HIGH PERFORMANCE AND SCALABLE MPI INTRA-NODE COMMUNICATION MIDDLEWARE FOR MULTI-CORE CLUSTERS

DISSERTATION

Presented in Partial Fulfillment of the Requirements for

the Degree Doctor of Philosophy in the

Graduate School of The Ohio State University

By

Lei Chai, Master of Science

* * * * *

The Ohio State University

2009

Dissertation Committee:

Prof. D. K. Panda, Adviser

Prof. P. Sadayappan

Prof. F. Qin

Approved by

Adviser

Graduate Program in Computer Science and Engineering

ABSTRACT

Cluster of workstations is one of the most popular architectures in high performance computing, thanks to its cost-to-performance effectiveness. As multi-core technologies are becoming mainstream, more and more clusters are deploying multi-core processors as the build unit. In the latest Top500 supercomputer list published in November 2008, about 85% of the sites use multi-core processors from Intel and AMD. Message Passing Interface (MPI) is one of the most popular programming models for cluster computing. With increased deployment of multi-core systems in clusters, it is expected that considerable communication will take place within a node. This suggests that MPI intra-node communication is going to play a key role in the overall application performance.

This dissertation presents novel MPI intra-node communication designs, including user level shared memory based approach, kernel assisted direct copy approach, and efficient multi-core aware hybrid approach. The user level shared memory based approach is portable across operating systems and platforms. The processes copy messages into and from a shared memory area for communication. The shared buffers are organized in a way such that it is efficient in cache utilization and memory usage. The kernel assisted direct copy approach takes help from the operating system kernel and directly copies message from one process to another so that it only needs one copy and improves performance from the shared memory based approach. In this approach, the memory copy can be either CPU based or DMA based. This dissertation explores both directions and for DMA based memory copy, we take advantage of novel mechanism such as I/OAT to achieve better performance and computation and communication overlap. To optimize performance on multi-core systems, we efficiently combine the shared memory approach and the kernel assisted direct copy approach and propose a topology-aware and skew-aware hybrid approach. The dissertation also presents comprehensive performance evaluation and analysis of the approaches on contemporary multi-core systems such as Intel Clovertown cluster and AMD Barcelona cluster, both of which are quad-core processors based systems.

Software developed as a part of this dissertation is available in MVAPICH and MVAPICH2, which are popular open-source implementations of MPI-1 and MPI-2 libraries over InfiniBand and other RDMA-enabled networks and are used by several hundred top computing sites all around the world.

Dedicated to My Family

ACKNOWLEDGMENTS

I would like to thank my adviser, Prof. D. K. Panda for guiding me throughout the duration of my PhD study. I'm thankful for all the efforts he took for my dissertation. I would like to thank him for his friendship and counsel during the past years.

I would like to thank my committee members Prof. P. Sadayappan and Prof. F. Qin for their valuable guidance and suggestions.

I'm especially grateful to have had Dr. Jiesheng Wu as a mentor during my first year of graduate study. I'm also grateful to Donald Traub and Spencer Shepler for their guidance and advice during my internships at Sun Microsystems.

I would like to thank all my senior Nowlab members for their patience and guidance, Dr. Pavan Balaji, Weihang Jiang, Dr. Hyun-Wook Jin, Dr. Jiuxing Liu, Dr. Amith Mamidala, Dr. Sundeep Narravula, Dr. Ranjit Noronha, Dr. Sayantan Sur, Dr. Karthik Vaidyanathan, Dr. Abhinav Vishnu, and Dr. Weikuan Yu. I would also like to thank all my colleagues, Krishna Chaitanya, Tejus Gangadharappa, Karthik Gopalakrishnan, Wei Huang, Matthew Koop, Ping Lai, Greg Marsh, Xiangyong Ouyang, Jonathan Perkins, Ajay Sampat, Gopal Santhanaraman, Jaidev Sridhar, and Hari Subramoni. I'm especially grateful to Sayantan, Jin, Wei, Matt, and Weikuan and I'm lucky to have worked closely with them on different projects.

During all these years, I met many people at Ohio State, some of whom become very close friends, and I'm thankful for their friendship.

Finally, I would like to thank my family members, my husband Guoqiang, my son Terry, my dad and my mom. I would not have had made it this far without their love and support.

VITA

April 22, 1980	Born - Qingdao, China.
September 1999 - June 2003	B. Engr. Computer Science and Engineering, Zhejiang Univeristy, Hangzhou, China.
September 2003 - August 2004	Distinguished University Fellow, The Ohio State University.
October 2004 - December 2007	Graduate Research Associate, The Ohio State University.
June 2006 - September 2006	Summer Intern, Sun Microsystems, Austin, TX.
June 2007 - September 2007	Summer Intern, Sun Microsystems, Menlo Park, CA.
January 2008 - December 2008	Distinguished University Fellow, The Ohio State University.
January 2009 - March 2009	Graduate Research Associate, The Ohio State University.

PUBLICATIONS

L. Chai, P. Lai, H.-W. Jin and D. K. Panda, "Designing An Efficient Kernel-level and User-level Hybrid Approach for MPI Intra-node Communication on Multi-core Systems", International Conference on Parallel Processing (ICPP 2008), Sept. 2008.

L. Chai, X. Ouyang, R. Noronha and D.K. Panda, "pNFS/PVFS2 over InfiniBand: Early Experiences", Petascale Data Storage Workshop 2007, in conjunction with SuperComputing (SC) 2007, Reno, NV, November 2007.

H. -W. Jin, S. Sur, L. Chai, D. K. Panda, "Lightweight Kernel-Level Primitives for High-Performance MPI Intra-Node Communication over Multi-Core Systems", IEEE Cluster 2007 (Poster), Austin, TX, September 2007.

K. Vaidyanathan, L. Chai, W. Huang and D. K. Panda, "Efficient Asynchronous Memory Copy Operations on Multi-Core Systems and I/OAT", IEEE Cluster 2007, Austin, TX, September 2007.

R. Noronha, L. Chai, T. Talpey and D. K. Panda, "Designing NFS With RDMA For Security, Performance and Scalability", The 2007 International Conference on Parallel Processing (ICPP-07), Xi'an, China.

S. Sur, M. Koop, L. Chai and D. K. Panda, "Performance Analysis and Evaluation of Mellanox ConnectX InfiniBand Architecture with Multi-Core Platforms", 15th Symposium on High-Performance Interconnects (HOTI-15), August 2007.

L. Chai, Q. Gao and D. K. Panda, "Understanding the Impact of Multi-Core Architecture in Cluster Computing: A Case Study with Intel Dual-Core System", The 7th IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2007), May 2007.

L. Chai, A. Hartono and D. K. Panda, "Designing High Performance and Scalable MPI Intra-node Communication Support for Clusters", The IEEE International Conference on Cluster Computing (Cluster 2006), September 2006.

L. Chai, R. Noronha and D. K. Panda, "MPI over uDAPL: Can High Performance and Portability Exist Across Architectures?", The 6th IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2006), May 2006.

S. Sur, L. Chai, H.-W. Jin and D. K. Panda, "Shared Receive Queue Based Scalable MPI Design for InfiniBand Clusters", International Parallel and Distributed Processing Symposium (IPDPS 2006), April 25-29, 2006, Rhodes Island, Greece.

S. Sur, H.-W. Jin, L. Chai and D. K. Panda, "RDMA Read Based Rendezvous Protocol for MPI over InfiniBand: Design Alternatives and Benefits", Symposium on Principles and Practice of Parallel Programming (PPOPP 2006), March 29-31, 2006, Manhattan, New York City.

L. Chai, R. Noronha, P. Gupta, G. Brown, and D. K. Panda, "Designing a Portable MPI-2 over Modern Interconnects Using uDAPL Interface", EuroPVM/MPI 2005, Sept. 2005.

H.-W. Jin, S. Sur, L. Chai and D. K. Panda, "LiMIC: Support for High-Performance MPI Intra-Node Communication on Linux Clusters", International Conference on Parallel Processing (ICPP-05), June 14-17, 2005, Oslo, Norway.

L. Chai, S. Sur, H.-W. Jin and D. K. Panda, "Analysis of Design Considerations for Optimizing Multi-Channel MPI over InfiniBand", Workshop on Communication Architecture for Clusters (CAC 2005); In Conjunction with IPDPS, April 4-8, 2005, Denver, Colorado.

FIELDS OF STUDY

Major Field: Computer Science and Engineering

Studies in:

Computer Architecture	Prof. D. K. Panda
Computer Networks	Prof. D. Xuan
Software Systems	Prof. G. Agrawal

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The pace people pursuing computing power has never slowed down. Moore's Law has been proven to be true over the passage of time - the performance of microchips has been increasing at an exponential rate, doubling every two years. "In 1978, a commercial flight between New York and Paris cost around $900 and took seven hours. If the principles of Moore's Law had been applied to the airline industry the way they have to the semiconductor industry since 1978, that flight would now cost about a penny and take less than one second." (a statement from Intel) However, it becomes more difficult to speedup processors nowadays by increasing frequency. One major barrier is the overheat problem, which high-frequency CPU must deal with carefully. The other issue is power consumption. These concerns make it less cost-to-performance effective to increase processor clock rate. Therefore, computer architects have designed *multi-core* processor, which means to place two or more processing cores on the same chip [29]. Multi-core processors speedup application performance by dividing the workload to different cores. It is also referred to as *Chip Multiprocessor* (CMP).

On the other hand, clusters [4] have been one of the most popular environments in parallel computing for decades. The emergence of multi-core architecture has brought clusters into a multi-core era. As a matter of fact, multi-core processors have already been widely deployed in parallel computing. In the Top500 supercomputer list published in 2007, more than 77% processors are multi-core processors from Intel and AMD [24]. This number becomes 85% in the latest Top500 list published in November, 2008.

Message Passing Interface (MPI) [61] is one of the most popular programming models for cluster computing. With the rapid deployment of multi-core systems in clusters, more and more communication will take place inside a node, which means MPI intra-node communication will play a critical role to the overall application performance.

MVAPICH [15] is an MPI library that delivers high performance, scalability and fault tolerance for high-end computing systems and servers using InfiniBand [6], iWARP [13] and other RDMA-enabled [67] interconnect networking technologies. MVAPICH2 is MPI-2 [62] compliant. MVAPICH and MVAPICH2 are being used by more than 840 organizations world-wide to extract the potential of these emerging networking technologies for modern systems.

In this dissertation we use MVAPICH as the framework and explore the alternatives of designing MPI intra-node communication, come up with optimization strategies for multi-core clusters, and study on the factors that affect MPI intra-node communication performance. Further, we conduct in-depth evaluation and analysis on application characteristics on multi-core clusters.

The rest of the chapter is organized as follows. First we provide an overview of the architectures of multi-core processors. Then we introduce the basic MPI intra-node communication schemes. Following that we present the problem statement and our research approaches. And finally we provide an overview of this dissertation.

1.1 Architectures of Multi-core Clusters

Multi-core means to integrate two or more complete computational cores within a single chip [29]. The motivation of the development of multi-core processors is the fact that scaling up processor speed results in dramatic rise in power consumption and heat generation. In addition, it becomes more difficult to increase processor speed nowadays that even a little increase in performance will be costly. Realizing these factors, computer architects have proposed multi-core processors that speed up application performance by dividing the workload among multiple processing cores instead of using one "super fast" single processor. Multi-core processor is also referred to as *Chip Multiprocessor* (CMP). Since a processing core can be viewed as an independent processor, in this proposal we use *processor* and *core* interchangeably.

Most processor venders have multi-core products, e.g. Intel Quad-core [11] and Dual-core [9] Xeon, AMD Quad-core [21] and Dual-core Opteron [3], Sun Microsystems UltraSPARC T1 (8 cores) [25], IBM Cell [23], etc. There are various alternatives in designing cache hierarchy organization and memory access model. Figure 1.1 illustrates two typical multi-core system designs. The left box shows a NUMA [1] based dual-core system in which each core has its own L2 cache. Two cores on the same chip share the memory controller and local memory. Processors can also access remote memory, although local memory access is much faster. The right box shows a bus based dual-core system, in which two cores on the same chip share the same L2 cache and memory controller, and all the cores access the main memory through a shared bus. Intel Woodcrest processors [12] belong to this architecture. Intel Clovertown processors (quad-core) [7] are made of two Woodcrest processors. There are more advanced systems emerging recently, e.g. AMD Barcelona quad-core processors, in which four cores on the same chip have their own L2 caches but share the same L3 cache. The L3 cache is not a traditional inclusive cache, when data is loaded from the L3 cache to the L1 cache (L2 is always bypassed) the data can be removed from L3 or remain there depending on whether other cores are likely to access the data in the future. In addition, the L3 cache doesn't load data from the memory, it acts like a spill-over cache for items evicted from the L2 cache.

NUMA is a computer memory design where the memory access time depends on the memory location relative to a processor. Under NUMA, memory is shared between processors, but a processor can access its own local memory faster than non-local memory. Therefore, data locality is critical to the performance of an application. AMD systems are mostly based on NUMA architecture. Modern operating systems allocate memory in a NUMA-aware manner. Memory pages are always physically allocated local to processors where they are first touched, unless the desired memory is not available. Solaris has been supporting NUMA architecture for a number of years [71]. Linux also started to be NUMA-aware from 2.6 kernel. In our work so far we focus on Linux.

Due to its greater computing power and cost-to-performance effectiveness, multi-core processor has been deployed in cluster computing. In a multi-core cluster, there are three levels of communication as shown in Figure 1.1. The communication

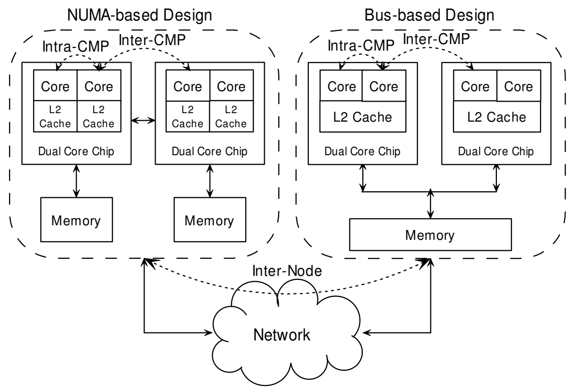


Figure 1.1: Illustration of Multi-Core Cluster

between two processors on the same chip is referred to as *intra-CMP communication* in this proposal. The communication across chips but within a node is referred to as *inter-CMP communication*. And the communication between two processors on different nodes is referred to as *inter-node communication*.

Multi-core cluster imposes new challenges in software design, both on middleware level and application level. How to design multi-core aware parallel programs and communication middleware to get optimal performance is a hot topic. There have been studies on multi-core systems. Koop, et al in [53] have evaluated the memory subsystem of Bensley platform using microbenchmarks. Alam, et al have done a scientific workloads characterization on AMD Opteron based multi-core systems [40]. Realizing the importance and popularity of multi-core architectures, researchers start to propose techniques for application optimization on multi-core systems. Some of the techniques are discussed in [36], [42], and [73]. Discussions of OpenMP on multi-core processors can be found in [39].

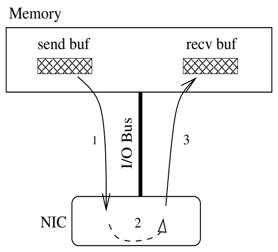
1.2 MPI Intra-node Communication

MPI stands for Message Passing Interface [61]. It is the *de facto* standard used for cluster computing. There are multiple MPI libraries in addition to MVAPICH, such as MPICH [45], MPICH2 [16], OpenMPI [17], HP MPI [5], Intel MPI [10], etc. Most clusters are built with multi-processor systems which means inter-node and intra-node communication co-exists in cluster computing. In this section we introduce the basic approaches for MPI intra-node communication.

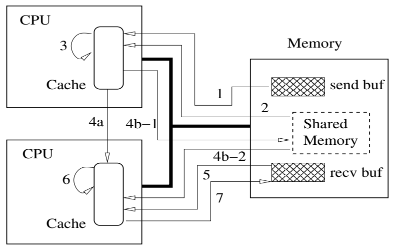
NIC-Based Message Loopback

An intelligent NIC can provide a NIC-based loopback. When a message transfer is initiated, the NIC can detect whether the destination is on the same physical node or not. By initiating a local DMA from the NIC memory back to the host memory as shown in Figure 1.2(a), we can eliminate overheads on the network link because the message is not injected into the network. However, there still exist two DMA operations. Although I/O buses are getting faster, the DMA overhead is still high. Further, the DMA operations cannot utilize the cache effect.

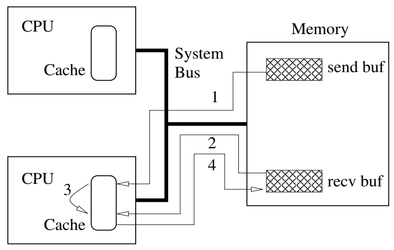
InfiniHost [59] is a Mellanox's second generation InfiniBand Host Channel Adapter (HCA). It provides internal loopback for packets transmitted between two Queue Pairs (connections) that are assigned to the same HCA port. Most of other high-speed interconnections such as Myrinet [27] and Quadrics [64] also provide NIC-based message loopback. Ciaccio [32] also utilized NIC-level loopback to implement an efficient `memcpy()`.



(a) NIC-Based Message Loopback



(b) User-Space Shared Memory



(c) Kernel-Based Memory Mapping

Figure 1.2: Memory Transactions for Different Intra-Node Communication Schemes

User-Space Shared Memory

This design alternative involves each MPI process on a local node, attaching itself to a shared memory region. This shared memory region can then be used amongst the local processes to exchange messages and other control information. The sending process copies the message to the shared memory area. The receiving process can then copy over the message to its own buffer. This approach involves minimal setup overhead for every message exchange and shows better performance for small and medium message sizes than NIC-level message loopback.

Figure 1.2(b) shows the various memory transactions which happen during the message transfer. In the first memory transaction labeled as 1; the MPI process needs to bring the send buffer to the cache. The second operation is a write into the shared memory buffer, labeled as 3. If the block of shared memory is not in cache, another memory transaction, labeled as 2 will occur to bring the block in cache. After this, the shared memory block will be accessed by the receiving MPI process. The memory transactions will depend on the policy of the cache coherency implementation and can result in either operation 4a or 4b-1 followed by 4b-2. Then the receiving process needs to write into the receive buffer, operation labeled as 6. If the receive buffer is not in cache, then it will result in operation labeled as 5. Finally, depending on the cache block replacement scheme, step 7 might occur. It is to be noted that there are at least two copies involved in the message exchange. This approach might tie down the CPU with memory copy time. In addition, as the size of the message grows, the performance deteriorates because vigorous copy-in and copy-out also destroys the cache contents for the end MPI application.

This shared memory based design has been used in MPICH-GM [63] and other MPI implementations such as MVAPICH [15]. In addition, Lumetta et al. [56] have dealt with efficient design of shared memory message passing protocol and multiprotocol implementation. MPICH-Madeleine [26] and MPICH-G2 [41, 52] also have suggested multi-protocol communication, which can provide a framework for having different channels for inter and intra-node communication.

CPU Based Kernel Modules for Memory Mapping

Kernel-Based Memory Mapping approach takes help from the operating system kernel to copy messages directly from one user process to another without any additional copy operation. The sender or the receiver process posts the message request descriptor in a message queue indicating its virtual address, tag, etc. This memory is mapped into the kernel address space when the other process arrives at the message exchange point. Then the kernel performs a direct copy from the sender buffer to the receiver application buffer. Thus this approach involves only one copy.

Figure 1.2(c) demonstrates the memory transactions needed for copying from the sender buffer directly to the receiver buffer. In step 1, the receiving process needs to bring the sending process' buffer into its cache block. Then in step 3, the receiving process can write this buffer into its own receive buffer. This may generate step 2 based on whether the block was in cache already or not. Then, depending on the cache block replacement policy, step 4 might be generated implicitly.

It is to be noted that the number of possible memory transactions for the Kernel-Based Memory Mapping is always less than the number in User-Space Shared Memory approach. We also note that due to the reduced number of copies to and from various buffers, we can maximize the cache utilization. However, there are other overheads. The overheads include time to trap into the kernel, memory mapping overhead, and TLB flush time. In addition, still the CPU resource is required to perform a copy operation. There are several previous works that adopt this approach, which include [43, 72]. We have explored the kernel based approaches, and implemented a kernel module called LiMIC which will be described in Chapter 4.

I/OAT Based Kernel Modules

As mentioned in Section 1.2, DMA based approaches usually have high overhead. Recently, Intel's I/O Acceleration Technology (I/OAT) [44, 57, 68] introduced an asynchronous DMA copy engine within the chip that has direct access to main memory to improve performance and reduce the overheads mentioned above. I/O Acceleration Technology offloads the data copy operation from the CPU with the addition of an asynchronous DMA copy engine. The copy engine is implemented as a PCI-enumerated device in the chipset and has multiple independent DMA channels with direct access to main memory. When the processor requests a block memory copy operation from the engine, it can then asynchronously perform the data transfer with no host processor intervention. When the engine completes a copy, it can optionally generate an interrupt. As mentioned in [44], I/OAT supports several interfaces in kernel space for copying data from a source page/buffer to a destination page/buffer. These interfaces are asynchronous and the copy is not guaranteed to be completed when the function returns. These interfaces return a non-negative cookie value on success, which is used to check for completion of a particular memory operation.

We have designed kernel modules to utilize I/OAT technology for memory copy. The details are described in Chapter 5.

1.3 Problem Statement

The scope of this dissertation is shown in Figure 1.3. In short, we aim to design high performance and scalable MPI intra-node communication schemes and study their impacts on applications in-depth. We intend to understand the characteristics of multi-core clusters, and optimize MPI performance on them. In Figure 1.3, the white boxes stand for the existing components, the dark shaded boxes indicate the components we have been working on, and the light shaded boxes are our future work.

We present the problem statement of this dissertation in detail as follows:

- **Can we have a significantly better understanding on application characteristics on multi-core clusters, especially with respect to communication performance, message distribution, cache utilization, and scalability?** - With the rapid emergence of multi-core architecture, clusters

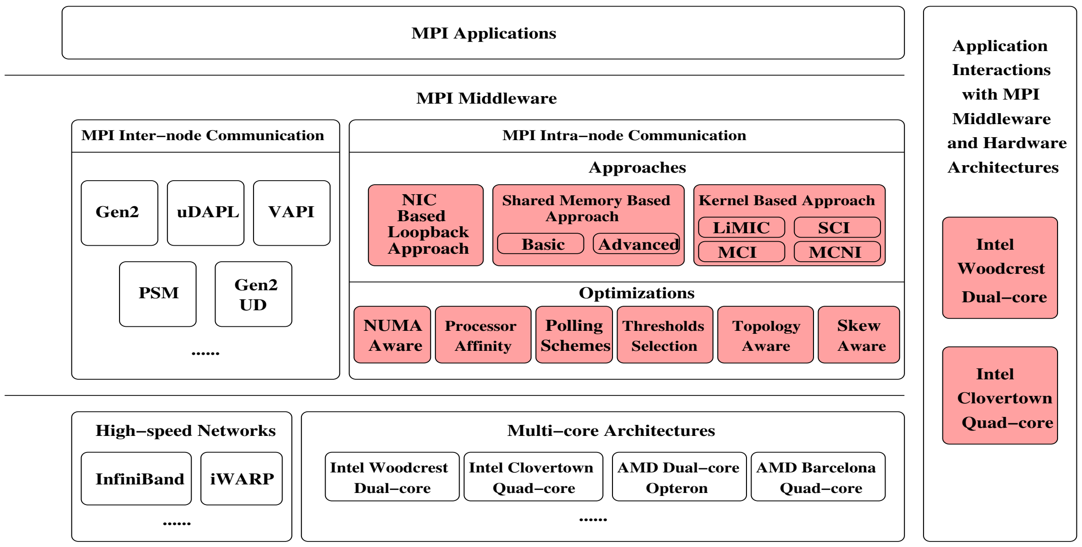


Figure 1.3: Problem Space of this Dissertation

have entered a multi-core era. In order to get optimal performance, it is crucial to have in-depth understanding on application behaviors and trends on multi-core clusters. It is also very important to identify potential bottlenecks in multi-core clusters through evaluation, and explore possible solutions. However, since multi-core is a relatively new technology, few research has been done in the literature.

- **Can we design a shared memory based approach to allow MVAPICH to have better intra-node communication performance?** - The original MVAPICH used to use NIC base loopback approach. While it eases code design - we do not need to distinguish between intra- and inter-node communication, the performance is not optimal. Further, with the emergence of multi-core systems, more and more cores can reside within one node, and the NIC based loopback approach may not be scalable since all the intra-node communication will go through the PCI bus and the PCI bus may become a bottleneck. It is essential to have a more efficient intra-node communication scheme.

