enabling and scaling biomolecular simulations of 100 million atoms on petascale machines with a multicore-optimized message-driven runtime. In *Proceedings of the 2011 ACM/IEEE conference on Supercomputing*.

[25] Chao Mei, Gengbin Zheng, Filippo Gioachin, and Laxmikant V. Kalé. Optimizing a Parallel Runtime System for Multicore Clusters: A Case Study. In *TeraGrid'10*, number 10-13, Pittsburgh, PA, USA, August 2010.

[26] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: a scalable memory allocator for multithreaded applications. *SIGPLAN Not.*, 35(11):117–128, November 2000.

[27] Maged M. Michael. Scalable lock-free dynamic memory allocation. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, PLDI '04, pages 35–46, New York, NY, USA, 2004. ACM.

[28] Sangmin Seo, Junghyun Kim, and Jaejin Lee. Sfmalloc: A lock-free and mostly synchronization-free dynamic memory allocator for manycores. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 253–263, Washington, DC, USA, 2011. IEEE Computer Society.

[29] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996.

[30] Henry Massalin and Calton Pu. A lock-free multiprocessor os kernel, 1991.

[31] Maged M. Michael and Michael L. Scott. Fast and practical non-blocking and blocking concurrent queue algorithms. In *Proc. 15th ACM Symp. on Principles of Distributed Computing*, pages 267–275, 1996.

[32] John L. Hennessy and David A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[33] Tong Li, Dan Baumberger, David A. Koufaty, and Scott Hahn. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–11, New York, NY, USA, 2007. ACM.

[34] Chi Zhang, Xin Yuan, and Ashok Srinivasan. Processor affinity and mpi performance on smp-cmp clusters. In *IPDPS Workshops*, pages 1–8, 2010.

[35] Guillaume Mercier and Emmanuel Jeannot. Improving mpi applications performance on multicore clusters with rank reordering. In *EuroMPI*, pages 39–49, 2011.

[36] T. Darden, D. York, and L. Pedersen. Particle mesh ewald: An $N \log(N)$ method for ewald sums in large systems. *Journal of Chemical Physics*, 98, 1993.

[37] U. Essmann, L. Perera, M. L. Berkowitz, T. Darden, H. Lee, and L. Pedersen. A smooth particle mesh ewald method. *Journal of Chemical Physics*, 103, 1995.

[38] Heidi Pan. *Cooperative hierarchical resource management for efficient composition of parallel software*. PhD thesis, Massachusetts Institute of Technology, 2010. http://dspace.mit.edu/handle/1721.1/60172.

[39] Parallel Programming Laboratory. *The Charm++ Programming Language Manual*. Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL. http://charm.cs.illinois.edu/manuals/html/charm++/manual.html.

[40] L.V. Kalé and Amitabh Sinha. Projections : A scalable performance tool. In *Parallel Systems Fair, International Parallel Processing Sympos ium*, pages 108–114, April 1993.

[41] Sameer Kumar, Gabor Dozsa, Gheorghe Almasi, Philip Heidelberger, Dong Chen, Mark E. Giampapa, Blockso Michael, Ahmad Faraj, Jeff Parker, Joseph Ratterman, Brian Smith, and Charles J. Archer. The deep computing messaging framework: generalized scalable message passing on the blue gene/p supercomputer. In *Proceedings of the 22nd annual international conference on Supercomputing*, ICS '08, pages 94–103, New York, NY, USA, 2008. ACM.

[42] Howard Pritchard, Igor Gorodetsky, and Darius Buntinas. A ugni-based mpich2 nemesis network module for the cray xe. 6960:110–119, 2011.

[43] Dhabaleswar K. Panda and Pavan Balaji. Designing high-end computing systems with infiniband and10-gigabit ethernet iwarp. In *Proceedings of the 2007 IEEE International Conference on Cluster Computing*, CLUSTER '07, Washington, DC, USA, 2007. IEEE Computer Society.

[44] Gautam Shah, Jarek Nieplocha, Jamshed H. Mirza, Chulho Kim, Robert J. Harrison, Rama Govindaraju, Kevin J. Gildea, Paul DiNicola, and Carl A. Bender. Performance and experience with lapi - a new high-performance communication library for the ibm rs/6000 sp. In *IPPS/SPDP*, pages 260–266, 1998.