- **Can we optimize the shared memory based approach to have lower latency, better cache utilization, and reduced memory usage, thus have improved performance especially on multi-core clusters?** - There are limitations in the current existing shared memory schemes. Some are not scalable with respect to memory usage, and some require locking mechanisms among processes to maintain consistency. Thus the performance is suboptimal for a large number of processes. Moreover, few research has been done to study the interaction between the multi-core systems and MPI implementations. We need to take on the challenges and optimize the current shared memory based schemes to improve MPI intra-node communication performance.

- **Can we design MVAPICH intra-node communication to utilize kernel module based approach to reduce the number of copies and potentially benefit applications?** - As mentioned in Section 1.2, one approach to avoid extra message copies is to use operating system kernel to provide a direct copy from one process to another. Inside the kernel module, it can either use CPU to do memory copy, or take advantage of any DMA engines that are available for memory copy. Since this kind of approach requires only one memory copy, it may improve MVAPICH intra-node communication performance. And if we use the DMA for memory copy, we can potentially achieve better computation and communication overlap.

- **Can we design an efficient hybrid approach that utilizes both the kernel module based approach and the shared memory based approach to get optimal performance, especially on multi-core clusters?** - User-level shared memory and kernel assisted direct copy are two popular approaches. Both of them have advantages and disadvantages. However, we do not know if one of these approaches is sufficient for multi-core clusters. In order to obtain optimized performance, it is important to have a comprehensive understanding of these two approaches and combine them effectively.

- **What are the factors that affect MVAPICH Intra-node communication and how can we tune them to get the optimal performance?** - To optimize communication performance, many MPI implementations such as MVAPICH provide multiple communication channels. These channels may be used either for intra- or inter-node communication. Two important factors that affect application performance are channel polling and threshold selection. It is important to understand how the applications perform with these factors and have efficient channel polling and threshold selection algorithms to improve on performance.

1.4 Research Approaches

In this section we present our general approaches to the above mentioned issues.

1. **Understanding the application characteristics on multi-core clusters** - We have designed a set of experiments to study the impact of multi-core architecture on cluster computing. The purpose is to give both application and communication middleware developers insights on how to improve overall performance on multi-core clusters. The study includes MPI intra-node communication characteristics on multi-core clusters, message distribution in terms of both communication channel and message size, cache utilization/potential bottleneck identification, and initial scalability study.

2. **Designing a basic user-level shared memory based approach for MPI intra-node communication** - We have designed a shared memory based implementation for MVAPICH intra-node communication. A temporary file is created and all the processes map the temporary file to their own memory spaces as a shared memory area and use this shared memory area for communication.

3. **Designing an advanced user-level shared memory based approach for MPI intra-node communication for optimized performance** - We have optimized the basic shared memory based design to get better performance and scalability. We want to achieve two goals in our design: *1. To obtain low latency and high bandwidth between processes, and 2. To have reduced memory usage for better scalability.* We achieve the first goal by efficiently utilizing the L2 cache and avoiding the use of lock. We achieve the second goal by separating the buffer structures for small and large messages, and using a shared buffer pool for each process to send large messages. We have also

explored various optimization strategies to further improve the performance, such as reducing the polling overhead, etc.

4. **Designing kernel assisted direct copy approaches to eliminate extra copies and achieve better computation and communication overlap** - We have designed two major kernel modules for MPI intra-node communication. One is called LiMIC/LiMIC2, which uses CPU based memory copy. And the other uses Intel I/OAT which is an on-chip DMA to do memory copy. We have also modified MVAPICH and MVAPICH2 to utilize the kernel modules.

5. **Designing an efficient user-level and kernel-level hybrid approach for multi-core clusters** - We have carefully considered the characteristics of the shared memory and kernel module based approaches, especially how they perform with multi-core processors. We have analyzed these approaches and come up with a topology-aware and skew-aware approach that combines the two approaches efficiently for multi-core clusters.

6. **Analyzing factors that affect multi-channel MPI performance and designing optimization schemes** - Channel polling and threshold selection are two important factors for multi-channel MPI implementations. We have designed efficient polling schemes among multiple channels. We have also explored methodologies to decide the thresholds between multiple channels. We consider latency, bandwidth, and CPU resource requirement of each channel to decide the thresholds.

1.5 Dissertation Overview

We present our research over the next several chapters. In Chapter 2, we present our study of application characteristics on multi-core clusters. We have done a comprehensive performance evaluation, profiling, and analysis using both microbenchmarks and application level benchmarks. We have several interesting observations from the experimental results, including the impact of procesor topology, the importance of MPI intra-node communication, the potential bottlenecks in multi-core systems, and scalability of multi-core clusters.

In Chapter 3, we present our shared memory based designs for MPI intra-node communication. In the shared memory based designs, all the processes map a temporary file to their own memory spaces and use it as a shared memory area for communication. We start with a basic design, in which the buffers are organized such that every process has a receive queue corresponding to every other process. We then present an advance design that reorganizes the communication buffers in a more efficient way so that we can get lower latency, higher bandwidth, and less memory usage.

In Chapters 4 and 5, we take on the challenges and design kernel assisted approaches for MPI intra-node communication. We have designed two major kernel modules, one using CPU based memory copy and other using Intel I/OAT. Both the kernel modules eliminate the extra copies and achieve better performance, and using I/OAT can also achieve better computation and communication overlap.

In Chapter 6, we use a three-step methodology to design a hybrid approach for MPI intra-node communication using two popular approaches, shared memory (MVAPICH) and OS kernel assisted direct copy (MVAPICH-LiMIC2). The study has been done on an Intel quad-core (Clovertown) cluster. We have evaluated the impacts of processor topology, communication buffer reuse, and process skew effects on these two approaches, and profiled the L2 cache utilization. And based on the results and analysis we have proposed topology-aware and skew-aware thresholds to build an efficient hybrid approach which shows promising results on multi-core clusters.

Since many MPI implementations utilize multiple channels for communication, in Chapter 7 we have studied important factors to optimize multi-channel MPI. We have proposed several different schemes for polling communication channels, including static polling scheme and dynamic polling scheme. In addition, since multiple channels can be used for MPI intra-node communication, we have also evaluated thresholds for each channel both based on raw MPI latencies and bandwidths and also CPU utilization. These optimizations demonstrate large performance improvement.

CHAPTER 2

UNDERSTANDING THE COMMUNICATION CHARACTERISTICS ON MULTI-CORE CLUSTERS

Clusters have been one of the most popular environments in parallel computing for decades. The emergence of multi-core architecture is bringing clusters into a multi-core era. In order to get optimal performance, it is crucial to have in-depth understanding on application behaviors and trends on multi-core clusters. It is also very important to identify potential bottlenecks in multi-core clusters through evaluation, and explore possible solutions. In this chapter, we design a set of experiments to study the impact of multi-core architecture on cluster computing. We aim to answer the following questions:

- What are the application communication characteristics on multi-core clusters?
- What are the potential communication bottlenecks in multi-core clusters and how to possibly avoid them?
- Can multi-core clusters scale well?

The rest of the chapter is organized as follows: In Section 2.1 we describe the methodology of our evaluation. The evaluation results and analysis are presented in Section 2.2. Finally we summarize the results and impact of this work in Section 2.3.

2.1 Design of Experiments for Evaluating Multi-core Clusters

To answer the questions mentioned in the beginning of this chapter, we describe the evaluation methodology and explain the design and rational of each experiment.

2.1.1 Programming Model and Benchmarks

We choose to use MPI [14] as the programming model because it is the *de facto* standard used in cluster computing. The MPI library used is MVAPICH2 [15]. In MVAPICH2, intra-node communication, including both intra-CMP and inter-CMP, is achieved by user level memory copy.

We evaluate both microbenchmarks and application level benchmarks to get a comprehensive understanding on the system. Microbenchmarks include latency and bandwidth tests. And application level benchmarks include HPL from HPCC benchmark suite [47], NAMD [65] apoa1 data set, and NAS parallel benchmarks [38].

2.1.2 Design of Experiments

We have designed to carry out four sets of experiments for our study: latency and bandwidth, message distribution, potential bottleneck identification, and scalability tests. We describe them in detail below.

- Latency and Bandwidth: These are standard ping-pong latency and bandwidth tests to characterize the three levels of communication in multi-core cluster: intra-CMP, inter-CMP, and inter-node communication.

- Message Distribution: We define message distribution as a two dimensional metric. One dimension is with respect to the communication channel, i.e. the percentage of traffic going through intra-CMP, inter-CMP, and inter-node respectively. The other dimension is in terms of message size. This experiment is very important because understanding message distribution facilitates communication middleware developers, e.g. MPI implementors, to optimize critical communication channels and message size range for applications. The message distribution is measured in terms of both number of messages and data volume.

- Potential Bottleneck Identification: In this experiment, we run application level benchmarks on different configurations, e.g. four processes on the same node, four processes on two different nodes, and four processes on four different nodes. We want to discover the potential bottlenecks in multi-core cluster if any, and explore approaches to alleviate or eliminate the bottlenecks. This will give insights to application writers how to optimize algorithms and/or data distribution for multi-core cluster. We also design an example to demonstrate the effect of multi-core aware algorithm.

- Scalability Tests: This set of experiments is carried out to study the scalability of multi-core cluster.

2.1.3 Processor Affinity

In all our experiments, we use *sched_affinity* system call to ensure the binding of process with processor. The effect of processor affinity is two-fold. First, it eases our analysis, because we know exactly the mapping of processes with processors. And second, it makes application performance more stable, because process migration requires cache invalidation and may degrade performance.

2.2 Performance Evaluation

In this section we present the experimental results and the analysis of the results. We use the format pxq to represent a configuration. Here p is the number of nodes, and q is the number of processors per node.

Evaluation Platforms: We use two multi-core clusters and one single-core cluster for the experiments. Their setup is specified below:

Cluster A: Cluster A consists of 4 Intel Bensley systems connected by InfiniBand. Each node is equipped with two sets of dual-core 2.6GHz Woodcrest processor, i.e. 4 processors per node. Two processors on the same chip share a 4MB L2 cache. The overall architecture is similar to that shown in the right box in Figure 1.1. However, Bensley system has added more dedicated memory bandwidth per processor by doubling up on memory buses, with one bus dedicated to each of Bensley's two CPU chips. The InfiniBand HCA is Mellanox MT25208 DDR and the operating system is Linux 2.6.

Cluster B: Cluster B is an Intel Clovertown cluster with 72 nodes. Each node is equipped with dual quad-core Xeon processor, i.e. 8 cores per node, running at 2.0GHz. Each node has 4GB main memory. The nodes are connected by Mellanox

InfiniBand DDR cards. The operating system is Linux 2.6.18 We use 32 nodes in Cluster B for our experiments.

Cluster C: Cluster C is a single-core Intel cluster connected by InfiniBand. Each node is equipped with dual Intel Xeon 3.6GHz processor and each processor has a 2MB L2 cache. Cluster C is used to compare the scalability.

In the following sections, Cluster A is used by default unless specified explicitly.

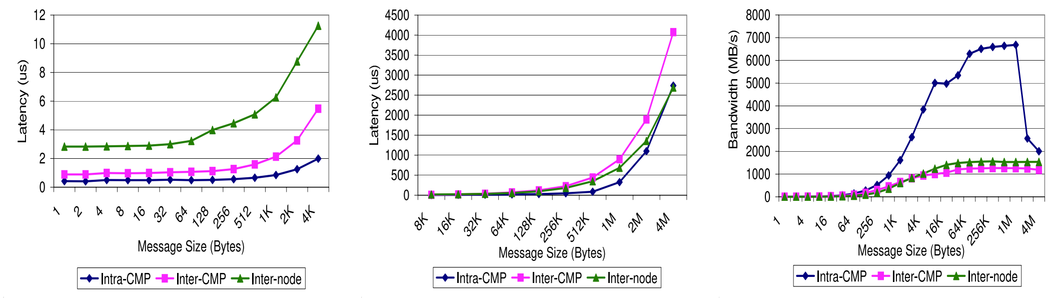
2.2.1 Latency and Bandwidth

Figure 2.1 shows the basic latency and bandwidth of the three levels of communication in a multi-core cluster. The numbers are taken at the MPI level. The small message latency is 0.42us, 0.89us, and 2.83us for intra-CMP, inter-CMP, and inter-node communication respectively. The corresponding peak bandwidth is 6684MB/s, 1258MB/s, and 1532MB/s.

From Figure 2.1 we can see that intra-CMP performance is far better than inter-CMP and inter-node performance, especially for small and medium messages. This is because in Intel Bensley system two cores on the same chip share the same L2 cache. Therefore, the communication just involves two cache operations if the communication buffers are in the cache. From the figure we can also see that for large messages, inter-CMP performance is not as good as inter-node performance, although memory performance is supposed to be better than network performance. This is because the intra-node communication is achieved through a shared buffer, where two memory copies are involved. On the other hand, the inter-node communication uses the Remote Direct Memory Access (RDMA) operation provided by

InfiniBand and rendezvous protocol [55], which forms a zero-copy and high performance scheme. This also explains why for large messages (when the buffers are out of cache) intra-CMP and inter-node perform comparably.

This set of results indicate that to optimize MPI intra-node communication performance, one way is to have better L2 cache utilization to keep communication buffers in the L2 cache as much as possible, and the other way is to reduce the number of memory copies. We have proposed a preliminary enhanced MPI intra-node communication design in our previous work [30].



(a) Small Message Latency

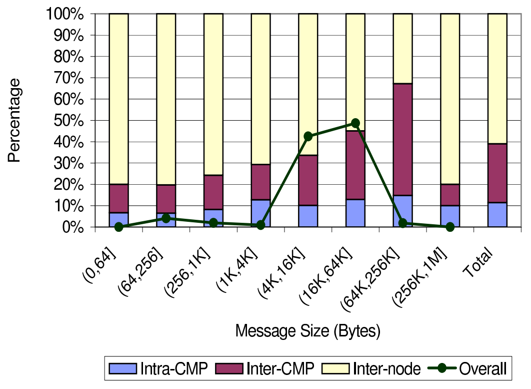
(b) Large Message Latency

(c) Bandwidth

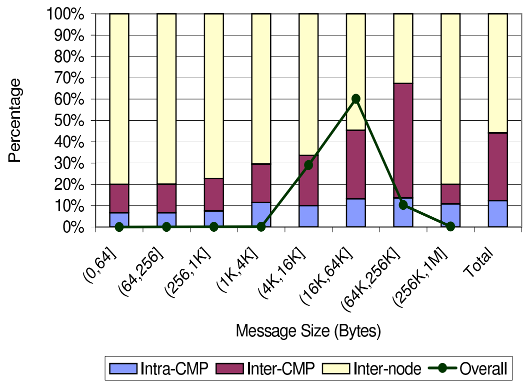
Figure 2.1: Latency and Bandwidth in Multi-core Cluster

2.2.2 Message Distribution

As mentioned in Section 2.1, this set of experiments is designed to get more insights with respect to the usage pattern of the communication channels, as well as the message size distribution. In this section, we first present the results measured on Cluster A and then present the results on Cluster B.

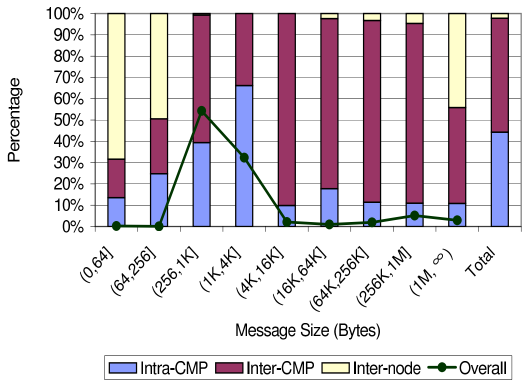


(a) Number of Messages

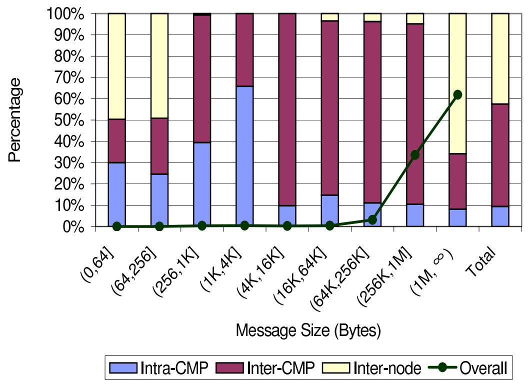


(b) Data Volume

Figure 2.2: Message Distribution of NAMD on 16 Cores



(a) Number of Messages



(b) Data Volume

Figure 2.3: Message Distribution of HPL on 16 Cores

Message Distribution on Cluster A: Figures 2.2 and 2.3 show the profiling results for NAMD and HPL respectively. The results for NAS benchmarks are listed in Table 6.1. The experiments are carried out on a 4x4 configuration and the numbers are the average of all the processes.

Figures 2.2 and 2.3 are interpreted as the following. Suppose there are n messages transferred during the application run, in which m messages are in the range $(a, b]$. Also suppose in these m messages, $m1$ are transferred through intra-CMP, $m2$ through inter-CMP, and $m3$ through inter-node. Then:

- Bar Intra-CMP(a, b] = m1/m
- Bar Inter-CMP(a, b] = m2/m
- Bar Inter-node(a, b] = m3/m
- Point Overall(a, b] = m/n

From Figure 2.2 we have observed that most of the messages in NAMD are of size 4KB to 64KB. Messages in this range take more than 90% of the total number of messages and byte volume. Optimizing medium message communication is important to NAMD performance. In the 4KB to 64KB message range, about 10% messages are transferred through intra-CMP, 30% are transferred through inter-CMP, and 60% are transferred through inter-node. This is interesting and kind of surprising. Intuitively, in a cluster environment intra-node communication is much less than inter-node communication, because a process has much more inter-node peers than intra-node peers. E.g. in our testbed, a process has 1 intra-CMP peer, 2 inter-CMP peers, and 12 inter-node peers. If a process has the same chance to communicate with every other process, then theoretically:

- Intra-CMP = 1/15 = 6.7%
- Inter-CMP = 2/15 = 13.3%
- Inter-node = 12/15 = 80%

If we call this distribution *even distribution*, then we see that intra-node communication in NAMD is well above that in even distribution, for almost all the message sizes. Optimizing intra-node communication is as important as optimizing inter-node communication to NAMD.

From Figure 2.3 we observe that most messages are small messages in HPL, from 256 bytes to 4KB. However, with respect to data volume messages larger than 256KB take more percentage. We also find that almost all the messages are transferred through intra-node in our experiment. However, this is a special case. In HPL, a process only talks to processes on the same row or column with itself. In our 4x4 configuration, a process and its row or column peers are always mapped to the same node, therefore, almost all the communication take place within a node. We have also conducted the same experiment on a 32x8 configuration for HPL. The results are shown later in this section.

Table 6.1 presents the total message distribution in NAS benchmarks, in terms of communication channel. Again, we see that the amount of intra-node (intra-CMP and inter-CMP) communication is much larger than that in even distribution for most benchmarks. On an average, about 50% messages going through intra-node communication. This trend is not random. It is because most applications have certain communication patterns, e.g. row or column based communication, ring based communication, etc. which increase the intra-node communication chance.

Therefore, even in a large multi-core cluster, optimizing intra-node communication is critical to the overall application performance.

Table 2.1: Message Distribution in NAS Benchmarks Class B on 16 Cores

metric	bench.	intra-cmp	inter-cmp	inter-node
number of messages	IS	13%	18%	69%
	FT	9%	16%	75%
	CG	45%	45%	10%
	MG	32%	32%	36%
	BT	1%	33%	66%
	SP	1%	33%	66%
	LU	1%	50%	49%
data volume	IS	7%	13%	80%
	FT	7%	13%	80%
	CG	36%	37%	27%
	MG	25%	25%	50%
	BT	0	33%	67%
	SP	0	33%	67%
	LU	0	50%	50%

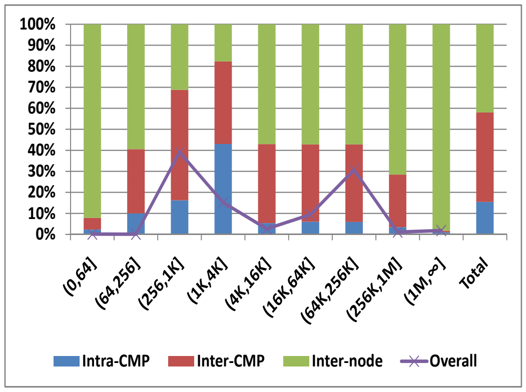
Message Distribution on Cluster B: Figure 2.4 shows the message distribution of HPL on Cluster B with a 32x8 configuration. In this configuration, the even distribution is calculated as follows:

- Intra-CMP = 1/255 = 0.4%
- Inter-CMP = 7/255 = 2.7%
- Inter-node = 248/255 = 96.1%

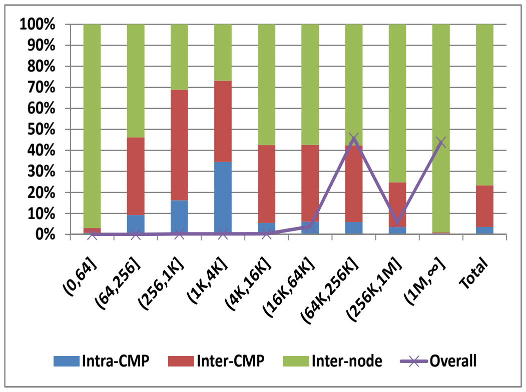
From the experimental results we see that the percentage of intra-node traffic is much higher than that in even distribution. The overall message distribution during HPL execution is summarized as the follows:

- Intra-CMP = 15.4% (number of messages), 3.5% (data volume)
- Inter-CMP = 42.6% (number of messages), 19.9% (data volume)
- Inter-node = 42.0% (number of messages), 76.6% (data volume)

The NAS message distribution on Cluster B is shown in Table 2.2 which shows the same trend that the intra-node traffic is much higher than that in even distribution for many applications. From this set of experiments we can conclude that even in a large cluster, intra-node communication plays a critical role.



(a) Number of Messages

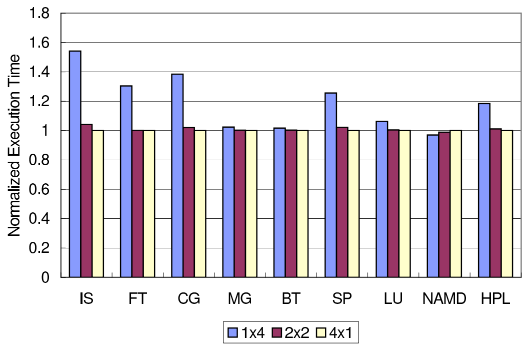


(b) Data Volume

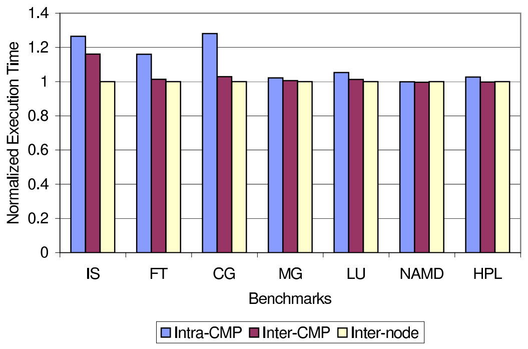
Figure 2.4: Message Distribution of HPL on 256 Cores

Table 2.2: Message Distribution in NAS Benchmarks Class C on 256 Cores

metric	bench.	intra-cmp	inter-cmp	inter-node
number of messages	IS	1%	4%	95%
	FT	1%	3%	96%
	CG	23%	47%	30%
	MG	15%	32%	53%
	BT	0%	29%	71%
	SP	0%	29%	71%
	LU	0%	47%	53%
data volume	IS	1%	4%	95%
	FT	1%	2%	97%
	CG	20%	41%	39%
	MG	20%	19%	61%
	BT	0	29%	71%
	SP	0	29%	71%
	LU	0	47%	53%



(a) 4 Processes



(b) 2 Processes

Figure 2.5: Application Performance on Different Configurations

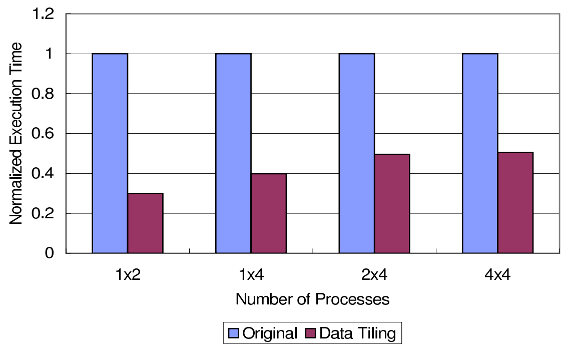


Figure 2.6: Effect of Data Tiling

2.2.3 Potential Cache and Memory Contention

In this experiment, we run all the benchmarks on 1x4, 2x2, and 4x1 configurations respectively, to examine the potential bottleneck in the system. As mentioned in the beginning of Section 2.2, we use the format pxq to represent a configuration, in which p is the number of nodes, and q is the number of processors per node. The results are shown in Figure 2.5(a). The execution time is normalized to that on 4x1 configuration.