[45] Rajeev Thakur. Issues in developing a thread-safe mpi implementation. In *In Recent Advances in Parallel Virtual Machine and Message Passing Interface, 13th European PVM/MPI Users Group Meeting*, pages 12–21. Springer, 2006.

[46] Scott Atchley, David Dillow, Galen Shipman, Patrick Geoffray, Jeffrey M. Squyres, George Bosilca, and Ronald Minnich. The common communication interface (CCI). In *19th Annual IEEE Symposium on High-Performance Interconnects*, August 2011.

[47] Laxmikant V. Kale. Programming Models at Exascale: Adaptive Runtime Systems, Incomplete Simple Languages, and Interoperability. *The International Journal of High Performance Computing Applications*, 23(4):344–346, October 2009.

[48] Barbara Chapman. An evolutionary exascale programming model deserves revolutionary support, May 2012. http://www2.cs.uh.edu/ hpctools/hips12-presentation-chapman.pdf.

[49] Yanhua Sun, Gengbin Zheng, L. V. Kale, Terry R. Jones, and Ryan Olson. A uGNI-based Asynchronous Message-driven Runtime System
for Cray Supercomputers with Gemini Interconnect. In *Proceedings of 26th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Shanghai, China, May 2012.

[50] R. Alverson, D. Roweth, and L Kaplan. The Gemini System Interconnect. In *2010 IEEE 18th Annual Symposium on High Performance Interconnects (HOTI)*, 2011.

[51] Ron Brightwell and Keith Underwood. Evaluation of an eager protocol optimization for mpi. In *In Proceedings of EuroPVM/MPI*, pages 327–334, 2003.

[52] Tim S. Woodall, Galen M. Shipman, George Bosilca, Richard L. Graham, and Arthur B. Maccabe. High performance rdma protocols in hpc. In *Proceedings of the 13th European PVM/MPI User's Group conference on Recent advances in parallel virtual machine and message passing interface*, EuroPVM/MPI'06, pages 76–85, 2006.

[53] Cray Inc. *Using the GNI and DMAPP APIs*, 2010. `http://docs.cray.com/books/S-2446-3103/S-2446-3103.pdf`.

[54] Sayantan Sur, Hyun-Wook Jin, Lei Chai, and Dhabaleswar K. Panda. Rdma read based rendezvous protocol for mpi over infiniband: design alternatives and benefits. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '06, pages 32–39, 2006.

[55] S. Seelam, L. Fong, A. Tantawi, J. Lewars, J. Divirgilio, and K. Gildea. Extreme scale computing: Modeling the impact of system noise in multicore clustered systems. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1 –12, april 2010.

[56] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. Characterizing the influence of system noise on large-scale applications by simulation. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[57] Alessandro Morari, Roberto Gioiosa, Robert Wisniewski, Francisco J. Cazorla, and Mateo Valero. A quantitative analysis of os noise. In *IEEE Parallel and Distributed Processing Symposium, 2011*, 2011.

[58] Hartmut Kaiser, Maciek Brodowicz, and Thomas Sterling. Parallex an advanced parallel execution model for scaling-impaired applications. In *ICPPW '09: Proceedings of the 2009 International Conference on Parallel Processing Workshops*, pages 394–401, Washington, DC, USA, 2009. IEEE Computer Society.

[59] Christopher Lauderdale and Rishi Khan. Towards a codelet-based runtime for exascale computing: position paper. In *Proceedings of the 2nd International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, EXADAPT '12, pages 21–26, New York, NY, USA, 2012. ACM.

[60] L.V. Kalé and Amitabh Sinha. Projections: A preliminary performance tool for charm. In *Parallel Systems Fair, International Parallel Processing Symposium*, pages 108–114, Newport Beach, CA, April 1993.

[61] Laxmikant V. Kale, Gengbin Zheng, Chee Wai Lee, and Sameer Kumar. Scaling applications to massively parallel machines using projections performance analysis tool. In *Future Generation Computer Systems Special Issue on: Large-Scale System Performance Modeling and Analysis*, volume 22, pages 347–358, February 2006.

[62] Chee Wai Lee. *Techniques in Scalable and Effective Parallel Performance Analysis*. PhD thesis, Department of Computer Science, University of Illinois, Urbana-Champaign, December 2009.