One of the observations from Figure 2.5(a) is that 1x4 configuration does not perform as well as 2x2 and 4x1 configurations for many applications, e.g. IS, FT, CG, SP, and HPL. This is because in 1x4 configuration all the cores are activated for execution. As described earlier, on our evaluation platform, two cores on the same chip share the L2 cache and memory controller, thus cache and memory contention is a potential bottleneck. Memory contention is not a problem for processors on different chips, because Intel Bensley system has dedicated bus for each chip for higher memory bandwidth. This is why 2x2 and 4x1 configurations perform comparably.

The same trend can be observed from Figure 2.5(b). In this experiment, we run 2 processes on 2 processors from the same chip, 2 processors across chips, and 2 processors across nodes respectively. We see that inter-CMP and inter-node performance are comparable and higher than intra-CMP. The only special case is IS, whose inter-CMP performance is noticeably lower than inter-node. This is because IS uses many large messages and inter-node performs better than inter-CMP for large messages as shown in Figure 2.1.

This set of experiments indicates that to fully take advantage of multi-core architecture, both communication middleware and applications should be multi-core aware to reduce cache and memory contention. Communication middleware should avoid cache pollution as much as possible, e.g. increase communication buffer reuse [30], use cache bypass memory copy [28], or eliminate intermediate buffer [49]. Applications should be optimized to increase data locality. E.g. Data tiling [51] is a common technique to reduce unnecessary memory traffic. If a large data buffer is to be processed multiple times, then instead of going through the whole buffer multiple times, we can divide the buffer into smaller chunks and process the buffer in a chunk granularity so that the data chunks stay in the cache for multiple operations. We show a small example in the next section to demonstrate how data tiling can potentially improve application performance on multi-core system.

2.2.4 Benefits of Data Tiling

To study the benefits of data tiling on multi-core cluster, we design a microbenchmark, which does computation and communication in a ring-based manner. Each process has a piece of data (64MB) to be processed for a number of iterations.

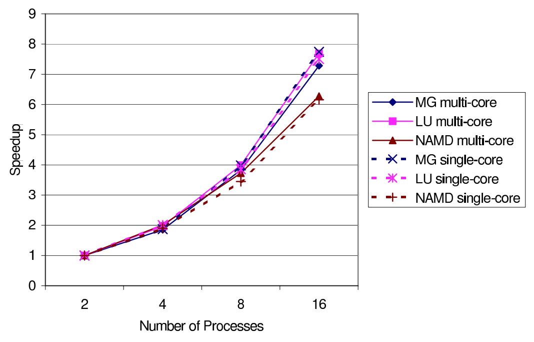
During execution, each process computes on its own data, sends them to its right neighbor and receives data from its left neighbor, and then starts another iteration of computation. In the original scheme, the data processed in the original chunk size (64MB) while in the data tiling scheme, the data are divided in to smaller chunks in the size of 256KB, which can easily fit in L2 cache.

Figure 2.6 shows the benefits of data-tiling, from which we observe that the execution time reduced significantly. This is because in the tiling case, since the intra-node communication is using CPU-based memory copy, the data are actually preloaded into L2 cache during the communication. In addition, we observe that in the cases where 2 processes running on 2 cores on the same chip, since most communication happens in L2 cache in data tiling case, the improvement is most significant, around 70% percent. The improvement in the case where 4 processes running on 4 cores on the same node, 8 processes running on 2 nodes, and 16 processes running on 4 nodes is 60%, 50%, and 50% respectively. The improvements are not as large as that in the 2 process case because the communication of inter-CMP and inter-node is not as efficient as the intra-CMP for 256KB message size.

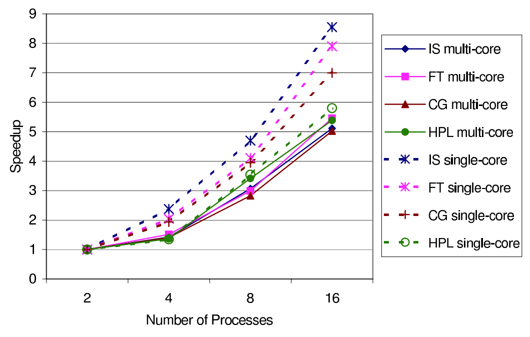
2.2.5 Scalability

In this section we present our initial results on multi-core cluster scalability. We also compare the scalability of multi-core cluster with that of single-core cluster. The results are shown in Figure 2.7. It is to be noted that the performance is normalized to that on 2 processes, so 8 is the ideal speedup for the 16 process case.

It can be seen from Figure 2.7(a) that some applications show almost ideal speedup on multi-core cluster, e.g. LU and MG. Compared with single-core cluster



(a) MG, LU, and NAMD



(b) IS, FT, CG, and HPL

Figure 2.7: Application Scalability

scalability, we find that for applications that show cache or memory contention in Figure 2.5(a), such as IS, FT, and CG, the scalability on single-core cluster is better than that on multi-core cluster. For other applications such as MG, LU and NAMD, multi-core cluster shows the same scalability as single-core cluster. As an initial study we find that multi-core cluster is promising in scalability.

2.3 Summary

In this chapter we have done a comprehensive performance evaluation, profiling, and analysis on multi-core cluster, using both microbenchmarks and application level benchmarks. We have several interesting observations from the experimental results that give insights to both application and communication middleware developers. From microbenchmark results, we see that there are three levels of communication in a multi-core cluster with different performances: intra-CMP, inter-CMP, and inter-node communication. Intra-CMP has the best performance because data can be shared through L2 cache. Large message performance of inter-CMP is not as good as inter-node because of memory copy cost. With respect to applications, the first observation is that counter-intuitively, much more intra-node communication takes place in applications than that in even distribution, which indicates that optimizing intra-node communication is as important as optimizing inter-node communication in a multi-core cluster. Another observation is that when all the cores are activated for execution, cache and memory contention may prevent the multi-core system from achieving best performance, because two cores on the same chip share the same L2 cache and memory controller. This indicates that communication middleware and applications should be written in a multi-core aware manner to get optimal performance. We have demonstrated an example on application optimization technique which improves benchmark performance by up to 70%. Compared with single-core cluster, multi-core cluster does not scale well for applications that show cache/memory contention. However, for other applications multi-core cluster has the same scalability as single-core cluster.

CHAPTER 3

SHARED MEMORY BASED DESIGN

As mentioned in Section 1.2, there exist several mechanisms for MPI intra-node communication, including *NIC-based loopback, kernel-assisted memory mapping,* and *user space memory copy.*

The user space memory copy scheme has several advantages. It provides much higher performance compared to NIC-based loopback. In addition, it is portable across different operating systems and versions. Due to these advantages, in this chapter we present our shared memory based designs.

The rest of the chapter is organized as follows: In Section 3.1 we describe the basic design of our shared memory based approach. We present the advanced design in Section 3.2 which improves both performance and memory usage over the basic design. The evaluation results and analysis are presented in Section 3.3. Finally we summarize the results and impact of this work in Section 3.4.

3.1 Basic Shared Memory Based Design

In this section we describe the basic shared memory based design and optimizations for MVAPICH.

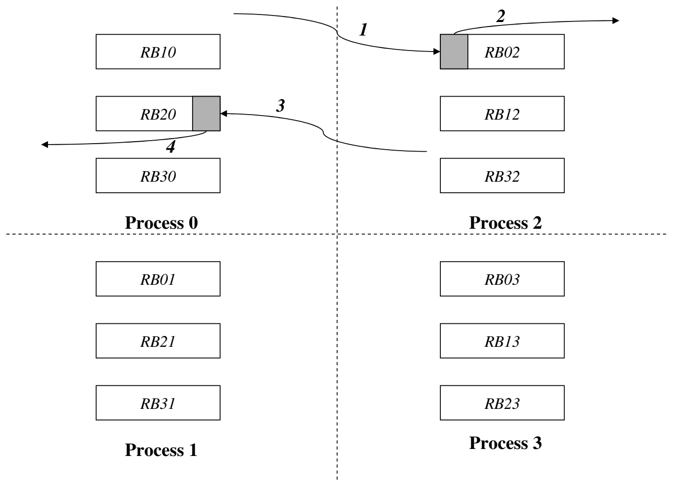


Figure 3.1: Basic Shared Memory Based Design

3.1.1 Design

The shared memory area is essentially a temporary file created by the first process on a node. The file name consists of the host name, the process id, and the user id, so that multiple jobs submitted by different users can run simultaneously on a node. Then all the processes map the shared memory area to their own memory space by calling *mmap()* system call. The shared memory area is then used for communication among local processes.

The shared memory area is essentially used as a FIFO queue. The sender writes data to the queue and the receiver reads data from the queue. There are two volatile variables that indicate how many bytes have been written to the queue and how many have been read out of the queue. The sender and the receiver change the values of these two variables respectively. The receiver polls on these two variables from time to time to detect incoming messages. If they do not match it indicates there are new data written to the queue and it can pull the data out. Message matching is performed based on *source rank*, *tag*, and *context id* which identifies the communicator. Message ordering is ensured by the memory consistency model and use of memory barrier if the underlying memory model is not consistent.

To avoid locking, each pair of processes on the same node allocate two shared memory buffers between them for exchanging messages to each other. If P processes are present on the same node, the total size of the shared memory region that needs to be allocated will be *P*(P-1)*BufSize*, where *BufSize* is the size of each shared buffer. As an example, Figure 3.1 illustrates the scenario for four processes on the same node. Each process maintains three shared buffers represented with $RBxy$, which refers to a Receive Buffer of process y that holds messages particularly sent by process x.

Eager protocol: Small messages are sent eagerly. Figure 3.1 illustrates an example where processes 0 and 2 exchange messages to each other in parallel. The sending process writes the data from its source buffer into the shared buffer corresponding to the designated process (Steps 1 and 3). After the sender finishes copying the data, then the receiving process copies the data from the shared buffer into its destination local buffer (Steps 2 and 4).

Rendezvous protocol: Since there is a limit on the shared buffer size, messages larger than the total shared buffer size cannot be sent eagerly. We use a rendezvous protocol for large messages, explained below:

- Step 1: Sender sends a *request_to_send* message.
- Step 2: Upon receiving the *request_to_send* message,the receiver acknowledges by sending back an *ok_to_send* message.

- Step 3: Upon receiving the *ok_to_send* message, the sender sends the actual data chunk by chunk. If the shared buffer is used up before the message completes, the sender will insert a *request_to_send* message again to indicate there is more data to come, and the receiver will acknowledge with an *ok_to_send* message when there is freed space in the shared buffer.

3.1.2 Optimization for NUMA systems

As mentioned in Section 1.1, accessing a processor's local memory is much more efficient than accessing remote memory on NUMA systems. Since the shared memory area is frequently used throughout the application run, it is wise to allocate it in either the sender or the receiver's memory. We choose to allocate it in sender's memory because if we allocate it in the receiver's memory, then the sender always needs to go through the long latency and put the data into a remote memory. Since the sender usually just sends out a message and proceeds with its work, this will always delay the sender. Whereas if we allocate it in the sender's memory, there are cases that it takes some time for the receiver to come to the receive point after the sender sends out the message (process skew), and in these cases the delay caused by accessing the remote memory is usually negligible compared to the process skew.

Most recent operating systems are NUMA aware and allocate buffers in the local memory of the process which first touches them. Therefore, we let all the processes touch their send buffers in the MPI initialization phase to make sure the shared buffers are allocated in the sender's memory. By touching the buffers in advance, we also save the time to allocate physical memory during application's run time, because the operating systems usually allocate physical memory when processes are really touching the buffers.

3.1.3 Exploiting Processor Affinity

Although we try to allocate buffers in the sender's local memory, the operating system may migrate a process to some other processor at a later stage due to the reason of load balancing, thus make the process away from its data. To prevent process migration, we want to bind a process to a specific processor. Under Linux 2.6 kernel, this can be accomplished by using the *sched_setaffinity* system call [37]. We apply this approach to our design to keep the data locality. Processor affinity is also good for multi-core processor systems, because it prevents a process migrating away from the cache which contains its data.

3.2 Advanced Shared Memory Based Design

In this section, we provide a detailed illustration of our advanced shared memory based design and the results.

Our design goal is to develop a shared memory communication model that is efficient and scalable with respect to both performance and memory usage. In the following subsections, we start with the overall design architecture, followed by a description on how the algorithm of intra-node communication works. Design analysis and several optimization strategies are presented in the end of this section.

3.2.1 Overall Architecture

Throughout this section, we use a notation P to symbolize the number of processes running in the same node. Each process has $P-1$ small-sized *Receive Buffers*

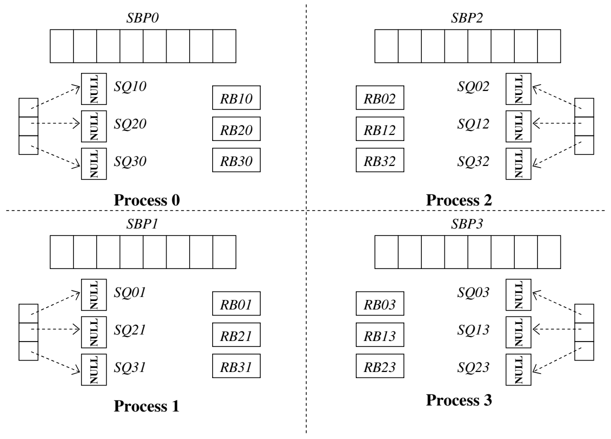


Figure 3.2: Overall Architecture of the Proposed Design

(RB), one *Send Buffer Pool (SBP)*, and a collection of $P-1$ *Send Queues (SQ)*. Figure 3.2 illustrates the overall architecture, where four processes are involved in the intra-node communication. In this illustration, we use notations x and y to denote a process local ID. The shared memory space denoted as $RBxy$ refers to a Receive Buffer of process y, which retains messages specifically sent by process x. A Send Buffer Pool that belongs to a process with local ID x is represented with $SBPx$. A buffer in the pool is called a *cell*. Every process owns an array of pointers, where each pointer points to the head of a queue represented with $SQxy$, which refers to a Send Queue of process y that holds data directed to process x.

The sizes of the receive buffer and the buffer cell as well as the number of cells in the pool are tunable parameters that can be determined empirically to achieve optimal performance. Based on our experiments, we choose to set the size of receive buffer to be 32 KB, the size of the buffer cell to be 8 KB, and the total number of cells in each send buffer pool to be 128.

3.2.2 Message Transfer Schemes

From our past experience, transferring small messages usually occurs more frequently than large messages. Therefore, sending small messages should be prioritized and handled efficiently with the purpose of improving the overall performance. In our design, small messages are exchanged through copying directly into receiving process' receive buffer. This approach is so simple that extra overhead is minimized. On the other hand, as the message size grows, the memory size required for the data transfer increases as well, which may lead to performance degradation if it is not handled properly. Therefore, we suggest different ways of handling small and large messages.

The workflows of sending and receiving small and large messages are presented in the following.

Small Message Transfer Procedure

Figure 3.3 depicts how a small message is transferred by one process and retrieved by another. In this example, process 0 is the sender, while process 1 is the receiver. The figure does not show the processes 2 and 3 since they do not participate in the data transfer. The send/receive mechanism for small messages is straightforward as explained below.

1. The sending process directly accesses the receiving process' receive buffer to write the actual data to be sent, which is obtained from the source buffer.

2. The receiving process copies the data from its receive buffer into its final spot in the destination buffer.

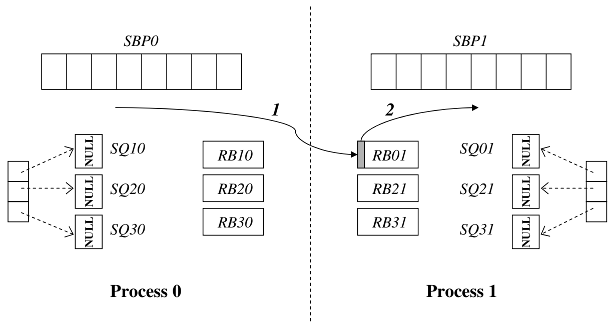


Figure 3.3: Send/Receive Mechanism for a Small Message

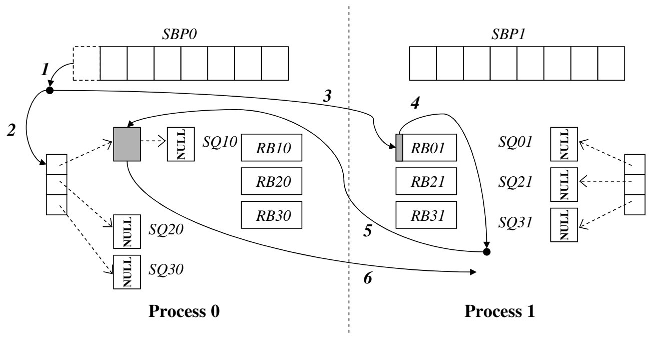


Figure 3.4: Send/Receive Mechanism for a Large Message

This procedural simplicity minimizes unnecessary setup overhead for every message exchange.

Large Message Transfer Procedure

Figure 3.4 demonstrates a send/receive progression between two processes, where process 0 sends a message to process 1. For compactness, processes 2 and 3 are not shown in the figure since they are not involved in the communication process.

A sending procedure comprises of the following three steps:

1. The sending process fetches a free cell from its send buffer pool, copies the message from its source buffer into the free cell, and then marks the cell *busy*.

2. The process enqueues the loaded cell into the corresponding send queue.

3. The process sends a control message, which contains the address location information of the loaded cell, and writes it into the receiving process' receive buffer.

A receiving procedure consists of the following three steps:

4. The receiving process reads the received control message from its receive buffer to get the address location of the cell containing the data being transferred.

5. Using the address information obtained from the previous step, the process directly accesses the cell containing the transferred data, which is stored in the sending process' send queue.

6. The process copies the actual data from the referenced cell into its own destination buffer, and subsequently marks the cell *free*.

In this design, when the message to be transferred is larger than the cell size, it is packetized into smaller packets, each transferred independently. The packetization contributes to a better throughput because of the pipelining effect, where the receiver can start copying the data out before the entire message is completely copied in.

In Steps 1 and 6, a cell is marked *busy* and *free*, respectively. A busy cell indicates that the cell has been loaded with the data and should not be disturbed until the corresponding receiver finishes reading the data in the cell; whereas a free cell simply indicates that the cell can be used for transferring a new message. After the receiving process marks a cell free, the free cell remains residing in the sending process' send queue, until reclaimed by the sender. The cell reclamation process is done by the sender at the time it initiates a new data transfer (Step 1). We call this cell reclamation scheme *mark-and-sweep*.

Transferring large messages utilizes *indirection*, which means the sender puts a control message to the receiver's receive buffer to instruct the receiver to get the actual data. There are two reasons to use indirection instead of letting the receiver poll both its receive buffer and the send queue corresponding to it at the sender side. First, polling more buffers adds unnecessary overhead; and second, the receiver needs to explicitly handle message ordering if messages come from different channels.

3.2.3 Analysis of the Design

In this section we analyze our proposed design based on the important issues in designing an efficient and scalable shared memory model.

Lock Avoidance

A locking mechanism is required to maintain consistency when two or more processes attempt to access a shared resource. A locking operation carries a fair amount of overhead and may delay memory activity from other processes. Therefore, it is desirable to design a lock-free model.

In our design, locking is avoided by imposing a rule that only one reader and one writer exist for each resource. It is obvious that there are only one reader and one writer for each send queue and receive buffer, hence they are free from locking mechanism. However, enforcing one-reader-one-writer rule on the send buffer pools can be tricky. After a receiving process finishes copying data from a cell, the cell needs to be placed back into the sender's send buffer pool for future reuse. Intuitively, the receiving process should be the one that returns the cell back into the send buffer pool, however, this may lead to multiple processes returning free cells to one sending process at the same time and cause consistency issue. In order to maintain both consistency and good performance, we use a *mark-and-sweep* technique to impose the one-reader-one-writer rule on the send buffer pools, as explained in Section 3.2.2.

Effective Cache Utilization

In this section we analyze the cache utilization for small and large messages respectively. In our design, small messages are transferred through receive buffers directly. Since the receive buffers are solely designed for small messages, the buffer size can be really small that it can completely fit in the cache. Therefore, successive accesses into the same receive buffer will result in more cache hits and lead to a better performance.

In the communication design for large messages, after the receiver finishes copying data out from the loaded cell, the cell will be marked free and reclaimed by the sender for future reuse. Since the sender can reuse cells that it used previously, there is a chance that the cells are still resident in the cache, therefore, the sender gets the benefit that it does not need to access the memory for every send. If the receiver also has the same cell in its cache, then the receiver also does not need to access the memory, because only cache-to-cache transfer is needed.

Efficient Memory Usage

We first illustrate the scalability issue in the current MVAPICH intra-node communication support. As we mentioned in Section 3.1, the basic shared memory based design allocates a shared memory region of size $P * (P-1) * BufSize$, where *BufSize* is the size of each receive buffer (1 MB by default). This implies that the shared memory consumption becomes huge for large values of P.

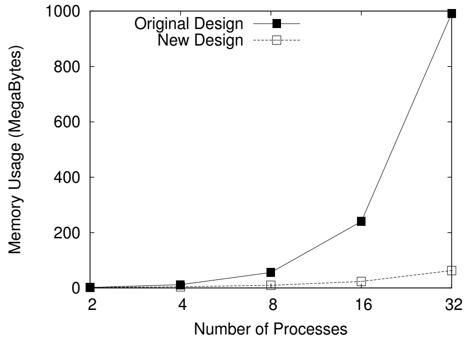


Figure 3.5: Memory Usage of the Proposed New Design Within a Node

In contrast, the proposed design provides a better scalability as it only necessitates one send buffer pool per process, regardless of how many processes participate in the intra-node communication. The new design uses the same method as the original MVAPICH design for small message communication, which requires $P * (P-1)$ number of receive buffers. Despite such polynomial complexity, the total memory space pre-allocated for receive buffers is still low due to the small size design of receive buffers. It is to be noted that simply reducing the receive buffer size in the basic design is not practical because large messages will suffer from lack of shared memory space. Simply having a send buffer pool without the receive buffers might be also not efficient because small messages may waste a large portion of the buffer.

We calculated the total shared memory usage of both MVAPICH (the original design) and the new design. In Figure 3.5, we can observe that the shared memory consumption of the new design is substantially lower than the original design when the number processes that are involved in the intra-node communication gets larger.

3.2.4 Optimization Strategies

We discuss several optimization strategies to our design in order to further improve performance.

Reducing Polling Overhead

Each process needs to poll its receive buffers to detect incoming new messages. Two variables are maintained for buffer polling: *total-in* and *total-out*, which keep track of how many bytes of data have entered and exited the buffer. When *total-in* is equal to *total-out*, it means there is no new messages residing in the polled buffer. If *total-in* is greater than *total-out*, it means the polled buffer contains a new message. *total-in* can never be less than *total-out*.

In our design, every process has $P-1$ receive buffers that it needs to poll. To alleviate this polling overhead, we arrange the two variables (i.e. *total-in* and *total-out*) associated with the $P-1$ buffers in a contiguous array. Such arrangement will significantly reduce the polling time by exploiting cache spatial locality, where the variables can be accessed directly from the cache.

Reducing Indirection Overhead

Utilizing the indirection technique, which is explained in Section 3.2.2, results in additional overhead because, to retrieve a message, the receiving process needs to perform two memory accesses: to read the control message and to read the actual data packet. Our solution to alleviate this overhead is to associate only one control message with multiple data packets. But it is to be noted that if we send too many data packets before sending any control message, the receiver might not be able to detect incoming messages timely. Thus the optimal value of the number of control messages should be determined experimentally.

3.3 Performance Evaluation

In this section, we present the performance evaluation of the advanced shared memory based intra-node communication design, and compare it with the basic shared memory based design. The latency and bandwidth experiments were carried out on both NUMA and dual core NUMA clusters. We also present the application performance on Intel Clovertown systems at the end of this section.

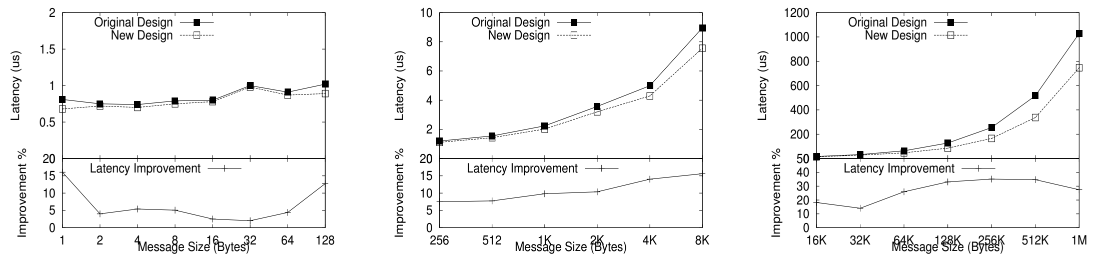
Experimental Setup: The NUMA cluster is composed of two nodes. Each node is equipped with quad AMD Opteron Processor (single core) running at 2.0 GHz. Each processor has a 1024 KB L2 cache. The two nodes are connected by InfiniBand. We refer to this cluster as cluster A in the following sections. The dual core NUMA cluster, referred to as cluster B, also has two nodes connected by InfiniBand. Each node is equipped with four Dual Core AMD Opteron Processor (two cores on the same chip and two chips in total). The processor speed is 2.0

GHz, and the L2 cache size is 1024 KB per core. The operating system on the two clusters is Linux 2.6.16. The MVAPICH version used is 0.9.7.

We compare the performance of our design to the design in MVAPICH. In the following sections, we refer to the basic shared memory based design as the *Original Design*, and the advanced design as the *New Design*. Latency is measured in unit of *micro second (us)*, and bandwidth is measured in *million bytes per second (MB/sec)*.

3.3.1 Latency and Bandwidth on NUMA Cluster

In this section we evaluate the basic ping pong latency and uni-directional bandwidth on one node in cluster A. From Figure 3.6 we can see that the new design improves the latency of small and medium messages by up to 15%, and improves the large message latency by up to 35%. The bandwidth is improved by up to 50% as shown in Figure 3.7. The peak bandwidth is raised from 1200 MB/sec to 1650 MB/sec.



(a) Small Messages (b) Medium Messages (c) Large Messages

Figure 3.6: Latency on NUMA Cluster

3.3.2 L2 Cache Miss Rate

To further analyze the reason of the performance gain presented in Section 3.3.1, we measured the L2 cache miss rate while running the latency and bandwidth benchmarks. The tool used to measure the cache miss rate is *Valgrind* [2], and the benchmarks are the same as used in Section 3.3.1. The results are shown in Figure 3.8. The results indicate that a large portion of the performance gain comes from the efficient use of the L2 cache by the new design. This conforms well to our theoretical analysis of the new design discussed in Section 3.2.3.

3.3.3 Impact on MPI Collective Functions

MPI collective functions are frequently used in MPI applications, and their performance is critical to many of the applications. Since MPI collective functions can be implemented on top of point-to-point based algorithms, in this section we study the impact of the new design on MPI collective calls. The experiments were conducted on cluster A.

Figure 3.9 shows the performance of *MPI_Barrier*, which is one of the most frequently used MPI collective functions. We can see from the figure that the new design improves *MPI_Barrier* performance by 17% and 19% on 2 and 4 processes respectively, and the improvement is 8% on 8 processes. The drop of performance improvement on 8 processes is caused by the mixture of intra- and inter-node communication that takes place within the two separate nodes in cluster A. Therefore, only a fraction of the overall performance can be enhanced by the intra-node communication.

Figure 6.9(a) presents the performance of another important collective call *MPI_Alltoall* on one node with 4 processes on cluster A. In *MPI_Alltoall* every process does a personalized send to every other process. This figure shows that the performance can be improved by up to 10% for small and medium messages and 25% for large messages.

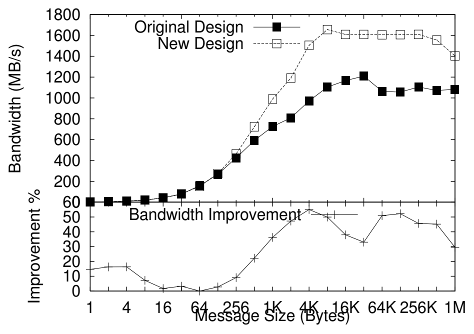


Figure 3.7: Bandwidth on NUMA Cluster

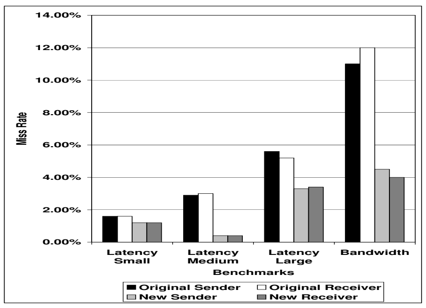


Figure 3.8: L2 Cache Miss Rate

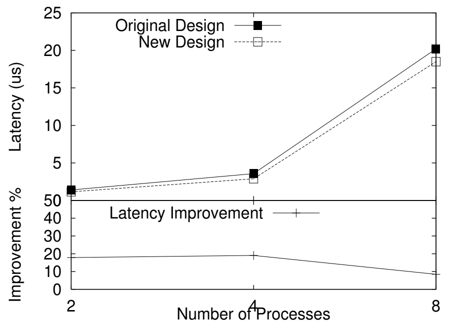


Figure 3.9: MPI_Barrier Performance

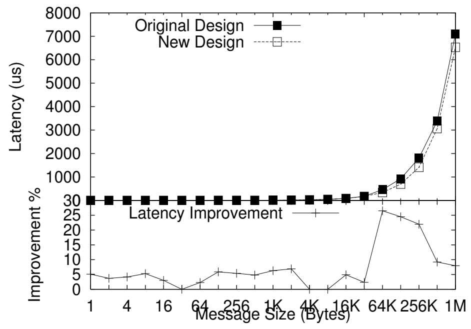
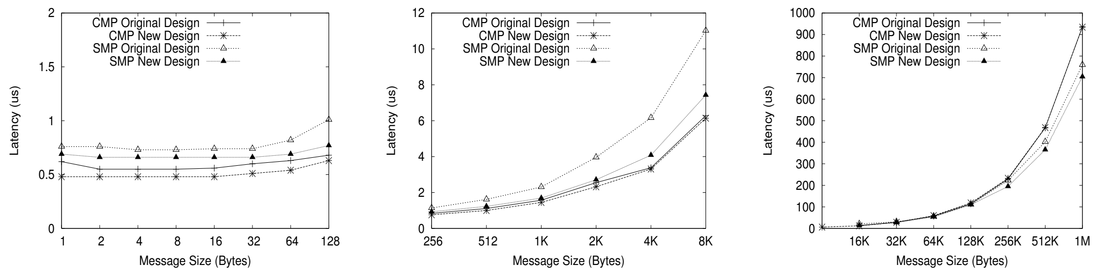


Figure 3.10: MPI_Alltoall Performance



(a) Small Messages (b) Medium Messages (c) Large Messages

Figure 3.11: Latency on Dual Core NUMA Cluster

3.3.4 Latency and Bandwidth on Dual Core NUMA Cluster

Multi-core processor is an emerging new processor architecture that few study has been done with respect to how it interacts with MPI implementations. Our initial research on such topic is presented next, and we plan to do more in-depth analysis in the future. The experiments were carried out on cluster B.

Figure 3.11 demonstrates the latency of small, medium, and large messages respectively. *CMP* stands for *Chip-level MultiProcessing*, which we use to represent the communication between two processors (cores) on the same chip. We refer to communication between two processors on different chips as *SMP (Symmetric MultiProcessing)*. From Figure 3.11 we notice that CMP has a lower latency for small and medium messages than SMP. This is because when the message is small enough to be resident in the cache, the processors do not need to access the main memory, thus only cache-to-cache transfer is needed. Cache-to-cache transfer is much faster if two processors are on the same chip. However, when the message is large and the processors need to access the main memory to get the data, CMP has a higher latency because the two processors on the same chip will have contention for memory. Figure 3.11 also shows that the new design improves the SMP latency for all message sizes. It also improves CMP latency for small and medium messages, but not for large messages. Further investigation is needed to fully understand the reason.

The bandwidth results, shown in Figure 3.12, indicate the same trend. Again, the new design improves SMP bandwidth for all message sizes, and CMP bandwidth for small and medium messages.

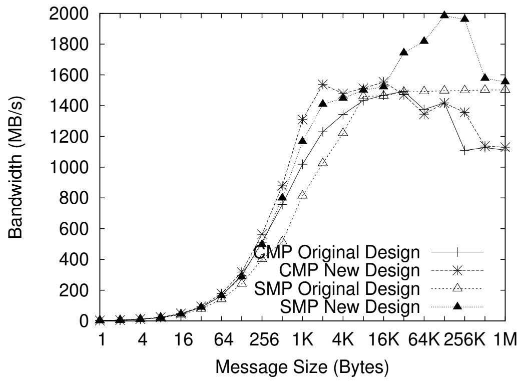


Figure 3.12: Bandwidth on Dual Core NUMA Cluster

3.3.5 Application Performance on Intel Clovertown Cluster

In this section we show the application level performance of the advanced shared memory based design.

Experimental Setup: We used a four-node cluster, each node is equipped with dual Intel Clovertown (quad-core) processor, that is 8 cores per node. The processor speed is 2.33GHz. A Clovertown chip is made of two Woodcrest chips, which means two cores share a 4MB L2 cache.

The benchmarks we used include IS from NAS parallel benchmarks and PSTSWM which is a shallow water modeling application. The results are shown in Figure 3.13, from which we can see that the advanced shared memory based design improves application performance by up to 5%. This is mainly due to the efficient cache utilization of the new design.

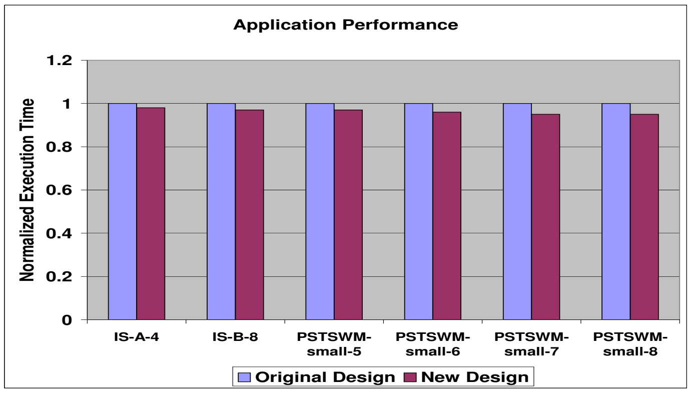


Figure 3.13: Application Performance Comparison

3.4 Summary

In this chapter, we have designed and implemented shared memory based schemes for MPI intra-node communication. We start with designing a basic approach and its optimizations. Then we propose an advanced approach which uses the system cache efficiently, requires no locking mechanisms, and has low memory usage. The advanced approach shows both high performance and good scalability. Our experimental results show that the advanced design can improve MPI intra-node latency by up to 35% compared to the basic design on single core NUMA systems, and improve bandwidth by up to 50%. The improvement in point-to-point communication also reduces MPI collective call latency - up to 19% for *MPI_Barrier* and 25% for *MPI_Alltoall*. We have done study on the interaction between multi-core systems and MPI. From the experimental results we see that the advanced design can also improve intra-node communication performance for multi-core systems. For MPI applications, the advanced approach improves performance by up to 5%.

CHAPTER 4

CPU BASED KERNEL ASSISTED DIRECT COPY

The shared memory approach described in Chapter 3 provides high performance, but the performance is not optimal mainly due to several message copies involved. Every process has its own virtual address space and cannot directly access another process's message buffer. One approach to avoid extra message copies is to use operating system kernel to provide a direct copy.

In this chapter, we propose, design and implement a portable approach to intra-node message passing at the kernel level. To achieve this goal, we design and implement a Linux kernel module that provides MPI friendly interfaces. This module is independent of any communication library or interconnection network. It also offers portability across the Linux kernels. We call this kernel module as LiMIC (**Li**nux kernel module for **M**PI **I**ntra-node **C**ommunication). We have implemented two versions of LiMIC. The second generation is referred to as LiMIC2. The main difference between LiMIC and LiMIC2 is the interface exposed to the MPI libraries.

The rest of the section is organized as the follows: In Section 4.1 we describe the existing kernel based approach, its limitations, and our approach. We present the detailed design and implementation issues in Section 4.2. The evaluation results and analysis are presented in Section 4.3. Finally we summarize the results and impact of this work in Section 4.4.

4.1 Limitations of the Existing Approach and Overall Design of LiMIC

In this section, we describe the existing kernel based solution and its limitations. We then propose our approach: LiMIC.

4.1.1 Kernel-Based Memory Mapping

Kernel-based memory mapping approach takes help from the operating system kernel to copy messages directly from one user process to another without any additional copy operation. The sender or the receiver process posts the message request descriptor in a message queue indicating its virtual address, tag, etc. This memory is mapped into the kernel address space when the other process arrives at the message exchange point. Then the kernel performs a direct copy from the sender buffer to the receiver application buffer. Thus this approach involves only one copy.

Figure 1.2(c) demonstrates the memory transactions needed for copying from the sender buffer directly to the receiver buffer. In step 1, the receiving process needs to bring the sending process' buffer into cache. Then in step 3, the receiving process can write this buffer into its own receive buffer. This may generate step 2 based on whether the buffer was in cache already or not. Then, depending on the cache replacement policy, step 4 might be generated implicitly.

It is to be noted that the number of possible memory transactions for the Kernel-based memory mapping is always less than the number in User-space shared memory approach. We also note that due to the reduced number of copies to and from various buffers, we can maximize the cache utilization. However, there are other overheads. The overheads include time to trap into the kernel, memory mapping overhead, and TLB flush time. In addition, still the CPU resource is required to perform a copy operation.

There are several previous works that adopt this approach, which include [43, 72]. However, their designs lack portability across different networks and deny flexibility to the MPI library developer. To the best of our knowledge, no other current generation open source MPI implementations provide such a kernel support. SGI MPT (Message Passing Toolkit) provides a single copy support, but it depends on XPMEM which is an SGI proprietary driver [69].

4.1.2 Our Approach: LiMIC

It is to be noted that the kernel-based approach has the potential to provide efficient MPI intra-node communication. In this chapter we are taking this approach, providing unique features such as portability across various interconnects and different communication libraries. This section sharply distinguishes our approach and design philosophy from earlier research in this direction. Our design principles and details of this approach are described in Section 4.2.

Traditionally, researchers have explored kernel based approaches as an extension to the features available in user-level protocols. A high level description of these earlier methodologies is shown in Figure 4.1(a). As a result, most of these methodologies have been non-portable to other user-level protocols or other MPI implementations. In addition, these earlier designs do not take into account MPI message matching semantics and message queues. Further, the MPI library blindly calls routines provided by the user-level communication library. Since some of the communication libraries are proprietary, this mechanism denies any sort of optimization-space for the MPI library developer.

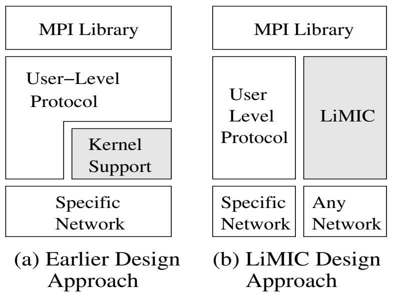


Figure 4.1: Approaches for Kernel-Based Design

In order to avoid the limitations of the past approaches we look towards generalizing the kernel-access interface and making it MPI friendly. Our implementation of this interface is called LiMIC (**Li**nux kernel module for **M**PI **I**ntra-node **C**ommunication). Its high level diagram is shown in Figure 4.1(b). We note that such a design is readily portable across different interconnects because its interface and data structures are not required to be dependent on a specific user-level protocol or interconnect. Also, this design gives the flexibility to the MPI library developer to optimize various schemes to make appropriate use of the one copy kernel mechanism. For instance, LiMIC provides flexibility to the MPI library developer to easily choose thresholds for the hybrid approach with other intra-node communication mechanisms and tune the library for specific applications. Such flexibility is discussed in [31]. As a result, LiMIC can provide portability on different interconnects and flexibility for MPI performance optimization.

4.2 Design and Implementation Issues

In this section, we discuss the detailed design issues of LiMIC and its integration with MPI.

4.2.1 Portable and MPI Friendly Interface

In order to achieve portability across various Linux systems, we design LiMIC to be a runtime loadable module. This means that no modifications to the kernel code is necessary. Kernel modules are usually portable across major versions of mainstream Linux. The LiMIC kernel module can be either an independent module with device driver of interconnection network or a part of the device driver. In addition, the interface is designed to avoid using communication library specific or MPI implementation specific information.

In order to utilize the interface functions, very little modification to the MPI layer are needed. These are required just to place the hooks of the send, receive and completion of messages. The LiMIC interface traps into the kernel internally by using the `ioctl()` system call. We briefly describe the major interface functions provided by LiMIC.

- `LiMIC_Isend(int dest, int tag, int context_id, void* buf, int len, MPI_Request* req)`: This call issues a non blocking send to a specified destination with appropriate message tags.

- `LiMIC_Irecv(int src, int tag, int context_id, void* buf, int len, MPI_Request* req)`: This call issues a non-blocking receive. It is to be noted that blocking send and receive can be easily implemented over non-blocking and wait primitives.

- `LiMIC_Wait(int src/dest, MPI_Request* req)`: This call just polls the LiMIC completion queue once for incoming sends/receives.

As described in Section 4.1.2, we can observe that the interface provided by LiMIC does not include any specific information on a user-level protocol or interconnect. The interface only defines the MPI related information and has an MPI standard similar format.

4.2.2 Memory Mapping Mechanism

To achieve one-copy intra-node message passing, a process should be able to access the other processes' virtual address space so that the process can copy the message to/from the other's address space directly. This can be achieved by memory mapping mechanism that maps a part of the other processes' address space into its own address space. After the memory mapping the process can access mapped area as its own.

For memory mapping, we use `kiobuf` provided by the Linux kernel. The `kiobuf` structure supports the abstraction that hides the complexity of the virtual memory system from device drivers. The `kiobuf` structure consists of several fields that store user buffer information such as page descriptors corresponding to the user buffer, offset to valid data inside the first page, and total length of the buffer. The Linux kernel exposes functions to allocate `kiobuf` structures and make a mapping between `kiobuf` and page descriptors of user buffer. In addition, since `kiobuf` internally takes care of pinning down the memory area, we can easily guarantee that the user buffer is present in the physical memory when another process tries to access it. Therefore, we can take advantage of `kiobuf` as a simple and safe way of memory mapping and page locking.

Although the `kiobuf` provides many features, there are several issues we must address in our implementation. The `kiobuf` functions provide a way to map between `kiobuf` and page descriptors of target user buffer only. Therefore, we still need to map the physical memory into the address space of the process, which wants to access the target buffer. To do so, we use the `kmap()` kernel function. Another issue is a large allocation overhead of `kiobuf` structures. We performed tests on `kiobuf` allocation time on our cluster (Cluster A in Section 4.3) and found that it takes around 60μs to allocate one `kiobuf`. To remove this overhead from the critical path, LiMIC kernel module preallocates some amount of `kiobuf` structures during the module loading phase and manages this `kiobuf` pool.

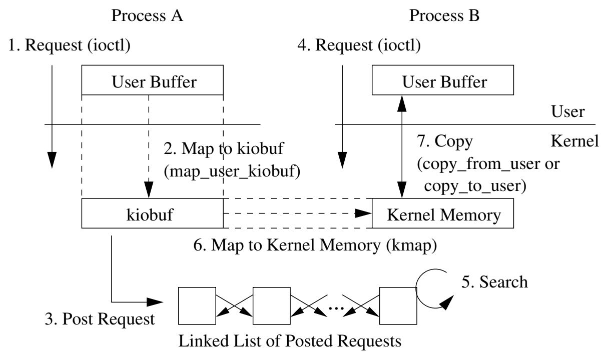


Figure 4.2: Memory Mapping Mechanism

Figure 4.2 shows the internal memory mapping operation performed by LiMIC. When either of the message exchanging processes arrives, it issues a request through `ioctl()` (Step 1). If there is no posted request that can be matched with the issued request, the kernel module simply saves information of page descriptors for the user buffer and pins down it by calling `map_user_kiobuf()` (Step 2). Then, the kernel module puts this request into the request queue (Step 3). After that when the other message partner issues a request (Step 4), the kernel module finds the posted request (Step 5) and maps the user buffer to the kernel memory by calling `kmap()` (Step 6). Finally, if the process is the receiver, the kernel module copies the data from kernel memory to user buffer using `copy_to_user()`, otherwise the data is copied from user buffer to kernel memory by `copy_from_user()` (Step 7). The data structures in the kernel module are shared between different instances of the kernel executing on the sending and receiving processes. To guarantee consistency, LiMIC takes care of locking the shared data structures.

4.2.3 Copy Mechanism

Since the copy needs CPU resources and needs to access pinned memory, we have to carefully decide the timing of the message copy. The message copy could be done in either of the three ways: copy on function calls of receiver, copy on wait function call, and copy on send and receive calls.

We suggest the design where the copy operation is performed by send and receive functions (i.e., `LiMIC_Isend` and `LiMIC_Irecv`) so that we can provide better progress and less resource usage. In addition, this approach is not prone to skew between processes. The actual copy operation is performed by the process which arrives later at the communication call. So, regardless of the sender or receiver, the operation can be completed as soon as both the processes have arrived. In addition, only the first process is required to pin down the user buffer.

4.2.4 MPI Message Matching

There are separate message queues for messages sent or received through the kernel module. This is done to allow portability to various other MPI like message queues. So, in general the LiMIC does not assume any specific message queue structure. MPI messages are matched based on *Source*, *Tag* and *Context ID*. Message matching can also be done by using wild cards like `MPI_ANY_SOURCE` or `MPI_ANY_TAG`. LiMIC implements MPI message matching in the following manner:

- **Source in the same node**: In this case, the receive request is directly posted into the queue maintained by LiMIC. On the arrival of the message, the kernel instance at the receiver side matches the message based on the source, tag and context id information and then it passes the buffer into user space.

- **Source in a different node**: In this case, LiMIC is no longer responsible for matching the message. The interface hooks provided in the MPI should take care of not posting the receive request into the kernel message queue.

- **Source in the same node and `MPI_ANY_TAG`**: As in the first case, the receive request is not posted in the generic MPI message queue, but directly into the LiMIC message queue. Now, the matching is done only by the source and context id.

- `MPI_ANY_SOURCE` **and** `MPI_ANY_TAG`: In this case, the source of the message might be on the same physical node but also it can be some other node which is communicating via the network. So the receive request is posted in the MPI queue. Then the MPI internal function that senses an arrival of message checks the send queue in the kernel module as well by using a LiMIC interface, `LiMIC_Iprobe`, and performs message matching with requests in the MPI queue. If the function finds a message which matches the request, the function performs the receive operation by calling the LiMIC receive interface.

Some specialized MPI implementations offload several MPI functions into the NIC. For example, Quadrics performs MPI message matching at the NIC-level [64]. The LiMIC might need an extended interface for such MPI implementations while most of MPI implementations can easily employ LiMIC.

4.3 Performance Evaluation

In this section we evaluate various performance characteristics of LiMIC and LiMIC2 on different platforms. We also present the performance impact on MPI+OpenMP model.

4.3.1 Performance Evaluation of LiMIC on a Single-core Cluster

As described in section 1.2, there are various design alternatives to implement efficient intra-node message passing. MVAPICH [15] version 0.9.4 implements a hybrid mechanism of User-space shared memory and NIC-level loopback. The message size threshold used by MVAPICH-0.9.4 to switch from User-space shared memory to NIC-level loopback is 256KB. In this section, we use a hybrid approach for LiMIC, in which User-space shared memory is used for short messages (up to 4KB) and then Kernel-based memory mapping is used to perform an one copy transfer for larger messages. The choice of this threshold is explained below in section 4.3.1. However, each application can set a different threshold. Here on, all references to MVAPICH-0.9.4 and LiMIC refer to the hybrid designs mentioned above. In addition, we also provide performance results for each of the individual design alternatives, namely, User-space shared memory, NIC loopback, and Kernel module.

We conducted experiments on two 8-node clusters with the following configurations:

- **Cluster A:** SuperMicro SUPER X5DL8-GG nodes with dual Intel Xeon 3.0 GHz processors, 512 KB L2 cache, PCI-X 64-bit 133 MHz bus
- **Cluster B:** SuperMicro SUPER P4DL6 nodes with dual Intel Xeon 2.4 GHz processors, 512 KB L2 cache, PCI-X 64-bit 133 MHz bus

The Linux kernel version used was 2.4.22smp from kernel.org. All the nodes are equipped with Mellanox InfiniHost MT23108 HCAs. The nodes are connected using Mellanox MTS 2400 24-port switch. Test configurations are named (2x1), (2x2), etc. to denote two processes on one node, four processes on two nodes, and so on.

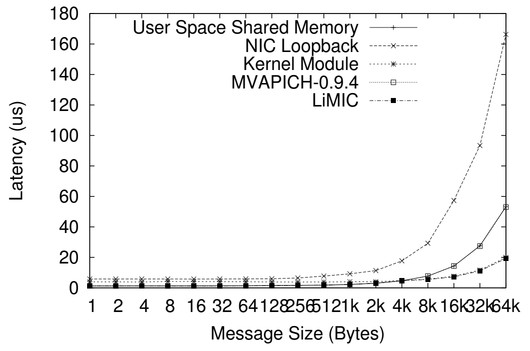
First, we evaluate our designs at microbenchmarks level. Second, we present experimental results on message transfer and descriptor post breakdown. Then we evaluate the scalability of performance offered by LiMIC for larger clusters. Finally, we evaluate the impact of LiMIC on NAS Integer Sort application kernel.