[63] Isaac Dooley, Chao Mei, and Laxmikant V. Kale. Noiseminer: An algorithm for scalable automatic computational noise and software interference detection. In *Proceedings of HIPS Workshop at IEEE International Parallel and Distributed Processing Symposium 2008*, 2008.

[64] J. Liu and D.K. Panda. Implementing efficient and scalable flow control schemes in mpi over infiniband. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, April 2004.

[65] John E. Stone, James C. Phillips, Peter L. Freddolino, David J. Hardy, Leonardo G. Trabuco, and Klaus Schulten. Accelerating molecular modeling applications with graphics processors. *Journal of Computational Chemistry*, 28:2618–2640, 2007.

[66] James C. Phillips, John E. Stone, and Klaus Schulten. Adapting a message-driven parallel application to gpu-accelerated clusters. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 8:1–8:9, Piscataway, NJ, USA, 2008. IEEE Press.

[67] Christopher I. Rodrigues, David J. Hardy, John E. Stone, Klaus Schulten, and Wen-Mei W. Hwu. Gpu acceleration of cutoff pair potentials for molecular modeling applications. In *Proceedings of the 5th conference on Computing frontiers*, CF '08, pages 273–282, New York, NY, USA, 2008. ACM.

[68] NVIDIA. *CUDA 2.0 Programming Guide*. NVIDIA Corporation, Santa Clara, CA, USA, June 2008.

[69] Gregory A. Koenig and Laxmikant V. Kale. Optimizing distributed application performance using dynamic grid topology-aware load balancing. In *21st IEEE International Parallel and Distributed Processing Symposium*, March 2007.

[70] Eduardo R. Rodrigues, Philippe O. A. Navaux, Jairo Panetta, Alvaro Fazenda, Celso L. Mendes, and Laxmikant V. Kale. A Comparative Analysis of Load Balancing Algorithms Applied to a Weather Forecast Model. In *Proceedings of 22nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, Itaipava, Brazil, 2010.

[71] Laxmikant Kale Jonathan Lifflander, Sriram Krishnamoorthy. Work stealing and persistence-based load balancers for iterative overdecomposed applications. In *The 21st International ACM Symposium on High-Performance Parallel and Distributed Computing*, 2012.

[72] L. V. Kale, Milind Bhandarkar, and Robert Brunner. Run-time Support for Adaptive Load Balancing. In J. Rolim, editor, *Lecture Notes in Computer Science, Proceedings of 4th Workshop on Runtime Systems for Parallel Programming (RTSPP) Cancun - Mexico*, volume 1800, pages 1152–1159, March 2000.

[73] Tarun Agarwal, Amit Sharma, and Laxmikant V. Kalé. Topology-aware task mapping for reducing communication contention on large parallel machines. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2006*, April 2006.

[74] Gengbin Zheng, Esteban Meneses, Abhinav Bhatele, and Laxmikant V. Kale. Hierarchical Load Balancing for Charm++ Applications on Large Supercomputers. In *Proceedings of the Third International Workshop on Parallel Programming Models and Systems Software for High-End Computing (P2S2)*, San Diego, California, USA, September 2010.

[75] R. Hood, H. Jin, P. Mehrotra, J. Chang, J. Djomehri, S. Gavali, D. Jespersen, K. Taylor, and R. Biswas. Performance impact of resource contention in multicore systems. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, april 2010.

[76] Gengbin Zheng, Lixia Shi, and Laxmikant V. Kalé. FTC-Charm++: An In-Memory Checkpoint-Based Fault Tolerant Runtime for Charm++ and MPI. In *2004 IEEE International Conference on Cluster Computing*, pages 93–103, San Diego, CA, September 2004.

[77] Gengbin Zheng, Xiang Ni, and L. V. Kale. A Scalable Double In-memory Checkpoint and Restart Scheme towards Exascale. In *Proceedings of the 2nd Workshop on Fault-Tolerance for HPC at Extreme Scale (FTXS)*, Boston, USA, June 2012.

[78] Chao Mei. A preliminary investigation of emulating applications that use petabytes of memory on petascale machines. Master's thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 2007. http://charm.cs.uiuc.edu/papers/ChaoMeiMSThesis07.shtml.

[79] Osman Sarood, Abhishek Gupta, and Laxmikant V. Kale. Temperature aware load balancing for parallel applications: Preliminary work. In *The Seventh Workshop on High-Performance, Power-Aware Computing (HPPAC'11)*, Anchorage, Alaska, USA, 5 2011.

[80] Osman Sarood and Laxmikant V. Kalé. A 'cool' load balancer for parallel applications. In *Proceedings of the 2011 ACM/IEEE conference on Supercomputing*, Seattle, WA, November 2011.

[81] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst. hwloc: A generic framework for managing hardware affinities in hpc applications. In *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, pages 180 –186, feb. 2010.

[82] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *Int. J. High Perform. Comput. Appl.*, 14(3):189–204, 2000.

[83] Abhinav Bhatelé, Eric Bohm, and Laxmikant V. Kalé. A Case Study of Communication Optimizations on 3D Mesh Interconnects. In *Euro-Par 2009, LNCS 5704*, pages 1015–1028, 2009.

[84] Abhinav Bhatele. *Automating Topology Aware Mapping for Supercomputers*. PhD thesis, Dept. of Computer Science, University of Illinois, August 2010. `http://hdl.handle.net/2142/16578`.

[85] George Karypis and Vipin Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. In *Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*, page 35, 1996.

[86] Cdric Chevalier, Franois Pellegrini, Inria Futurs, and Universit Bordeaux I. Improvement of the efficiency of genetic algorithms for scalable parallel graph partitioning in a multi-level framework. In *In Proceedings of Euro-Par 2006, LNCS*, pages 243–252, 2006.

[87] Gengbin Zheng, Abhinav Bhatele, Esteban Meneses, and Laxmikant V. Kale. Periodic Hierarchical Load Balancing for Large Supercomputers. *IJHPCA*, March 2011.

[88] Esteban Meneses, Greg Bronevetsky, and Laxmikant V. Kale. Dynamic load balance for optimized message logging in fault tolerant hpc applications. In *IEEE International Conference on Cluster Computing (Cluster) 2011*, September 2011.

[89] Gengbin Zheng. *Achieving high performance on extremely large parallel machines: performance prediction and load balancing*. PhD thesis, Department of Computer Science, UIUC, 2005.

[90] Samuel Williams, Jonathan Carter, Leonid Oliker, John Shalf, and Katherine Yelick. Optimization of a lattice boltzmann computation on state-of-the-art multicore platforms. *Journal of Parallel and Distributed Computing*, 69(9):762 – 777, 2009.

[91] M.D. Jones, R. Yao, and C.P. Bhole. Hybrid mpi-openmp programming for parallel osem pet reconstruction. *Nuclear Science, IEEE Transactions on*, 53(5):2752–2758, oct. 2006.

[92] Kevin J. Bowers, Edmond Chow, Huafeng Xu, Ron O. Dror, Michael P. Eastwood, Brent A. Gregersen, John L. Klepeis, Istvan Kolossvary, Mark A. Moraes, Federico D. Sacerdoti, John K. Salmon, Yibing Shan, and David E. Shaw. Scalable algorithms for molecular dynamics simulations on commodity clusters. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, New York, NY, USA, 2006. ACM Press.

[93] Guoping Tang, Eduardo F. D'Azevedo, Fan Zhang, Jack C. Parker, David B. Watson, and Philip M. Jardine. Application of a hybrid mpi/openmp approach for parallel groundwater model calibration using multi-core computers. *Comput. Geosci.*, 36:1451–1460, November 2010.

[94] Aparna Chandramowlishwarany, Kamesh Madduri, and Richard Vuduc. Diagnosis, tuning, and redesign for multicore performance: A case study of the fast multipole method. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–12, Washington, DC, USA, 2010. IEEE Computer Society.

[95] Kevin J. Bowers, Edmond Chow, Huafeng Xu, Ron O. Dror, Michael P. Eastwood, Brent A. Gregersen, John L. Klepeis, Istvan Kolossvary, Mark A. Moraes, Federico D. Sacerdoti, John K. Salmon, Yibing Shan, and David E. Shaw. Molecular dynamics—scalable algorithms for molecular dynamics simulations on commodity clusters. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 84, New York, NY, USA, 2006. ACM Press.

[96] Lei Chai, Ping Lai, Hyun-Wook Jin, and Dhabaleswar K. Panda. Designing an efficient kernel-level and user-level hybrid approach for MPI intra-node communication on multi-core systems. In *ICPP '08: Proceedings of the 2008 37th International Conference on Parallel Processing*, pages 222–229, Washington, DC, USA, 2008.