Microbenchmarks

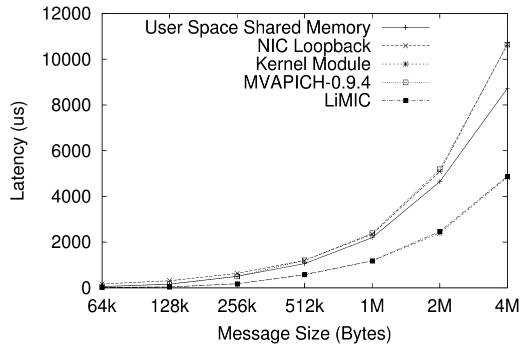
In this section, we describe our tests for microbenchmarks such as point-to-point latency and bandwidth. The tests were conducted on Cluster A.

The latency test is carried out in a standard ping-pong fashion. The latency microbenchmark is available from [15]. The results for one-way latency is shown in Figures 4.3(a) and 4.3(b). We observe an improvement of 71% for latency as compared to MVAPICH-0.9.4 for 64KB message size. The results clearly show that on this experimental platform, it is most expensive to use NIC-level loopback for large messages. The User-space shared memory implementation is good for small messages. This avoids extra overheads of polling the network or trapping into the kernel. However, as the message size increases, the application buffers and the intermediate shared memory buffer no longer fit into the cache and the copy overhead increases. The Kernel module on the other hand can reduce one copy, hence maximizing the cache effect. As can be noted from the latency figure, after the message size of 4KB, it becomes more beneficial to use the Kernel module than User-space shared memory. Therefore, LiMIC hybrid uses User-space shared memory for messages smaller than 4KB and the Kernel module for larger messages.

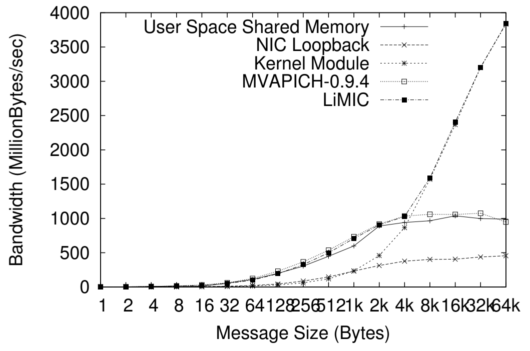
For measuring the point-to-point bandwidth, a simple window based communication approach was used. The bandwidth microbenchmark is available from [15]. The bandwidth graphs are shown in Figures 4.3(c) and 4.3(d). We observe an improvement of 405% for bandwidth for 64KB message size as compared to MVAPICH-0.9.4. We also observe that the bandwidth offered by LiMIC drops at 256KB message size. This is due to the fact that the cache size on the nodes in Cluster A is 512KB. Both sender and receiver buffers and some additional data cannot fit into the cache beyond this message size. However, the bandwidth offered by LiMIC is still greater than MVAPICH-0.9.4.



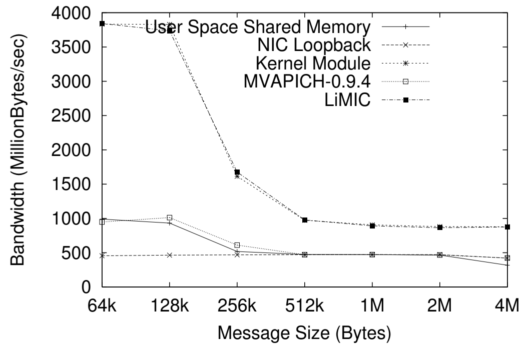
(a) Small Message Latency



(b) Large Message Latency



(c) Small Message Bandwidth

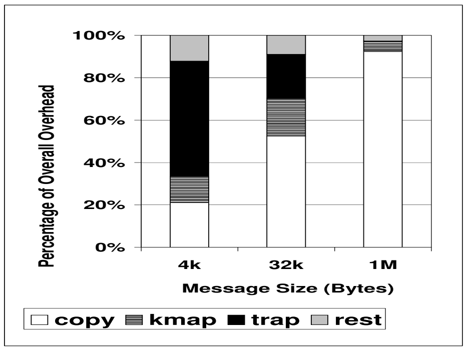


(d) Large Message Bandwidth

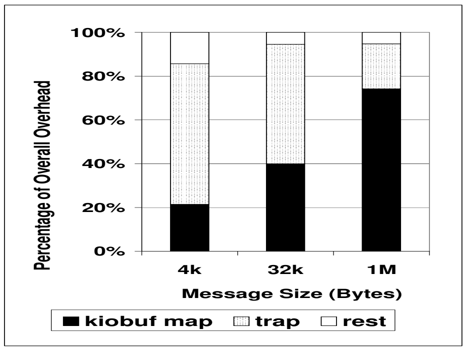
Figure 4.3: MPI Level Latency and Bandwidth

LiMIC Cost Breakdown

In order to evaluate the cost of various operations which LiMIC has to perform for message transfer, we profiled the time spent by LiMIC during a ping-pong latency



(a) Message Transfer Breakdown



(b) Descriptor Post Breakdown

Figure 4.4: LiMIC Cost Breakdown (Percentage of Overall Overhead)

test. In this section, we present results on the various relative cost breakdowns on Cluster A.

The overhead breakdown for message transfer in percentages is shown in Figure 4.4(a). We observe that the message copy time dominates the overall send/receive operation as the message size increases. For shorter messages, we see that a considerable amount of time is spent in the kernel trap (around 3μs) and around 0.5μs in queueing and locking overheads (indicated as "rest"), which are shown as 55% and 12% of the overall message transfer overhead for 4KB message in Figure 4.4(a). We also observe that the time to map the user buffer to the kernel address space (using `kmap()`) increases as the number of pages in the user buffer increases.

The overhead breakdown for descriptor posting in percentages is shown in Figure 4.4(b). We observe that the time to map the `kiobuf` with the page descriptors of the user buffer forms a large portion of the time to post a descriptor. It is because the `kiobuf` mapping overhead increases in proportional to the number of pages. This step also involves the pinning of the user buffer into physical memory. The column labeled “rest” indicates again the queuing and locking overheads.

HPCC Effective Bandwidth

To evaluate the impact of the improvement of intra-node bandwidth on a larger cluster of dual SMP systems, we conducted effective bandwidth test on Clusters A and B. For measuring the effective bandwidth of the clusters, we used b_eff [66] benchmark. This benchmark measures the accumulated bandwidth of the communication network of parallel and distributed computing systems. This benchmark is featured in the High Performance Computing Challenge benchmark suite (HPCC) [47].

Table 4.1 shows the performance results of LiMIC compared with MVAPICH-0.9.4. It is observed that when both processes are on the same physical node (2x1), LiMIC improves effective bandwidth by 61% on Cluster A. It is also observed that even for a 16 process experiment (2x8) the cluster can achieve 12% improved bandwidth.

The table also shows the performance results on Cluster B. The results follow the same trend as that of Cluster A. It is to be noted that the message latency on User-space shared memory and Kernel module depends on the speed of CPU while the NIC-level loopback message latency depends on the speed of I/O bus. Since the I/O bus speed remains the same between Clusters A and B, and only the CPU speed reduces, the improvement offered by LiMIC reduces in Cluster B.

In our next experiment, we increased the number of processes as to include nodes in both Clusters A and B. The motivation was to see the scaling of the improvement in effective bandwidth as the number of processes is increased. It is to be noted that the improvement percentage remains constant (5%) as the number of processes is increased.

Table 4.1: b_eff Results Comparisons (MB/s)

Cluster	Config.	MVAPICH	LiMIC	Improv.
A	2x1	152	244	61%
	2x2	317	378	19%
	2x4	619	694	12%
	2x8	1222	1373	12%
B	2x1	139	183	31%
	2x2	282	308	9%
	2x4	545	572	5%
	2x8	1052	1108	5%
A & B	2x16	2114	2223	5%

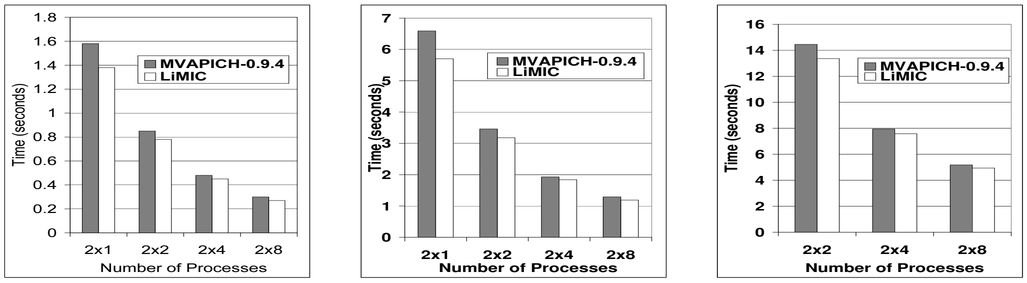


Figure 4.5: IS Total Execution Time Comparisons: (a) Class A, (b) Class B, and (c) Class C

NAS Integer Sort

We conducted performance evaluation of LiMIC on IS in NAS Parallel Benchmark suite [38] on Cluster A. IS is an integer sort benchmark kernel that stresses the communication aspect of the network. We conducted experiments with classes A, B and C on configurations (2x1), (2x2), (2x4), and (2x8). The results are shown in Figure 4.5. Since the class C is a large problem size, we could run it on the system sizes larger than (2x2). We can observe that LiMIC can achieve 10%, 8%, and 5% improvement of execution time running classes A, B, and C respectively, on (2x8) configuration. The improvements are shown in Figure 4.6.

To understand the insights behind the performance improvement, we profiled the number of intra-node messages larger than 1KB and their sizes being used by IS within a node. The results with class A are shown in Table 4.2. We can see that as the system size increases, the size of the messages reduces. The trend is the same on classes B and C while the message size becomes larger than class A. Since LiMIC performs better for medium and larger message sizes, we see overall less impact of LiMIC on IS performance as the system size increases. Also, it is to be noted that since the message size reduces as the system size increases, the message size eventually fits in the cache size on (2x8) configuration. This results in maximizing the benefit of LiMIC and raising the improvement at the (2x8) system size as shown in Figure 4.6.

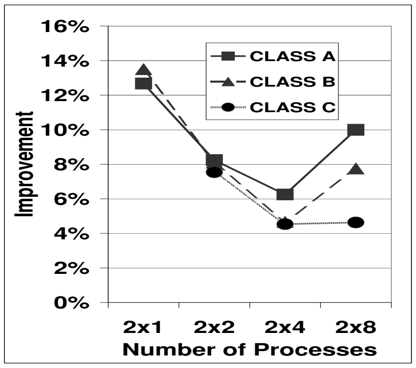


Figure 4.6: IS Performance Improvement

Table 4.2: Intra-Node Message Size Distribution for IS Class A

Message Size (Bytes)	2x1	2x2	2x4	2x8
1K-8K	44	44	44	44
32K-256K	0	0	0	22
256K-1M	0	0	22	0
1M-4M	0	22	0	0
4M-16M	22	0	0	0

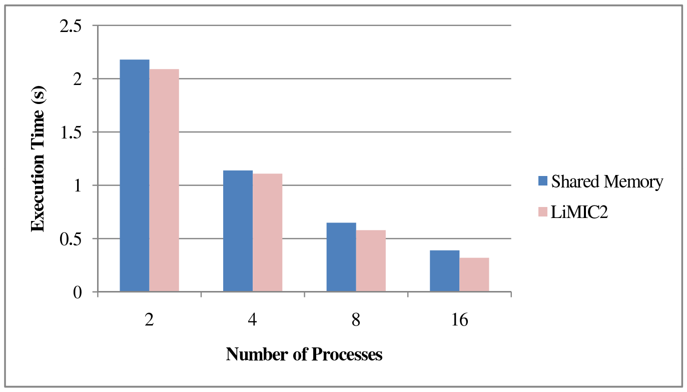


Figure 4.7: Application Performance of LiMIC2 on an AMD Barcelona System

4.3.2 Application Performance of LiMIC2 on an AMD Barcelona System

In this section, we evaluate the performance of LiMIC2 on an AMD Barcelona system using IS class A in NAS, and compare with the shared memory approach. The results are shown in Figure 4.7. The system has four quad-core Opteron chips (16 cores on a node) running at 2GHz. Each core has a 512KB L2 cache. The operating system is Linux 2.6.18. From Figure 4.7 we can see that LiMIC2 improves IS performance by up to 18%.

4.3.3 Performance Impact on MPI+OpenMP Model

MPI+OpenMP [46] model explores two levels of parallelism. It uses OpenMP [39] for multiprocessing within a node and MPI for communication across nodes. MPI+OpenMP was proposed because the communication overhead in MPI was high and it was more efficient to use OpenMP, essentially the threads and shared memory model, within a node. Our work on MPI intra-node communication has largely reduced the communication overhead and it is interesting to re-examine the relative performance of pure MPI versus MPI+OpenMP. In this section, we evaluate the performance of these two models using LU-MZ and SP-MZ [35], the multi-zone version of LU and SP in NAS benchmarks, which are implemented with MPI+OpenMP. The results are shown in Figure 4.8.

In this experiment, we use two Intel Clovertown systems. Each node has two quad-core Intel Clovertown processors and two nodes are connected by InfiniBand. Each socket has two chips and two cores on the same chip share a 4MB L2 cache. In the legend, 2x8 means there are 2 processes, each running on one node with 8 OpenMP threads, which is the traditional MPI+OpenMP model. 16x1 means there are 16 MPI processes and each process only has one thread, which is essentially the pure MPI model. Similarly, 4x4 means 4 processes with 4 threads per process and 8x2 means 8 processes with 2 threads per process. It is to be noted that in the 4x4 mode, each MPI process runs a socket, and in the 8x2 mode, each MPI process runs on a chip. We have two observations from Figure 4.8. First, if we compare the performance of the traditional MPI+OpenMP with pure MPI, i.e. compare 2x8 with 16x1, we can see that they perform almost the same, actually pure MPI is even slightly better. Second, we find that 4x4 and 8x2 perform better than both 2x8 and 16x1. These indicate that with efficient MPI intra-node communication, pure MPI can perform as well as the traditional OpenMP+MPI model for some applications and OpenMP+MPI needs to change to smaller granularity for better performance. When OpenMP+MPI uses socket or chip granularity, the improvement on MPI intra-node communication performance will benefit the OpenMP+MPI model. The

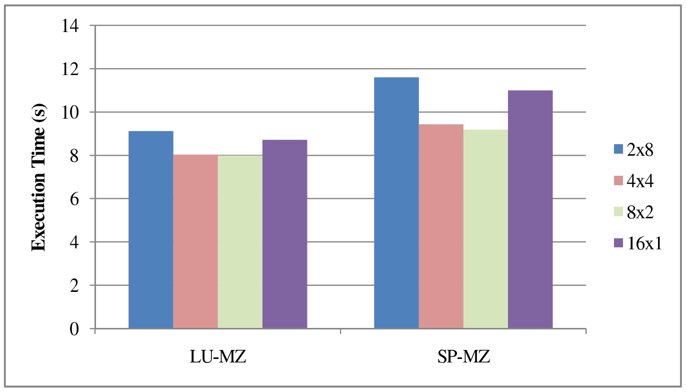


Figure 4.8: Performance Impact on MPI+OpenMP Model

relative performance of MPI and MPI+OpenMP also depends on application patterns and problem sizes and will need to be thoroughly studied in the future.

4.4 Summary

In this chapter we have designed and implemented a high performance Linux kernel module (called LiMIC) for MPI intra-node message passing. LiMIC is able to provide MPI friendly interface and independence from proprietary communication libraries and interconnects.

To measure the performance of LiMIC, we have integrated it with MVAPICH. Through the benchmark results, we could observe that LiMIC improved the point-to-point latency and bandwidth up to 71% and 405%, respectively. In addition, we observed that employing LiMIC in an 8-node InfiniBand cluster, increased the HPCC effective bandwidth by 12%. Also, our experiments on a larger 16-node cluster revealed that the improvement in HPCC effective bandwidth remains constant as the number of processes increased. Further, LiMIC improved the NAS IS benchmark execution time by 10%, 8%, and 5% for classes A, B, and C respectively, on an 8-node cluster. Similarly, we observe that LiMIC2 has improved IS performance on an AMD Barcelona system by up to 18%. We have also conducted preliminary study on the MPI+OpenMP model and find that MPI+OpenMP can also benefit from our work.

CHAPTER 5

DMA BASED KERNEL ASSISTED DIRECT COPY

Direct Memory Access (DMA) has been traditionally used to transfer the data directly from the host memory to any input/output device without the host CPU intervention. Networks such as InfiniBand [6] provide a zero-copy data transfer support. However, such solutions are mainly used for transferring data from one node to another [54]. Researchers in the past have attempted to use DMA engines to accelerate bulk data movement within a node [33]. Many of these approaches have not entirely succeeded due to huge DMA startup costs, completion notification costs and other performance-related issues. Recently, Intel's I/O Acceleration Technology (I/OAT) [44, 57, 68] introduced an asynchronous DMA copy engine within the chip that has direct access to main memory to improve performance and reduce the overheads mentioned above. In this chapter, we present our DMA based kernel assisted direct copy approach for MPI intra-node communication.

The rest of the chapter is organized as the follows: We introduce three schemes we have designed for IPC in Section 5.1 and describe the integration of these scheme in MPI in Section 5.2. We present the MPI level performance evaluation in Section 5.3 and finally summarize in Section 5.4.

5.1 Design of the DMA Based Schemes

We have designed three schemes, namely SCI, MCI, and MCNI. In this section we describe the detailed design of these schemes.

5.1.1 SCI (Single-Core with I/OAT)

The SCI scheme *offloads* the memory copy operation to the I/OAT's hardware copy engine and uses the kernel module to expose the features of the hardware copy engine to user applications in order to perform asynchronous memory copy operations. We have extended the support of asynchronous memory copy operations for both single process as an *offloaded memcpy* and IPC. User applications contact the kernel module (referred to as memory copy module in Figure 5.1(b)) for offloading the copy operation. The kernel module takes help from the underlying DMA module in initiating the memory copy operation across each of the DMA channels. On a completion notification request, the kernel module checks the progress of memory copy operation and informs the application accordingly. In addition, tasks such as pinning the application buffers, posting the descriptors, releasing the buffers are also handled by the kernel module. The SCI scheme also supports page caching mechanism to avoid pinning of application buffers while performing memory copy operations. In this mechanism, the kernel module caches the virtual to physical page mappings after locking the application buffers. Once the memory copy operation finishes, the kernel module does not unlock the application buffers in order to avoid the pinning cost if the same application buffer is reused for another memory copy operation.

For single process operations, we provide *memcpy* like interfaces as shown in Table 5.1. And for IPC, we provide *socket* like interfaces which are illustrated later in Table 5.2 in Section 5.2.

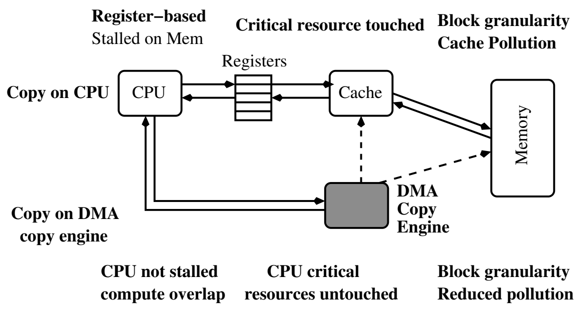
Table 5.1: Basic Interfaces for Using I/OAT Copy Engine

Operation	Description
ioat_copy(src, dst, len)	Blocking copy routine
ioat_icopy(src, dst, len)	Non-blocking copy routine
ioat_check_copy(cookie)	(Non-blocking) check for completion
ioat_wait_copy(cookie)	(Blocking) wait for completion

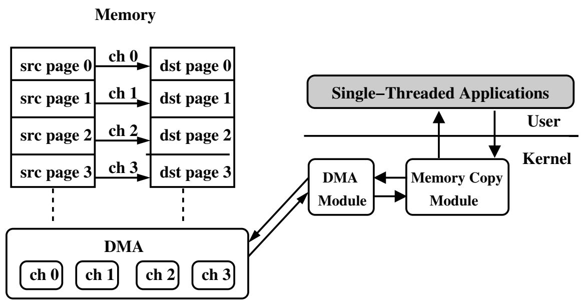
5.1.2 MCI (Multi-Core with I/OAT)

While the SCI scheme helps user applications to *offload* memory copy operations, several critical operations still remain in the critical path, causing overheads such as copy engine initiation overheads, page locking overheads, context switch overheads, synchronization overheads, etc. In this section, we describe the MCI scheme which is designed to alleviate these overheads to achieve maximum overlap between memory copy operation and computation.

The main idea of MCI scheme is to *offload* the copy operation to the hardware copy engine and *onload* the tasks that fall in the critical path to another core or a processor so that applications can exploit complete overlap of memory operation with computation.



(a) Copy Execution on CPU vs Copy Engines(Courtesy [74])



(b) SCI Scheme

Figure 5.1: Copy Engine Architecture and SCI Scheme

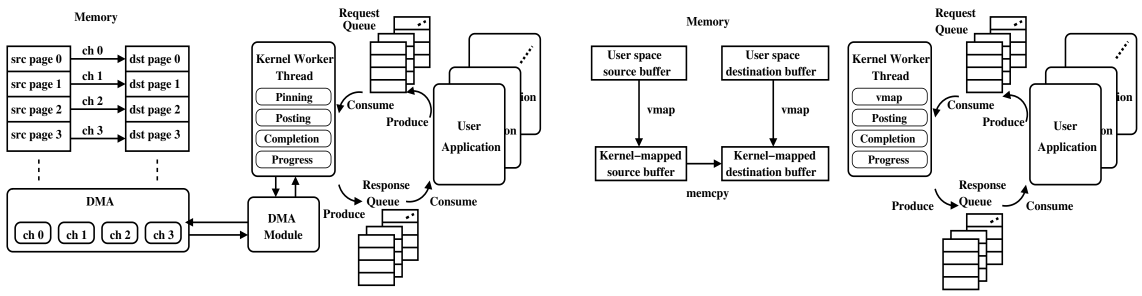


Figure 5.2: Asynchronous Memory Copy Operations: (a) MCI Scheme and (b) MCNI Scheme

Figure 5.2a shows the various components of the proposed scheme. Since the copy engine is accessible only in the kernel space, we dedicate a kernel thread to handle all copy engine related tasks and allow user applications to communicate with the kernel thread to perform the copy operation. The kernel thread also maintains a list of incomplete requests and attempts to make progress for these initiated requests. Apart from servicing multiple user applications, the dedicated kernel thread also handles tasks such as locking the application buffers, posting the descriptors for each user request on appropriate channels, checking for device completions, releasing the locked buffers after completion events. Since the critical tasks are onloaded to this kernel thread, the user application is free to execute other computation or even execute other memory copy operations while the copy operation is still in progress thus allowing almost total overlap of memory copy operation and computation.

5.1.3 MCNI (Multi-Core with No I/OAT)

In order to provide asynchronous memory copy operations for systems without the copy engine support, we have proposed a MCNI scheme (Multi-Core systems with No I/OAT) that *onloads* the memory copy operation to another processor or a core in the system. This scheme is similar to the MCI scheme described above. In this scheme, we dedicate a kernel thread to handle all memory copy operations, thus relieving the main application thread to perform computation.

5.2 Integration with MVAPICH

In this section, we describe our MPI intra-node communication implementation to take advantage of the kernel module assisted memory copy operations. Specifically we discuss how we integrate the kernel module that supports the SCI, MCI, and MCNI approaches described in Section 1.2 and 1.2 in MVAPICH.

The kernel module exposes the following user interface, as shown in Table 5.2, for applications to exchange messages across different processes. *ioat_read* and *ioat_write* operations read and write data onto another process. *ioat_iread* and *ioat_iwrite* operations initiate the data transfer.

Table 5.2: Kernel Module Interfaces for IPC

Operation	Description
ioat_iread(fd, addr, len)	Non-blocking read routine
ioat_iwrite(fd, addr, len)	Non-blocking write routine
ioat_read(fd, addr, len)	Blocking read routine
ioat_write(fd, addr, len)	Blocking write routine
ioat_check(cookie)	(Non-blocking) check for read/write completion
ioat_wait(cookie)	(Blocking) Wait for read/write completion

Because of the initiation overhead, it is only beneficial to use asynchronous memory copy operations for large messages. In our design, small messages are still transferred eagerly through the user space shared memory area. For large messages, we use the shared memory area for handshake messages, and asynchronous memory copy operations for transferring the data. The protocol is described as below:

- Step 1: The sender sends a *request_to_send* message.
- Step 2: The sender then posts its send request by initiating a non-blocking IPC write request to the kernel for performing asynchronous memory copy operations, and puts this request into a *pending_send_queue*.
- Step 3: Upon receiving the *request_to_send*, the receiver posts its receive request by initiating a non-blocking IPC read request to the kernel for performing asynchronous memory copy operations, and puts this request into a *pending_recv_queue*.
- Step 4: When the MPI program tries to make progress, the sender and the receiver check the completion of the pending operations by initiating a non-blocking IPC check request to the kernel to check for completion and inform the upper layer about the completion of the operations.