[97] Darius Buntinas, Brice Goglin, David Goodell, Guillaume Mercier, and Stéphanie Moreaud. Cache-efficient, intranode, large-message mpi communication with mpich2-nemesis. In *Proceedings of the 2009 International Conference on Parallel Processing*, ICPP '09, pages 462–469, 2009.

[98] Ron Brightwell, Kevin Pedretti, and Trammell Hudson. Smartmap: operating system support for efficient data sharing among processes on a multi-core processor. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 25:1–25:12, 2008.

[99] Ron Brightwell. A prototype implementation of mpi for smartmap. In *Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 102–110, 2008.

[100] Hong Tang and Tao Yang. Optimizing threaded mpi execution on smp clusters. In *In Proc. of 15th ACM International Conference on Supercomputing*, ICS '01, pages 381–392. ACM Press, 2001.

[101] Humaira Kamal and Alan Wagner. FG-MPI: Fine-Grain MPI for multicore and clusters. In *The 11th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDESC)*. IEEE, April 2010.

[102] Marc Prache, Patrick Carribault, and Herv Jourdren. Mpc-mpi: An mpi implementation reducing the overall memory consumption. In *PVM/MPI'09*, pages 94–103, 2009.

[103] Zhiqiang Liu, Kaijun Ren, and Junqiang Song. Mpiactor - a multicore-architecture adaptive and thread-based mpi program accelerator. In *Proceedings of the 2010 IEEE 12th International Conference on High Performance Computing and Communications*, HPCC '10, pages 98–107, Washington, DC, USA, 2010. IEEE Computer Society.

[104] Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, and Rajeev Thakur. Toward efficient support for multithreaded mpi communication. In *Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 120–129, Berlin, Heidelberg, 2008. Springer-Verlag.

[105] Emmanuel Jeannot and Guillaume Mercier. Near-optimal placement of mpi processes on hierarchical numa architectures. In *Proceedings of the 16th international Euro-Par conference on Parallel processing: Part II*, Euro-Par'10, Berlin, Heidelberg, 2010. Springer-Verlag.

[106] Stephanie Moreaud, Brice Goglin, Raymond Namyst, and David Goodell. Optimizing mpi communication within large multicore nodes with kernel assistance. In *IPDPS Workshops'10*, 2010.

[107] David Tam, Reza Azimi, and Michael Stumm. Thread clustering: sharing-aware scheduling on smp-cmp-smt multiprocessors. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 47–58, New York, NY, USA, 2007. ACM.

[108] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, pages 129–142, New York, NY, USA, 2010. ACM.

[109] T. Scogland, P. Balaji, W. Feng, and G. Narayanaswamy. Asymmetric interactions in symmetric multi-core systems: analysis, enhancements and evaluation. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 17:1–17:12, 2008.

[110] Abhinav Bhatelé, Laxmikant V. Kalé, and Sameer Kumar. Dynamic topology aware load balancing algorithms for molecular dynamics applications. In *23rd ACM International Conference on Supercomputing*, 2009.

[111] Yow-Jian Lin and Vipin Kumar. And-parallel execution of logic programs on a shared-memory multiprocessor. *J. Log. Program.*, 10(1/2/3&4):155–178, 1991.

[112] James Dinan, D. Brian Larkins, P. Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha. Scalable work stealing. In *Conference on High Performance Computing Networking, Storage and Analysis*, 2009.

[113] Marc Tchiboukdjian, Vincent Danjean, Thierry Gautier, Fabien Le Mentec, and Bruno Raffin. A work stealing scheduler for parallel loops on shared cache multicores. In *Proceedings of the 2010 conference on Parallel processing*, Euro-Par 2010, pages 99–107, Berlin, Heidelberg, 2011. Springer-Verlag.

[114] ROSE. http://www.rosecompiler.org/.

[115] Markus Schordan and Daniel Quinlan. A source-to-source architecture for user-defined optimizations. In *Lecture Notes in Computer Science: Proc. of Joint Modular Languages Conference (JMLCO03)*, volume 2789, pages 214–223. Springer-Verlag, June 2003.

[116] IBM. *PAMI Programming Guide, SA23-2273*, 2011. http://www-304.ibm.com/support/docview.wss?uid=pub1sa23227300.