The threshold to switch from *Eager* protocol to *Rendezvous* protocol is a run time parameter which should be tuned based on the system performance.

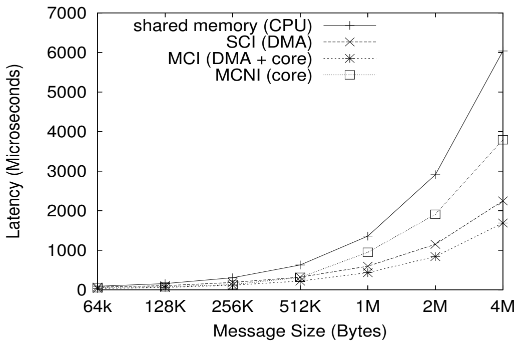
The potential benefits of using asynchronous memory copy operations for MPI intra-node communication come from several aspects. First, it reduces the number of memory copies. Second, the SCI and MCI approaches can achieve communication and computation overlap, since the memory copy is done by the DMA engine. And third, since the memory copy in the SCI and MCI approaches does not involve cache, communication buffers will not disturb the cache content.

5.3 Performance Evaluation

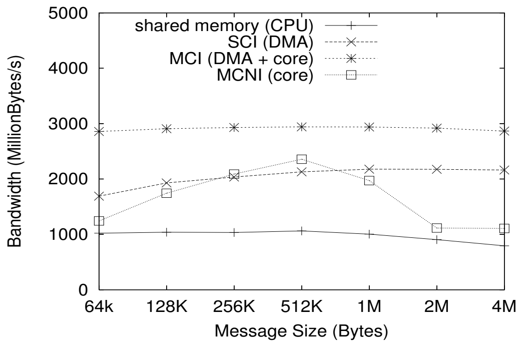
In this section we present the MPI level evaluation of kernel based approaches. We first present microbenchmark performance, followed by application level performance.

Figure 5.3 shows the MPI level intra-node latency and bandwidth. The *Rendezvous* threshold is 32KB, which means messages smaller than 32K are transferred through shared memory in all the schemes. Therefore, we only show results larger than 32KB. From Figure 5.3(a) we can see that all the kernel based asynchronous memory copy schemes are able to achieve better performance than shared memory scheme, e.g. the MCI scheme improves latency by up to 72% compared to shared memory scheme (SCNI). Among the three asynchronous memory copy schemes, the MCI scheme performs the best. The reasons are: compared with the SCI scheme, the MCI scheme onloads the operations in the critical path to another thread; and compared with the MCNI scheme, the MCI scheme uses the DMA engine which copies memory more efficiently for large blocks. The bandwidth result shown in Figure 5.3(b) reveals the same trend. Compared with the shared memory (SCNI) scheme, the MCI scheme improves bandwidth by up to 170%. It is to be noted that the bandwidth of both the shared memory scheme and the MCNI scheme drops at 2MB. This is because both of these schemes involve cache for memory operations and the L2 cache size is 2MB in our testbed. Therefore, when the message is larger than the cache size, there is an expected bandwidth drop.

We use IS in NAS parallel benchmarks [38] and PSTSWM [20] for our application level performance evaluation. The normalized execution time is shown in Figure 5.4. The results were taken on a single node. Since the MCI and the MCNI schemes



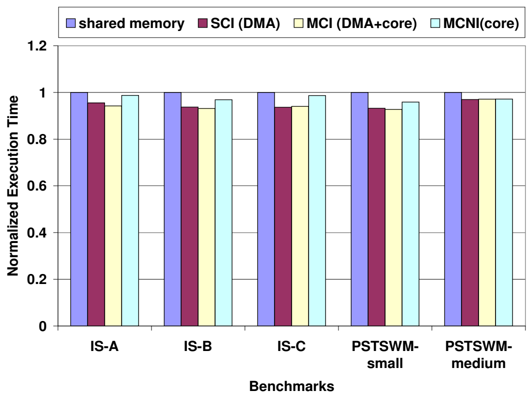
(a) Latency



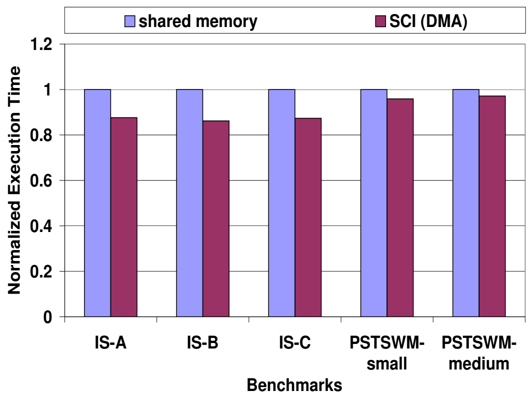
(b) Bandwidth

Figure 5.3: MPI-level Latency and Bandwidth

need an additional thread to handle some of the operations, it is not appropriate to use all the processors for MPI tasks, that is why we only show the performance of shared memory (SCNI) and SCI schemes for 4 processes. From Figure 5.4 we can see that the improvement in microbenchmarks have been translated into application performance. The asynchronous memory copy operations have improved IS performance by up to 12%, and PSTSWM performance by up to 7%. The improvement is expected because both IS and PSTSWM use a lot of large messages. The message size distribution is shown in Table 5.3, which is profiled in terms of number of messages. Further, we observe that although large messages dominate in PSTSWM, the improvement seen is not significant. This is because PSTSWM is a computation intensive benchmark, e.g. when running the medium problem size on 4 processes, only 6.6% of the total time is spent in MPI. From Figure 5.4 and Table 5.3 we can see that the asynchronous memory operations proposed in this paper will benefit MPI applications which have bulk data transfer.



(a) 2 Processes



(b) 4 Processes

Figure 5.4: MPI Application Performance

Table 5.3: Message Size Distribution of MPI benchmarks

Message Size	0 - 32KB	32KB - 1MB	1MB - 64MB
IS.A.2	68.1%	0	31.9%
IS.A.4	70.6%	0	29.4%
IS.B.2	68.1%	0	31.9%
IS.B.4	70.6%	0	29.4%
IS.C.2	68.1%	0	31.9%
IS.C.4	70.6%	0	29.4%
PSTSWM.small.2	4.0%	0.4%	95.6%
PSTSWM.small.4	3.6%	96.4%	0
PSTSWM.medium.2	4.0%	0	96.0%
PSTSWM.medium.4	3.0%	0.5%	96.5%

5.4 Summary

In this chapter, we have proposed three schemes to provide overlap of memory copy operation with computation. In the first scheme, SCI (Single-Core with I/OAT), we *offload* the memory copy operations to the Intel on-chip DMA engines. In the second scheme, MCI (Multi-Core with I/OAT), we not only offload the memory copy operation, but also *onload* the startup overheads associated with the copy engine to a dedicated core. For systems without any hardware copy engine support, we have proposed a third scheme, MCNI (Multi-Core with No I/OAT) that *onloads* the memory copy operation to a dedicate core. We have integrated the schemes with MPI library, and done MPI level performance evaluation. Our results show that MPI latency and bandwidth can be improved significantly and the performance of applications such as NAS and PSTSWM can be improved by up to 12% and 7%, respectively, compared to the traditional implementations.

CHAPTER 6

EFFICIENT KERNEL-LEVEL AND USER-LEVEL HYBRID APPROACH

Traditionally there have been three approaches for MPI intra-node communication: network loopback, user-level shared memory, and kernel assisted direct copy, as described in Section 1.2. In order to obtain optimized MPI intra-node communication performance, it is important to have a comprehensive understanding of the approaches and improve upon them. Since network loopback is not commonly used in modern MPI implementations due to its higher latency, in this chapter we only consider the shared memory and kernel-assisted approaches. To achieve high performance, in this chapter we design and develop a set of experiments and optimization schemes, and aim to answer the following questions:

- *What are the performance characteristics of these two approaches?*
- *What are the advantages and limitations of these two approaches?*
- *Can we design a hybrid scheme that takes advantages of both approaches?*
- *Can applications benefit from the hybrid scheme?*

We have carried out this study on an Intel quad-core (Clovertown) cluster and use a three-step methodology. The rest of the chapter is organized as the follows:

In Section 6.1 we introduce LiMIC2, the kernel based approach used in the study. We present the initial performance study using micro-benchmarks in Section 6.2 and propose an efficient hybrid approach in Section 6.3. We evaluate the hybrid approach using collective operations and applications in Section 6.4 and finally summarize in Section 6.5.

6.1 Introduction of LiMIC2

As described in Chapter 4, LiMIC is a Linux kernel module that directly copies messages from the user buffer of one process to another. It improves performance by eliminating the intermediate copy to shared memory buffer. The first generation of LiMIC [49] is a stand-alone library that provides MPI-like interfaces, such as LiMIC_send and LiMIC_recv. The second generation, LiMIC2 [50], provides a set of lightweight primitives that enables MPI libraries to do memory mapping and direct copy, and relies on the MPI library for message matching and queueing. Therefore, compared with LiMIC, LiMIC2 provides lower overhead and implementation complexity. In this chapter, we use MVAPICH-LiMIC2, which integrates MVAPICH with LiMIC2 for intra-node communication.

MVAPICH-LiMIC2 uses a rendezvous protocol for communication. The sender first sends a *request_to_send* message to the receiver together with the send buffer information. Upon receiving the request, the receiver maps the send buffer to the kernel space and copy the message to its receive buffer. When the copy finishes, the receiver sends a *complete* message to the sender.

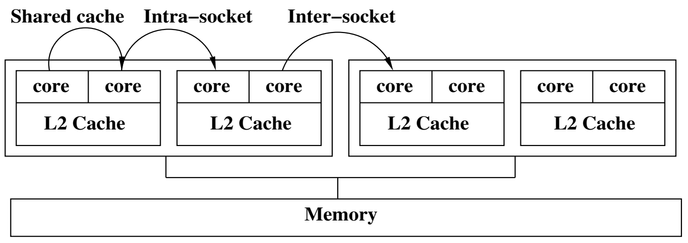


Figure 6.1: Illustration of Intel Clovertown Processor

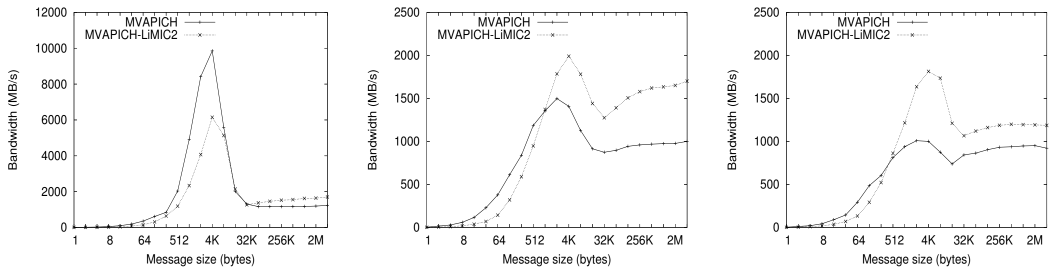
6.2 Initial Performance Evaluation and Analysis: Micro-Benchmarks

In this section we study the performance of shared-memory (MVAPICH) and LiMIC2 (MVAPICH-LiMIC2) approaches using micro-benchmarks.

Testbed: We use an Intel Clovertown cluster. Each node is equipped with dual quad-core Xeon processor, i.e. 8 cores per node, running at 2.0GHz. Each node has 4GB main memory. The nodes are connected by InfiniBand DDR cards. The nodes run Linux 2.6.18. We conduct the micro-benchmark experiments on a single node. As shown in Figure 6.1, there are three cases of intra-node communication: shared-cache, intra-socket, and inter-socket.

6.2.1 Impact of Processor Topology

As described above, there are three cases of intra-node communication on our system: shared cache, intra-socket, and inter-socket. In this section we examine the bandwidth of MVAPICH and MVAPICH-LiMIC2 in these three cases. We use multi-pair benchmarks [15] instead of single-pair because usually all the cores are activated when applications are running. On our system there are 8 cores per node,



(a) Shared Cache (b) Intra-socket (c) Inter-socket

Figure 6.2: Multi-pair Bandwidth

so we create 4 pairs of communication. The benchmark reports the total bandwidth for the 4 pairs.

The multi-pair bandwidth results are shown in Figure 6.2. In this benchmark, each sender sends 64 messages to the receiver. Each message is sent from and received to a different buffer. The send buffers are written at the beginning of the benchmark. When the receiver gets all the messages, it sends an acknowledgement. We measure the bandwidth achieved in this process.

From Figure 6.2(a), we see that MVAPICH performs better than MVAPICH-LiMIC2 up to 32KB for the shared cache case. In this case, because the two cores share the L2 cache, memory copies only involve intra-cache transactions as long as the data can fit in the cache. Therefore, although there is one more copy involved in MVAPICH, the cost of the extra copy is so small that it hardly impacts performance. On the other hand, MVAPICH-LiMIC2 uses operations such as trapping to the kernel and mapping memory. This overhead is sufficiently large to negate the benefit of having only one copy. Therefore, only for large messages that cannot totally fit in the cache we can see the benefit with MVAPICH-LiMIC2. We note that the L2 cache on our system is 4MB and shared between two cores; essentially each core has about 2MB cache space. Since in this experiment the window size is 64, for 32KB messages the total buffer is already larger than the available cache space (32KB x 64 = 2MB).

In comparison, if the cores do not share cache, then MVAPICH-LiMIC2 shows benefits for a much larger range of message sizes, starting from 2KB for intra-socket and 1KB for inter-socket (see Figures 6.2(b) and 6.2(c)). This is because in these two cases memory copies involve either cache-to-cache transaction or main memory access, which is relatively expensive. Therefore, saving a copy can improve performance significantly. We observe that with MVAPICH-LiMIC2, bandwidth is improved by up to 70% and 98% for intra-socket and inter-socket, respectively.

6.2.2 Impact of Buffer Reuse

Figure 6.2 clearly shows that communication is more efficient if the buffers are in the cache. Buffer reuse is one of the most commonly used strategies to improve cache utilization. In this section we examine the impact of buffer reuse on MVAPICH and MVAPICH-LiMIC2. There is no buffer reuse in the benchmark used in Section 6.2.1 since each message is sent from and received to a different buffer. To simulate the buffer reuse effect in applications, we modify the benchmark to run for multiple iterations so that starting from the second iteration the buffers are reused. In the beginning of each iteration we rewrite the send buffers with new content.

The intra-socket results are shown in Figure 6.3. The shared cache and inter-socket results follow the same trend. From Figure 6.3 we can see that the performance of both MVAPICH and MVAPICH-LiMIC2 improves with buffer reuse. This is mainly due to cache effect: starting from the second iteration, the buffers may already reside in the cache. For messages larger than 32KB, buffer reuse does not affect the performance of either MVAPICH or MVAPICH-LiMIC2 because the total buffer size is already larger than the cache size (32KB x 64 = 2MB).

Comparing the performance of MVAPICH and MVAPICH-LiMIC2 in the buffer reuse situation, we see that the benefit of using MVAPICH-LiMIC2 is larger than that in the no buffer-reuse case for medium messages. The reason is that MVAPICH-LiMIC2 does not use the intermediate buffer for data transfer, and thus has better cache utilization. We analyze cache utilization in detail in Section 6.2.3. From the results shown in this section we conclude that applications that have more buffer reuse potentially benefit more from MVAPICH-LiMIC2.

A similar trend can be observed with multi-pair latency test too. The results are not shown here to avoid redundancy.

6.2.3 L2 Cache Utilization

In this section, we analyze the cache effect in the buffer reuse experiment.

We use the same benchmark as in Section 6.2.2, and use *OProfile* [19] to profile the L2 cache misses during the experiment. We show the number of L2 cache misses as well as the improvement in cache utilization achieved by MVAPICH-LiMIC2 over MVAPICH in Figure 6.4. We start from 1KB since MVAPICH-LiMIC2 shows better performance starting from 1KB in Figure 6.3. As expected, we see that cache

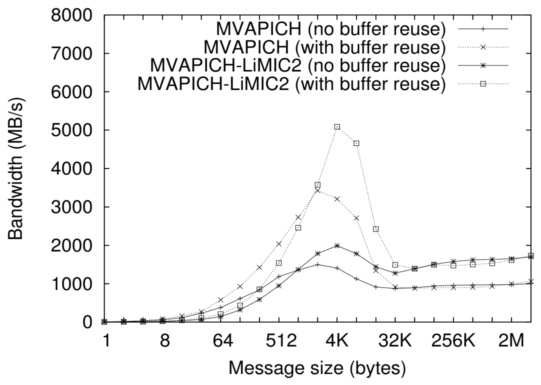


Figure 6.3: Impact of Buffer Reuse (Intra-socket)

misses increase with increase in message size. For the whole range of message sizes, MVAPICH-LiMIC2 has fewer cache misses than MVAPICH, showing a constant improvement of about 7% when the message is larger than 16KB. This is because MVAPICH-LiMIC2 does not involve an intermediate buffer like MVAPICH. Another interesting observation is that the improvement percentage presents almost the same trend as the performance comparison in Figure 6.3. This further explains the benefits obtained by MVAPICH-LiMIC2 and demonstrates our conclusion in Section 6.2.2.

6.2.4 Impact of Process Skew

Process skew can potentially degrade application performance. In this section, we want to examine the ability of MVAPICH and MVAPICH-LiMIC2 to overcome process skew effect.

As described in Section 6.1, MVAPICH-LiMIC2 copies messages directly from the sender's user buffer to the receiver's user buffer with the help of the OS kernel. Therefore, a send operation cannot complete until the matching receive completes.

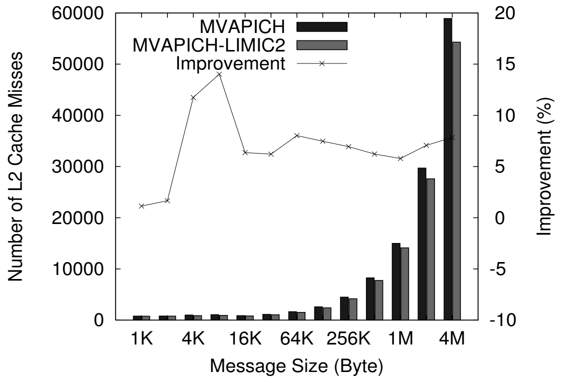


Figure 6.4: L2 Cache Misses

This means that the MVAPICH-LiMIC2 performance might potentially be influenced by process skew. On the other hand, MVAPICH uses an intermediate buffer and eager protocol for small and medium messages. This means that for small and medium messages, a send operation simply involves copying message to the intermediate buffer without interaction with the receive process. Therefore, MVAPICH is potentially more skew-tolerant.

We have designed a benchmark that simulates the process skew effect. Figure 6.5 illustrates the algorithm. There are two processes involved, a producer and a consumer. The producer computes for *c1* amount of time, and then sends the intermediate result to the consumer using the non-blocking *MPI_Isend*. The consumer receives this message using the blocking *MPI_Recv*, and does further processing on it for *c2* amount of time. This process repeats for *window_size* iterations, and then the producer calls *MPI_Waitall* to make sure all the *MPI_Isend*'s have been completed. This kind of scenario is commonly used in many applications. We set *c2* to be much larger than *c1* so that the two MPI processes are skewed. We measure the

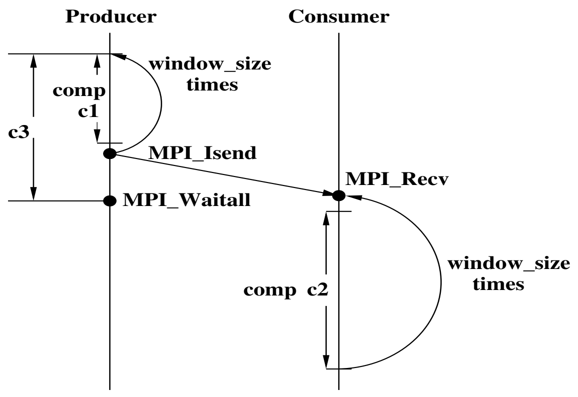


Figure 6.5: Process Skew Benchmark

total amount of time that the producer needs to complete this process, shown as *c3* in Figure 6.5. This is essentially the latency on the producer side before it can continue with other computation work.

Based on the characteristics of MVAPICH and MVAPICH-LiMIC2, theoretically we expect them to perform as follows:

*c3(MVAPICH) = (c1 + t(MPI_Isend)) * window_size + t(MPI_Waitall)*

*c3(MVAPICH-LiMIC2) = (t(MPI_Recv) + c2) * window_size + t(MPI_Waitall)*

Since *c2* is much larger than *c1*, we can expect *c3(MVAPICH-LiMIC2)* to be much larger than *c3(MVAPICH)*.

We show the experimental results in Figure 6.6. In this experiment, we set the message size as 16KB, *c1=1us* and *window_size=64*, and record the producer latency (*c3*) with different consumer computation time (*c2*). From Figure 6.6, we can see that the experimental result conforms to the theoretical expectation that *c3(MVAPICH)* is much lower than *c3(MVAPICH-LiMIC2)*. Further, *c3(MVAPICH)* does not increase as *c2* increases, indicating that MVAPICH is more resilient to

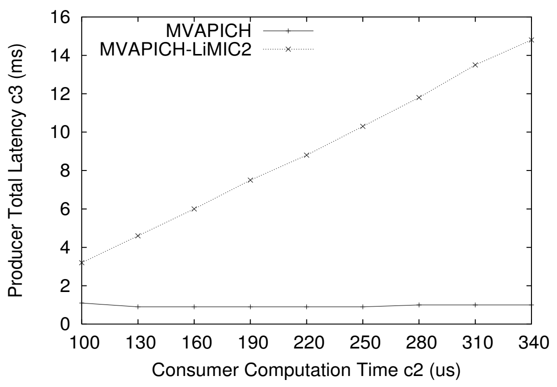


Figure 6.6: Impact of Process Skew

process skew. On the other hand, *c3(MVAPICH-LiMIC2)* grows linearly as *c2* increases, which could be a potential limitation of MVAPICH-LiMIC2. We will describe optimizations to best combine shared memory and LiMIC2 in Section 6.3.2 to alleviate process skew effect.

6.3 Designing the Hybrid Approach

From the micro-benchmark results and analysis, we have seen that MVAPICH and MVAPICH-LiMIC2 both have advantages and limitations in different situations and for different message sizes. In this section, we propose two optimization schemes, topology-aware thresholds and skew-aware thresholds, that efficiently combine the shared memory approach in MVAPICH with LiMIC2.

6.3.1 Topology Aware Thresholds

We need to carefully decide the threshold to switch from shared memory to LiMIC2 in order to efficiently combine these two approaches. From the results shown in Section 6.2.1, we know that the performance characteristics of MVAPICH and MVAPICH-LiMIC2 are different for different intra-node communication cases (shared cache, intra-socket, and inter-socket). Therefore, a single threshold may not suffice for all the cases. In this section, we illustrate our design of the topology aware thresholds.

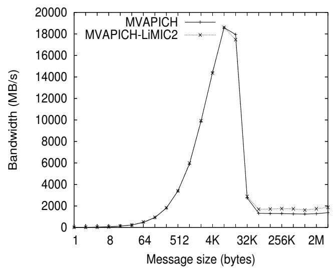
The latest Linux kernels have the ability to detect the topology of multi-core processors. The information is exported in "sysfs" file system [70]. The following fields exported under */sys/devices/system/cpu/cpuX/topology/* provide the topology information that we need (X in *cpuX* is the CPU number):

- physical_package_id: Physical socket id of the logical CPU
- core_id: Core id of the logical CPU on the socket

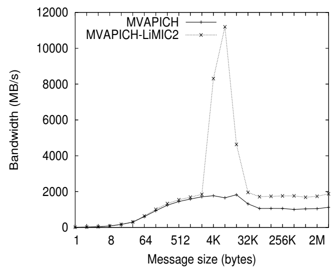
By parsing this information, every process has the knowledge about the topology. If the cache architecture is also known (Figure 6.1), for a given connection, a process knows which case it belongs to - shared cache, intra-socket, or inter-socket. It is thus able to use different thresholds for different cases. Of course, to make sure that the process does not migrate to other processors, we use the *CPU affinity* feature provided by MVAPICH [15].

Based on the results in Figure 6.2, we use 32KB as the threshold for the shared cache case, 2KB for intra-socket, and 1KB for inter-socket. After we apply these thresholds, we have the optimized results for all the cases. The results are presented in Figure 6.7.

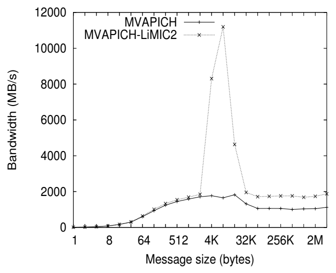
The topology detection method discussed in this section can be used on other Linux based platforms too, such as AMD multi-core systems. Also, different kinds of optimizations can be applied based on topology information and platform features.



(a) Shared Cache



(b) Intra-socket



(c) Inter-socket

Figure 6.7: Multi-pair Bandwidth with Topology Aware Thresholds

6.3.2 Skew Aware Thresholds

We have seen from Section 6.2.4 that the shared memory approach used in MVAPICH is more resilient to process skew for medium messages. On the other hand, MVAPICH-LiMIC2 provides higher performance for medium messages. To take advantages of both methods, we have designed an adaptive scheme that uses shared memory when there is process skew, and LiMIC2 otherwise.

We detect process skew by keeping track of the length of the *unexpected queue* at the receiver side. Messages that are received before the matching receive operations have been posted are called *unexpected messages.* Such requests are queued in an unexpected queue. When the matching receive is posted, the corresponding request

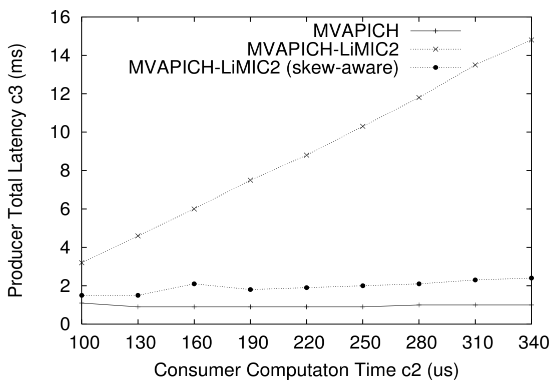


Figure 6.8: Impact of Skew Aware Thresholds

is removed from the unexpected queue. Therefore, the length of the unexpected queue reflects the extent of process skew. If the length is larger than the threshold for a long period of time, then the receiver determines that process skew has occurred, and sends a control message to the sender to indicate the situation. Upon receiving this message, the sender increases the threshold to switch to LiMIC2 for this connection so that medium messages will go through shared memory to alleviate the process skew effect. Later if the receiver detects process skew has gone, it can send another control message so that the sender will change back the threshold to use LiMIC2 for higher performance.

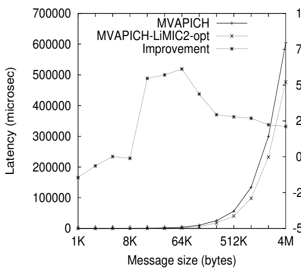
We show the results of the skew-aware thresholds in Figure 6.8. We used the same benchmark with the same set of parameters as described in Section 6.2.4. We see that the sending process can quickly notice the process skew situation and adapt the threshold to it. As a result, the skew-aware MVAPICH-LiMIC2 achieves much lower producer latency, close to that of MVAPICH.

6.4 Performance Evaluation with Collectives and Applications

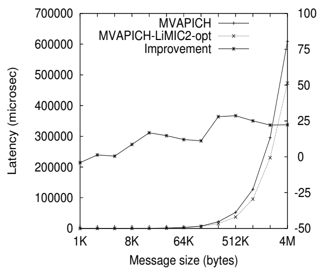
In this section we study the impact of the hybrid approach on MPI collective operations and applications. We refer to the hybrid approach as *MVAPICH-LiMIC2-opt* because it is essentially an optimized version of MVAPICH-LiMIC2. We use Intel MPI Benchmark (IMB) [8] for collectives, and NAS [38], PSTSWM [20] and HPL from HPCC benchmark suite [47] for applications. To better understand the application behaviors and relationship with MPI implementations we have also done profiling to the applications.

6.4.1 Impact on Collectives

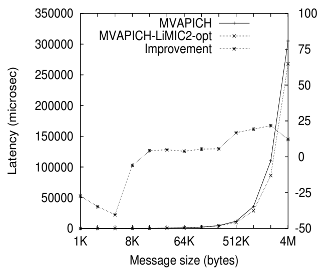
We show the results of three typical collective operations, MPI_Alltoall, MPI_Allgather, and MPI_Allreduce, in Figure 6.9. MPI collective operations can be implemented either on top of point-to-point communication or directly in the message passing layer using optimized algorithms. Currently MVAPICH-LiMIC2-opt uses point-to-point based collectives and MVAPICH uses optimized algorithms for MPI_Allreduce for messages up to 32KB [58]. From the figures we see that MPI collective operations can benefit from using MVAPICH-LiMIC2-opt, especially for large messages. The performance improves by up to 60%, 28%, and 21% for MPI_Alltoall, MPI_Allgather, and MPI_Allreduce, respectively. We note that for messages between 1KB and 8KB, MVAPICH performs better for MPI_Allreduce due to the use of the optimized algorithms. This indicates that the performance of LiMIC2 based collectives can be further optimized by using specially designed algorithms.



(a) MPI_Alltoall



(b) MPI_Allgather



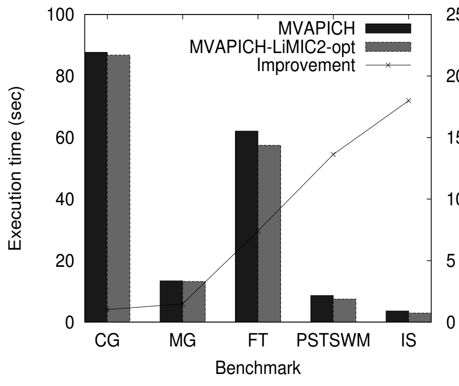
(c) MPI_Allreduce

Figure 6.9: Collective Results (Single Node 1x8)

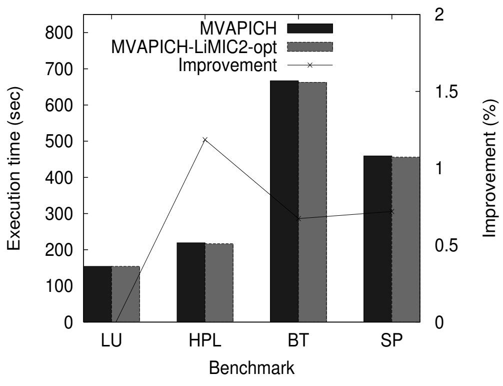
6.4.2 Impact on Applications

In this section we evaluate the impact of the hybrid approach on application performance. The single-node results are shown in Figures 6.10 and 6.11 (Class B for NAS and small problem size for PSTSWM). The corresponding message size distribution is shown in Table 6.1. The cluster-mode results are shown in Figure 6.12 (Class C for NAS and medium problem size for PSTSWM), in which we use 8 nodes and 8 processes per node (8x8).

From Figure 6.10(a) we see that MVAPICH-LiMIC2-opt can improve the performance of FT, PSTSWM, and IS significantly. The improvement is 8% for FT, 14% for PSTSWM, and 17% for IS, respectively. If we look at Figure 6.11(a) we find that MVAPICH-LiMIC2-opt has better cache utilization for these benchmarks. Most messages used in these benchmarks are large as shown in Table 6.1. This means that applications that use large messages will potentially benefit from MVAPICH-LiMIC2-opt.



(a)



(b)

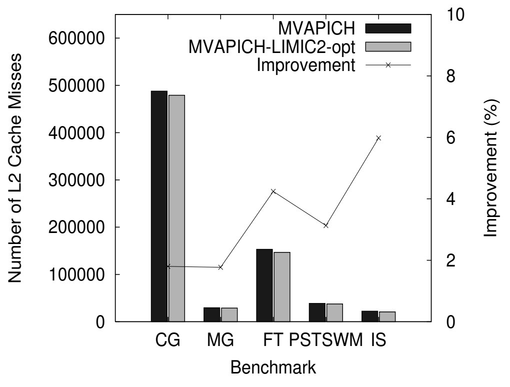
Figure 6.10: Application Performance (Single Node 1x8)

The improvement is under 5% for other benchmarks mostly because these benchmarks do not use many large messages. For BT and SP, although most messages are large, since the fraction of time spent on communication is not significant we do not observe large performance improvement.

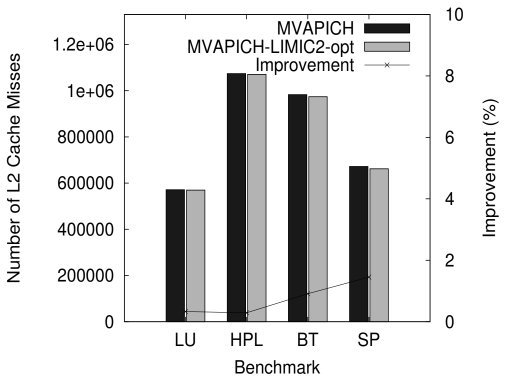
From Figure 6.12 we see that in cluster mode where there is a mix of intra-node and inter-node communication, applications can still benefit from using MVAPICH-LiMIC2-opt, e.g. PSTSWM performance improves by 6%, which suggests that MVAPICH-LiMIC2-opt is a promising approach for cluster computing.

6.5 Summary

In this chapter, we use a three-step methodology to design a hybrid approach for MPI intra-node communication using two popular approaches, shared memory (MVAPICH) and OS kernel assisted direct copy (MVAPICH-LiMIC2). The study



(a)



(b)

Figure 6.11: L2 Cache Misses in Applications (Single Node 1x8)

has been done on an Intel quad-core (Clovertown) cluster. We have evaluated the impacts of processor topology, communication buffer reuse, and process skew effects on these two approaches, and profiled the L2 cache utilization. From the results we find that MVAPICH-LiMIC2 in general provides better performance than MVAPICH for medium and large messages due to fewer number of copies and efficient cache utilization, but the relative performance varies in different situations. For example, depending on the physical topology of the sending and receiving processes, the thresholds to switch from shared memory to LiMIC2 can be different. In addition, if the application has higher buffer reuse rate, it can potentially benefit more from MVAPICH-LiMIC2. We also observe that MVAPICH-LiMIC2 has a potential limitation that it is not as skew-tolerant as MVAPICH. Based on the results and the analysis, we have proposed topology-aware and skew-aware thresholds to build an efficient hybrid approach. We have evaluated the hybrid approach using MPI

Table 6.1: Message Size Distribution (Single Node 1x8)

Apps	< 1K	1K-32K	32K-1M	> 1M
CG	62%	0	38%	0
MG	52%	28%	20%	0
FT	17%	0	0	83%
PSTSWM	2%	1%	97%	0
IS	44%	15%	0	41%
LU	30%	69%	1%	0
HPL	58%	37%	3%	2%
BT	1%	0%	99%	0
SP	1%	0%	99%	0

collective and application level benchmarks. We observe that the hybrid approach can improve the performance of MPI_Alltoall, MPI_Allgather, and MPI_Allreduce by up to 60%, 28%, and 21%, respectively. And for applications, it can improve the performance of FT, PSTSWM, and IS by 8%, 14%, and 17%, respectively.

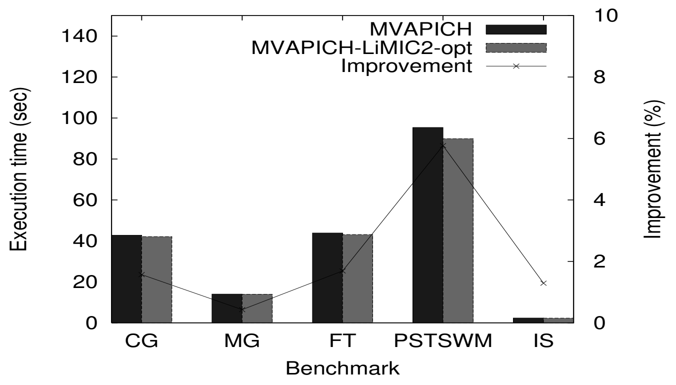


Figure 6.12: Application Performance on 8 nodes (8x8)

CHAPTER 7

ANALYSIS OF DESIGN CONSIDERATIONS FOR MULTI-CHANNEL MPI

To optimize communication performance, many MPI implementations such as MVAPICH [15] provide multiple communication channels. These channels may be used either for intra- or inter-node communication. Efficient polling of these communication channels for discovering new messages is often considered to be one of the key design issues in implementing MPI over any network layer. In addition, based on characteristics of each channel, we can utilize several channels for intra-node communication. In order to efficiently design and implement these channel interfaces, we need a centralized policy. Since communication patterns as well as the need for overlap of communication and computation vary widely over different applications, it becomes hard to design a general purpose policy. We need to carefully consider the overheads and benefits offered by each channel.

In this chapter, we try to bring forward important factors that should be considered to efficiently utilize several MPI channels through in-depth measurements and analysis. The rest of this chapter is organized as the follows: In Section 7.1, we study the polling schemes among multiple channels and their overheads. Then, we explore methodologies to decide the thresholds between multiple channels in Section 7.2.

We consider latency, bandwidth, and CPU resource requirement of each channel to decide the thresholds. We present our performance evaluation in Section 7.3 and finally summarize in Section 7.4.

7.1 Channel polling

In this section we discuss about channel polling overhead and schemes.

7.1.1 Channel polling overheads

Different channels have different polling overheads. In this section we analyze the polling overhead for each channel.

Network Channel Overhead: The network channel consists of RDMA and Send/Receive channels. Since RDMA is used for the RDMA channel, there is no software involvement at the receiver side. Therefore, the only way to check for incoming messages is by polling memory locations. The overhead involved in polling memory locations is around 0.03μs per connection. The overall polling overhead increases as the number of RDMA connections increases. The other network communication channel uses InfiniBand send/receive primitives, which generate message completion events. The receiver polls the completion queue to check new incoming messages. The overhead associated with polling the completion queue is constant regardless of the number of processes because the same completion queue is shared among all connections. However, it takes around 0.3μs to poll an empty completion queue, which is relatively high. In this section, we consider the polling overheads for RDMA and send/receive channels as the network channel polling overhead.

Shared Memory Channel Overhead: The shared memory channel uses a FIFO queue for each shared memory connection. In addition, the channel maintains a counter which indicates whether a new message is available for this connection. The polling overhead of this channel is around 0.06μs and increases as the number of processes running on the same node increases. It is to be noted that since most SMP nodes in clusters are 2-way to 16-way, this polling overhead is not significant. To compare shared memory channel polling overhead with the network channel overhead, we measured them on various system sizes as shown in Figure 7.1. We can observe that network channel polling overhead increases faster than shared memory channel as the system size increases. It is because the number of inter-node connections per process increases in proportion to $(P \times N)$, where P is the number of processors on one node and N is the number of nodes. On the other hand, the number of connections for intra-node communication increases in proportion to only P. It is to be noted that most of clusters have a much larger N value than P.

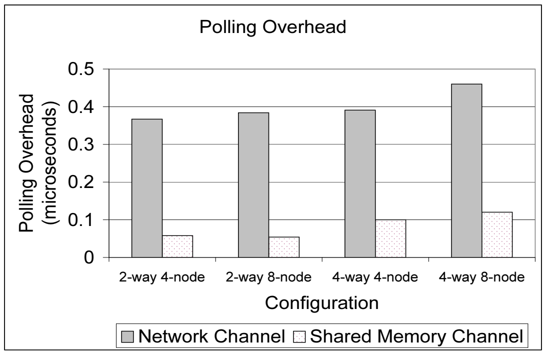


Figure 7.1: Polling overhead of network channel and shared memory channel

Kernel Module Channel Overhead: The kernel module channel [48] copies messages directly from the sender buffer to the receiver buffer. However, polling of the kernel module channel is expensive as it requires a context-switch to the kernel-space, which takes around 3μs. We can consider following two ways to poll on the kernel module channel:

- Busy polling of the kernel module in the blocking MPI send, receive, or wait functions. In this case, we poll the kernel module channel explicitly only when a message is expected to arrive from that channel.

- The kernel module can provide some signaling bit to indicate the arrival of new messages to the MPI layer. Although it can reduce the number of context switches, still we need to trap into the kernel to match MPI headers. In the worst case, if some unexpected message arrives in the kernel, the MPI layer still needs to poll that message because the signal bit does not have information about the MPI header.

In order to avoid multiple context switches and overhead to poll the kernel module, we place the polling of the kernel module outside the main MPI progress engine. So, if any messages are not expected from the kernel module channel, then that channel is not polled at all. All unexpected messages arriving through the kernel module channel are kept queued by the kernel module. The messages are copied when the receiver posts the matching receive.

7.1.2 Channel polling schemes

As described in section 7.1.1 there are different costs associated with polling of each channel. In this section we design different polling schemes to reduce the overhead associated with polling network and shared memory channels and enable faster message discovery. As we have described in section 7.1.1, polling of the kernel module is placed outside the main progress engine. So the kernel module is not polled if no messages are expected from it. Therefore, we exclude kernel module from the study of these polling schemes.

Static channel polling scheme: Static polling scheme decides the polling policy at the start of the MPI application. This scheme can assign different priorities (or weights) to different channels. The intuitive idea behind this scheme is that some channels may be used more frequently or faster than others. To decide the priority, we need to consider the following factors:

- Polling Overhead: If a channel has a significantly less polling overhead than others, we can consider to poll this channel more frequently. In this way we can reduce the message discovery time for the channel without adding a large overhead to poll other channels.

- Message Latency: If a channel has lower message passing latency and higher bandwidth than others, it may receive relatively more messages in a short period of time. Accordingly, we can assign higher priority to this channel.

In this section, we consider both factors. As we have discussed in section 7.1.1, the overhead of polling the shared memory channel is the least. Also we notice that this channel has the lower latency than the network channel as shown in Section 7.3.2. Therefore, we give most priority to the shared memory channel. In this scheme, we decide the frequency of polling between channels based on the priority ratio assigned statically at the application startup phase.

Dynamic channel polling scheme: Dynamic polling schemes can change polling priority over the course of the execution of the MPI application. There are various factors to be considered while designing such a dynamic scheme:

- Update Rate: This factor determines how often the priority ratios are updated. A very high update rate would imply increased overheads for short messages, whereas a low update rate would miss smaller bursts of messages from other channels.

- Message History: This factor determines the number of messages recorded for computing the new priority ratio. The more messages are considered, the slower the priority ratio will change. This might miss smaller bursts of messages, whereas when lower number of messages are considered a lot of fluctuation may occur even with small bursts of messages from a channel.

In this section, we use the following scheme to compute priority ratio: Suppose in the last h messages received, m of which are from shared memory channel, and n of which are from network channel, then $\textit{priority ratio} = m/n + 1$. Whenever h messages are received, we update the priority ratio, and reset h to zero. So the message history length here is the same with update rate. Also, for the reasons we stated in static polling scheme section, the polling priority of shared memory channel is always higher than or equal to that of network channel.

7.2 Channel thresholds

Network, shared memory, and kernel module can all be used for intra-node communication. These channels have different performance characteristics. Some channels have low startup latency and some channels have high bandwidth. In addition, some channels do not require the involvement of host CPU. In this section, we study on selecting appropriate thresholds for efficient intra-node message passing.

7.2.1 Communication startup and message transmission overheads

In the network channel, messages for intra-node communication are DMAed into the network interface card and looped back to the host memory. Therefore, there exist two DMA operations. Although I/O buses are getting faster, the DMA overhead is still high. Further, the DMA startup overhead is as high as several microseconds.

We note that the shared memory channel involves the minimal setup overhead (less than 1.2μs) for every message exchange. However, there are at least two copies involved in the message exchange. This approach might tie down the CPU with memory copy time. In addition, as the message size grows, the performance of the copy operation becomes even worse because vigorous copy-in and copy-out destroy the cache contents.

The kernel module channel involves only one copy and is able to maximize the cache effect. However, there are other overheads such as trap, memory mapping, and locking of data structures. The trap and locking overheads are involved for every message passing and larger than 3μs. The memory mapping overhead increases as the number of pages for the user buffer increases, which takes around 0.7μs per page. In addition, although the number of copy operations is reduced, the CPU resource is still required to perform the copy operation.

7.2.2 Threshold decision methodology

To decide the thresholds, we consider several important factors, such as latency, bandwidth, and CPU utilization, which can largely affect application performance. However, different thresholds might be required by different applications because each of them has different communication characteristics and programming assumptions. In this section, we discuss two different approaches for choosing appropriate thresholds.

Microbenchmark based decision: In general, it is very difficult to decide the threshold of communication channel for all applications. However, it is widely accepted that such decisions can be based on latency and bandwidth measurements. Therefore we can look at MPI microbenchmarks to see the basic performance of each channel.

CPU utilization based decision: In this approach we measure the overlapping of computation and communication. Although some channels might have higher message latency, they may effectively overlap computation and communication. This is beneficial for applications that are efficiently programmed to overlap them. Since many MPI implementations use the rendezvous protocol for large messages and make a communication progress within MPI calls, applications are usually required to call an MPI function such as MPI_Iprobe to make an efficient overlap between computation and communication. However, this is quite application dependent. For applications which mostly use blocking operations, simply selecting the channel with lowest latency would be enough.

7.3 Performance Evaluation

In this section we present our results on design considerations for multi-channel MPI, specifically results on polling schemes and threshold determination.

7.3.1 Evaluation of Polling Schemes

We conducted experiments on an 8-node cluster with the following configuration: Super Micro SUPER X5DL8-GG nodes with dual Intel Xeon 3.0 GHz processors, 512 KB L2 cache, 2 GB memory, PCI-X 64-bit 133 MHz bus. The Linux kernel version used was 2.4.22smp from kernel.org. All nodes are equipped with Mellanox InfiniHost MT23108 HCAs and installed the Mellanox InfiniBand stack [60]. The version of VAPI was 3.2 and firmware version 3.2. The nodes are connected through Mellanox MTS 2400 24-port switch.

One crucial factor to determine for static polling scheme is *"how much priority should be given to the shared memory channel?"* Obviously, if we give more priority to shared memory channel, then the shared memory latency will reduce. But at the same time the latency of messages coming over the network will also increase.

To find out the optimal priority ratio, we conducted the standard ping-pong latency test with different priority ratios. Figure 2 shows variation of ping-pong latency with various priority ratios for 4B and 2KB message sizes. We can observe from these figures that if we give shared memory channel a priority ratio of 50, then we can get a reasonably balanced improvement of intra-node latency - 12% improvement for 4B message and 9% improvement for 2KB message - without hurting network latency. For 4B message, our experiments indicate that we can achieve up to 37% improvement in intra-node latency using the static polling priority 1000, but it hurts the network channel latency significantly. As message size increases, the benefit of polling scheme reduces because the message transmission overhead becomes larger than the polling overhead.

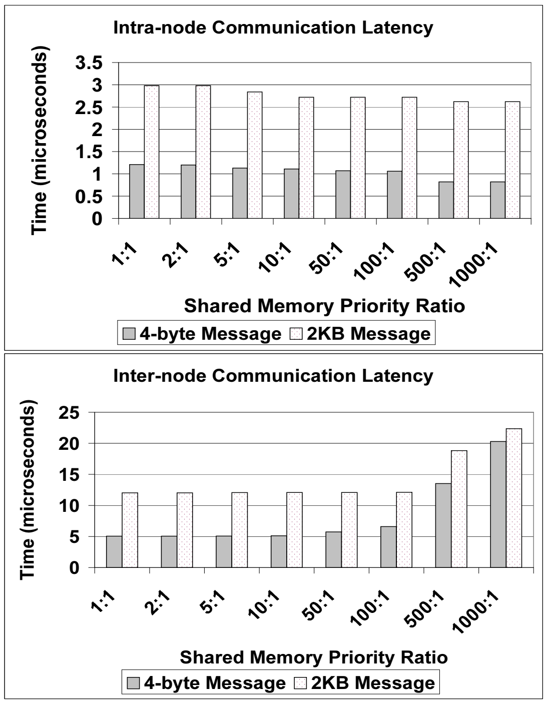


Figure 7.2: Latency of static polling scheme

In order to evaluate the dynamic polling scheme we need to devise a new MPI microbenchmark that appropriately captures the message discovery time at the MPI layer. There are three processes in the benchmark. Two processes are on the same node, whereas one process is on a separate node. This process sends messages over the network, whereas the process on the same node sends messages exclusively through shared memory channel. On the receipt of each message the *"root"* process replies with an ACK. The process sending the *"burst"* number of messages to the

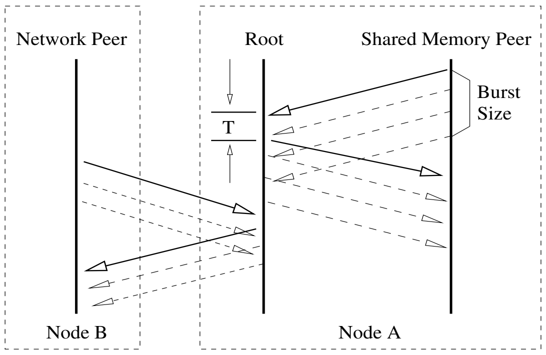


Figure 7.3: Message discovery microbenchmark

root is alternately selected between the network peer and the shared memory peer. This test captures the message discovery time by the root process before it can send an ACK to the peer process. Figure 7.3 illustrates this microbenchmark where we are trying to measure time T.

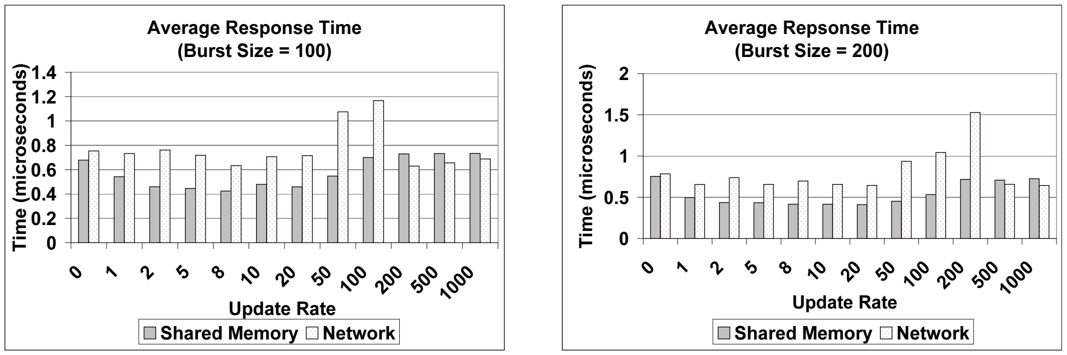


Figure 7.4: Message discovery time of dynamic polling scheme

Figure 4 shows the performance results of this microbenchmark with the burst sizes of 100 and 200 for 4B message. We observe that with the increase of update rate, the message discovery time actually decreases. The update rate of 8 or 10 is enough not to introduce too much overhead and also sustain fairly small burst of messages. Our experiments indicate that we can achieve up to 45% improvement rate of message discovery time with burst size of 200. However, when the update rate becomes higher, the overhead causes the discovery time to rise. We also observe that when the burst size is equal to the update rate, the discovery time increases significantly due to continuous wrong predictions.

7.3.2 Evaluation of Thresholds

In this section, we run the above mentioned decision approaches on the cluster described in section 7.3.1. We use the standard ping-pong latency and bandwidth to evaluate the threshold points for the three channels.

Figure 5 shows the experimental results of the latency and bandwidth tests. We find that for messages smaller than 4KB, it is beneficial to use shared memory channel. This is because shared memory channel avoids a high communication startup time such as kernel trap and DMA initialization. For messages greater than 4KB, it is useful to have the kernel module channel. This is mainly because the number of copies has been reduced to one. Also, we can observe that the bandwidth for the kernel module channel drops significantly from 256KB message size. It is because the cache size on the node used is 512KB. Both the sender and receiver buffers and some additional data structures cannot fit into the cache beyond this message size. However, the bandwidth offered by the kernel module channel is still greater than others.

To analyze different channels' capability of overlapping computation and communication, we conducted experiments as follows: Two processes running on the same node call MPI_Isend and MPI_Irecv. Then they execute a computation loop for a given computation time (i.e., values in x-axis of Figure 7.6). Within the computation loop, processes call MPI_Iprobe to make a communication progress for every 100μs. After the computation time, they call MPI_Waitall and calculate ($Total\_Time/Computation\_Time$), where $Total\_Time$ includes both computation and communication time. A value closer to 1 means more overlapping between computation and communication.

Figure 7.6 shows experimental results for 4B and 128KB messages, respectively. For small messages, the communication startup time is the dominant overhead while message transmission time is very small. Since the shared memory channel has the lowest communication startup time, this channel shows closer values to 1 than others with small computation time. It is to be noted that the network channel shows better overlapping than the kernel module channel for small messages. Although the network channel has a larger startup time than the kernel module, the DMA initialization time, which is the dominant startup overhead for the network channel, does not require CPU resource at all. Thus most of startup time of the network channel can be overlapped with computation, which results in the better overlapping than the kernel module channel. Since communication overhead becomes relatively smaller as the computation time grows, there is no difference among three channels with large computation time values.

For large messages, we observe that the network channel can make the computation and communication fully overlap. It is because the network channel does not need any CPU resource to move intra-node messages. However, the shared memory and kernel module channels require the CPU to copy messages. Therefore, it is difficult to expect them to achieve a good overlapping. Since the kernel module channel needs only one copy, this channel shows better overlapping than the shared memory channel. As the computation time increases, all three channels again show the same overlapping capability. It is because the computation time is too large comparing with communication time. Overall, to maximize the computation and communication overlapping, the shared memory and network channels are beneficial for small and large messages, respectively.

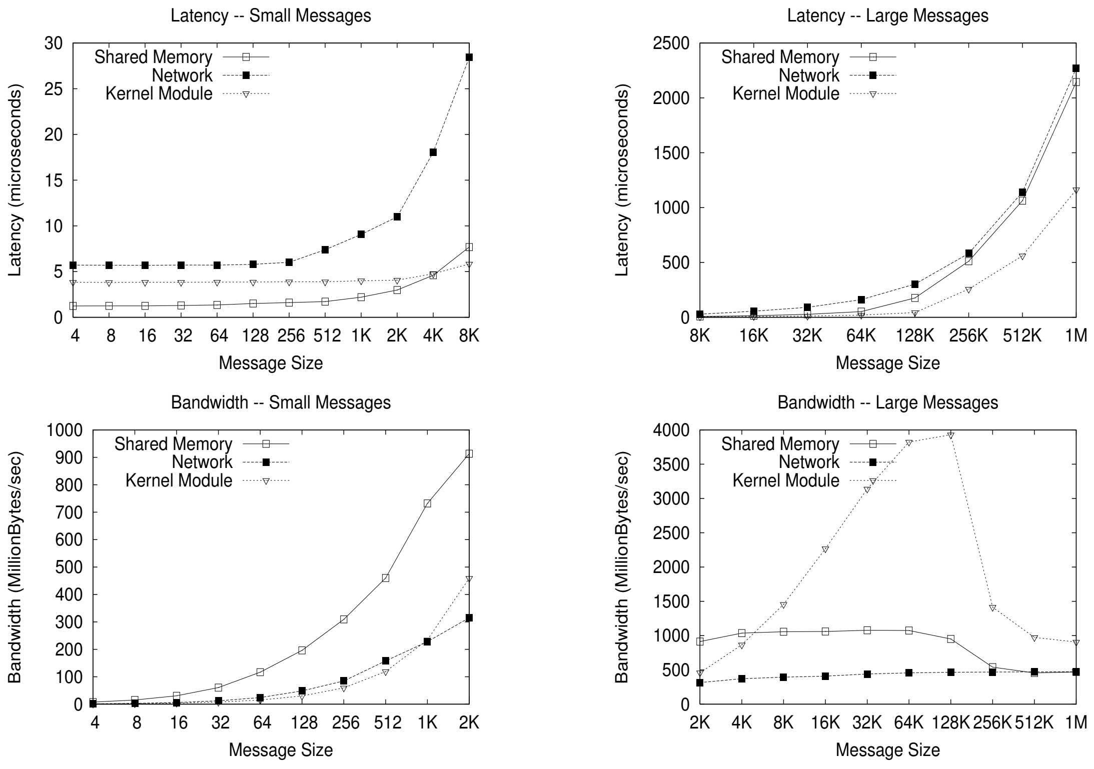
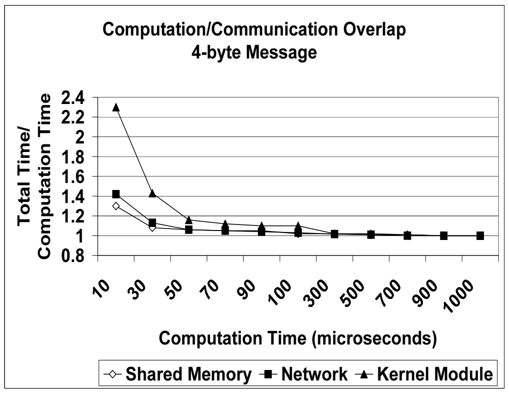


Figure 7.5: Latency and bandwidth comparisons



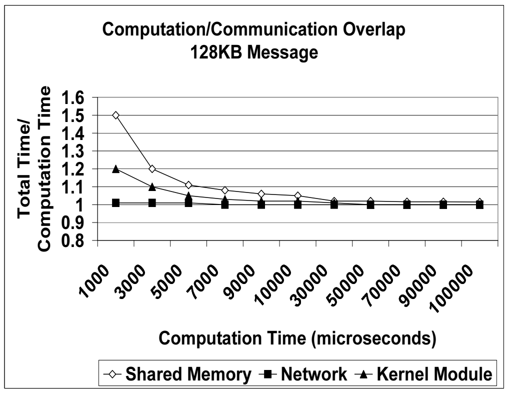


Figure 7.6: Computation/communication overlap

7.4 Summary

In this chapter, we have studied important factors to optimize multi-channel MPI. We have proposed several different schemes for polling communication channels and deciding thresholds for the hybrid of them in MVAPICH. To come up with an efficient static polling scheme, we have taken into account polling overhead and message latency. In addition, we have suggested a dynamic polling scheme, which updates the priority ratio based on update rate and message history. The experimental results show that the factors we have considered affect sensitively on the message discovery time. We note that the static polling scheme can reduce intra-node latency by 12% without hurting inter-node latency. By using the adaptive polling scheme we can reduce the message discovery overhead by 45%.

In addition, we have evaluated thresholds for each channel both based on raw MPI latencies and bandwidths and also CPU utilization. We have observed that kernel module channel can achieve a very low latency and high bandwidth for medium and large messages. On the other hand, for this message range, network channel can overlap computation and communication very well although this channel has a high latency and low bandwidth. For small messages, the shared memory channel shows better performance than others.

CHAPTER 8

OPEN SOURCE SOFTWARE RELEASE AND ITS IMPACT

The work described in this dissertation has been incorporated into our MVAPICH/MVAPICH2 software package and is distributed in an open-source manner. The duration of this work has spanned several release versions of this package, including the latest versions MVAPICH-1.1 and MVAPICH2-1.4. The results presented in this dissertation have reduced intra-node memory usage significantly and enabled MVAPICH/MVAPICH2 to run efficiently on large multi-core systems.

MVAPICH/MVAPICH2 supports many software interfaces, including OpenFabrics [18], uDAPL [34], and InfiniPath-PSM interface from QLogic [22]. The work presented in this dissertation is available in all these interfaces, and is portable across a wide variety of target architectures, like IA32, EM64T, X86_64 and IA64.

Since its release in 2002, more than 855 computing sites and organizations have downloaded this software. More than 27000 downloads have taken place. In addition, nearly every InfiniBand vendor and the Open Source OpenFabrics stack includes this software in their packages. Our software has been used on some of the most powerful computers, as ranked by Top500 [24]. Examples from the November 2008 rankings include *6th*, 62976-core Sun Blade System (Ranger) with Opteron

Quad Core 2.0 GHz at Texas Advanced Computing Center (TACC), 58*th*, 5848-core Dell PowerEdge Intel EM64T 2.66 GHz cluster at Texas Advanced Computing Center/Univ. of Texas, and 73*rd*, 9216-core Appro Quad Opteron dual Core 2.4 GHz at Lawrence Livermore National Laboratory.

CHAPTER 9

CONCLUSIONS AND FUTURE RESEARCH DIRECTIONS

The research in this dissertation has demonstrated the feasibility of running MPI applications efficiently on large multi-core systems with the aid of employing high performance and scalable intra-node communication techniques inside the MPI library. We have described how we can take advantage of shared memory, kernel modules, and on-chip DMAs to design efficient MPI intra-node communication schemes. We have also investigated multi-core aware and multi-channel MPI optimizations. In addition, our work has analyzed application characteristics on multi-core systems, potential bottlenecks, how next-generation MPI applications can be modified to obtain optimal performance, and scalability of multi-core clusters.

9.1 Summary of Research Contributions

The work proposed in this thesis aims towards designing high-performance and scalable MPI intra-node communication middleware, especially for contemporary multi-core systems. The advanced shared memory based approach described in this proposal has already been integrated into MVAPICH software package. MVAPICH is very widely used, including the $6th$ fastest supercomputer in the world: a

62976-core Sun Blade System (Ranger) with Opteron Quad Core 2.0 GHz at Texas Advanced Computing Center (TACC). The design enables applications to execute within a node in a high-performance and scalable manner. The kernel module based approach LiMIC2 has also been integrated into MVAPICH2 distribution.

We note that the ideas proposed and developed in this thesis are independent of any networks and portable across different operating systems. They can essentially be integrated into any MPI library. Thus, we foresee that the contribution of this thesis will be significant for the HPC community, especially as multi-core becomes main stream. Following is a more detailed summary of the research presented in this dissertation.

9.1.1 High Performance and Scalable MPI Intra-node Communication Designs

In Chapters 3, 4, and 5, we have presented several designs for MPI intra-node communication. The shared memory based design has the minimum startup time and administrative requirement, and is portable across different operating systems and platforms. It has shown very good latency and bandwidth. The kernel assisted direct copy approach takes help from the operating system and eliminates the intermediate copies and further improves performance. The I/OAT based approach does not only remove the extra copies but also has better communication and computation overlap. From our experimental results, we have observed that with these advanced designs MPI applications can run efficiently on large multi-core systems.

9.1.2 Multi-core Aware Optimizations

In Chapter 6, we have presented a hybrid approach to get optimized performance on multi-core systems. The approach efficiently combines the shared memory and the kernel assisted direct copy approaches in a topology-aware and skew-aware way. Our performance evaluation shows that the hybrid approach has optimized performance for all intra-node communication cases, namely shared-cache, intra-socket, and inter-socket. It also improves the performance of MPI collective operations and applications.

9.1.3 Comprehensive Analysis of Considerations for Multi-channel MPI

Since most MPI implementations use multiple channels for communication, such as shared memory channel, network channel, kernel module channel etc, it is important to understand and optimize on the factors that affect multi-channel MPI performance. In Chapter 7, we have done this study. We have shown that channel polling and threshold selection are two important factors and proposed efficient channel polling algorithms and threshold selection methods. Our experimental results show that our optimization can improve MPI performance significantly.

9.1.4 In-depth Understanding of Application Behaviors on Multi-core Clusters

In Chapter 2, we have done a comprehensive performance evaluation and analysis on application behaviors on multi-core clusters. Through our study we have found that MPI intra-node communication is very important for the overall performance. We have also observed that cache and memory contention is a potential bottleneck in multi-core systems, and applications should use techniques such as data tiling to avoid cache and memory contention as much as possible. Our scalability study shows that the scalability of multi-core clusters depends on the applications. For applications that are not memory intensive, multi-core clusters have the same scalability as single-core clusters. Our study gives insights to parallel application writers and MPI middleware developers and facilitates them to write code more efficiently for multi-core clusters.

9.2 Future Research Directions

In this dissertation, we have shown the methods to optimize MPI intra-node communication. However, there are several interesting research topics that are still left to be explored.

- **Topology Aware Dynamic Process Distribution** - As described in Section 6, there are multiple levels of communication existing in MPI intra-node communication. For example, there are three levels of communication in Intel Clovertown systems. The first level includes two cores on the same chip and share the L2 cache. The second level includes two cores on the same chip but do not share the L2 cache. And the third level includes two cores on different chips. These different levels of communication have different characteristics, e.g. the latency of the first level communication is the lowest because it just involves cache transactions. Based on the topology information and application characteristics, we can explore the feasibility of dynamic processes migration among physical cores within a node. This may have the potential benefit of minimizing communication overhead. This may be especially important for next-generation many-core systems, such as Intel 80-core system.

- **Efficient MPI Collective Operations** - MPI collective operations are frequently used in many applications, and their performance is critical to the overall performance. This thesis mostly focuses on point-to-point operations and in the future we would like to explore on collective operations too. There are different collective algorithms and they should be chosen based on various factors, such as message size, system size, platforms, etc. With our new designs of point-to-point communication, such as kernel assisted direct copy and I/OAT based design, we need to reconsider the collective algorithms and find out the optimal solution. We might also need to propose new collective algorithms to efficiently utilize the intra-node point-to-point communication schemes.

- **Efficient MPI One-sided Communication** - MPI defines one-sided communication operations that allow users to directly read from or write to the memory of a remote process [61]. One-sided communication both is convenient to use and has the potential to deliver higher performance than regular point-to-point (two-sided) communication. The semantic of one-sided communication matches well with the kernel assisted direct copy approach such as LiMIC/LiMIC2 in the sense that one process can access the memory of another process. In the future, we would like to explore efficient algorithms to use LiMIC/LiMIC2 for MPI one-sided communication operations.

- **Comprehensive Analysis of Intra-node Communication over AMD Barcelona System** - As mentioned in Section 1.1, AMD Barcelona processor is an emerging innovative quad-core architecture. A Barcelona chip includes four cores that have separate L2 cache but share the same L3 cache. The L3 cache is not a traditional inclusive cache, it is acting as a spill-over cache for items evicted by the L2 cache. And when L1 cache loads data from L3 cache (L2 cache is always bypassed) the data can be removed or retained in the L3 cache, depending on whether other cores are likely to access the data in the future. All these features make Barcelona very different from the systems we have studied on. We would like to carry out comprehensive and in-depth performance evaluation on AMD Barcelona systems, and find ways to optimize MPI intra-node communication performance on such systems.

- **Study and Optimizations on Future Multi-core Architectures** - Multi-core technoogy is advancing rapidly. Both Intel and AMD are planning to ship 6/8/12/16-core systems in the near future. In these systems, new architectures are being prososed for better performance and scalability. We will need to carefully study the intra-socket topology and communication characteristics of these new processors and optimize communication performance on them.

BIBLIOGRAPHY

[1] http://lse.sourceforge.net/numa/faq/.

[2] http://valgrind.org/.

[3] AMD's dual-core Opteron processors. http://techreport.com/articles.x/8236.

[4] Cluster (Computing). http://en.wikipedia.org/wiki/Computer_cluster.

[5] HP Message Passing Interface library (HP-MPI). http://h21007.www2.hp.com/portal/site/dspp/menuitem.863c3e4cbcdc3f3515b49c108973a801/?ciid=a308a8ea6ce02110a8ea6ce02110275d6e10RCRD.

[6] InfiniBand Trade Association. http://www.infinibandta.com.

[7] Intel Clovertown Quad Core Processor Review. http://www.maxitmag.com/hardware-reviews/processors/intel-clovertown-quad-core-processor-review.html.

[8] Intel Cluster Toolkit 3.1. http://www.intel.com/cd/software/products/asmo-na/eng/cluster/clustertoolkit/219848.htm.

[9] Intel Dual-core Technology. http://www.intel.com/technology/computing/dual-core/index.htm.

[10] Intel MPI Library 3.2 for Linux or Windows. http://www.intel.com/cd/software/products/asmo-na/eng/cluster/mpi/308295.htm.

[11] Intel Quad-core Technology. http://www.intel.com/technology/quad-core/index.htm.

[12] Intel's Woodcrest processor previewed. http://techreport.com/articles.x/10021/1.

[13] iWARP. http://en.wikipedia.org/wiki/IWARP.

[14] MPI: A Message-Passing Interface Standard. http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html.

[15] MPI over InfiniBand Project. http://nowlab.cse.ohio-state.edu/projects/mpi-iba/.

[16] MPICH2. http://www.mcs.anl.gov/mpi/.

[17] Open MPI : Open Source High Performance Computing. http://www.open-mpi.org.

[18] OpenFabrics Alliance. http://www.openfabrics.org.

[19] OProfile. http://oprofile.sourceforge.net.

[20] Parallel Spectral Transform Shallow Water Model. http://www.csm.ornl.gov/chammp/pstswm/.

[21] Product Brief: Quad-Core AMD Opteron Processor. http://www.amd.com/us-en/Processors/ProductInformation/0,,30_118_8796_15223,00.html.

[22] QLogic. http://www.qlogic.com.

[23] The Cell project at IBM Research. www.research.ibm.com/cell/.

[24] Top 500 SuperComputer Sites. http://www.top500.org/.

[25] UltraSPARC Processors. http://www.sun.com/processors/.

[26] Olivier Aumage and Guillaume Mercier. MPICH/MADIII: a Cluster of Clusters Enabled MPI Implementation. In *3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid2003)*, 2003.

[27] N. J. Boden, D. Cohen, et al. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, pages 29–35, Feb 1995.

[28] Darius Buntinas, Guillaume Mercier, and William Gropp. The Design and Evaluation of Nemesis, a Scalable Low-Latency Message-Passing Communication Subsystem. In *International Symposium on Cluster Computing and the Grid*, 2006.

[29] Thomas W. Burger. Intel Multi-Core Processors: Quick Reference Guide. http://cache-www.intel.com/cd/00/00/23/19/231912_231912.pdf.

[30] L. Chai, A. Hartono, and D. K. Panda. Designing High Performance and Scalable MPI Intra-node Communication Support for Clusters. In *The IEEE International Conference on Cluster Computing*, 2006.

[31] L. Chai, S. Sur, H.-W. Jin, and D. K. Panda. Analysis of Design Considerations for Optimizing Multi-Channel MPI over InfiniBand. In *CAC 2005*, 2005.

[32] G. Ciaccio. Using a Self-connected Gigabit Ethernet Adapter as a memcpy() Low-Overhead Engine for MPI. In *EuroPVM/MPI*, 2003.

[33] Giuseppe Ciaccio. Using a Self-connected Gigabit Ethernet Adapter as a memcpy() Low-Overhead Engine for MPI. In *Euro PVM/MPI*, 2003.

[34] DAT Collaborative. uDAPL: User Direct Access Programming Library Version 1.2. http://www.datcollaborative.org/udapl.html, July 2004.

[35] Rob F. Van der Wijngaart and Haoqiang Jin. NAS Parallel Benchmarks, Multi-Zone Versions. Technical report.

[36] Max Domeika and Lerie Kane. Optimization Techniques for Intel Multi-Core Processors. http://www3.intel.com/cd/ids/developer/asmona/eng/261221.htm?page=1.

[37] Per Ekman and Philip Mucci. Design Considerations for Shared Memory MPI Implementations on Linux NUMA Systems: An MPICH/MPICH2 Case Study. http://www.cs.utk.edu/ mucci/latest/pubs/AMD-MPI-05.pdf.

[38] D. H. Bailey et al. The NAS Parallel Benchmarks. volume 5, pages 63–73, Fall 1991.

[39] Matthew Curtis-Maury et al. An Evaluation of OpenMP on Current and Emerging Multithreaded/Multicore Processors. In *IWOMP*, 2005.

[40] Sadaf R. Alam et al. Characterization of Scientific Workloads on Systems with Multi-Core Processors. In *International Symposium on Workload Characterization*, 2006.

[41] I. Foster and N. T. Karonis. A Grid-Enabled MPI: Message Passing in Heterogenous Distributed Computing Systems. In *Proceedings of the Supercomputing Conference (SC)*, 1998.

[42] Kittur Ganesh. Optimization Techniques for Optimizing Application Performance on Multi-Core Processors. http://tree.celinuxforum.org/CelfPubWiki/ELC2006Present ations?action=AttachFile&do=get&target=Ganesh-CELF.pdf.

[43] P. Geoffray, C. Pham, and B. Tourancheau. A Software Suite for High-Performance Communications on Clusters of SMPs. *Cluster Computing*, 5(4):353–363, October 2002.

[44] Andrew Gover and Christopher Leech. Accelerating Network Receiver Processing. http://linux.inet.hr/files/ols2005/grover-reprint.pdf.

[45] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-Performance, Portable Implementation of the MPI, Message Passing Interface Standard. In *Parallel Computing*, 2006.

[46] Yun He and Chris Ding. Hybrid OpenMP and MPI Programming and Tuning. www.nersc.gov/nusers/services/training/classes/NUG/Jun04 /NUG2004_yhe_hybrid.ppt.

[47] Innovative Computing Laboratory (ICL). HPC Challenge Benchmark. http://icl.cs.utk.edu/hpcc/.

[48] H. W. Jin, S. Sur, L. Chai, and D. K. Panda. Design and Performance Evaluation of LiMIC (Linux Kernel Module for MPI Intra-node Communication) on InfiniBand Cluster. In *International Conference on Parallel Processing*, 2005.

[49] H. W. Jin, S. Sur, L. Chai, and D. K. Panda. Limic: Support for high-performance mpi intra-node communication on linux cluster. In *International Conference on Parallel Processing*, 2005.

[50] 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 *IEEE International Conference on Cluster Computing (poster presentation)*, 2007.

[51] I. Kadayif and M. Kandemir. Data Space-oriented Tiling for Enhancing Locality. *ACM Transactions on Embedded Computing Systems*, 4(2):388–414, 2005.

[52] N. Karonis, B. Toonen, and I. Foster. MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface. *Journal of Parallel and Distributed Computing (JPDC)*, 63(5):551–563, May 2003.

[53] M. Koop, W. Huang, A. Vishnu, and D. K. Panda. Memory Scalability Evaluation of the Next-Generation Intel Bensley Platform with InfiniBand. In *Hot Interconnect*, 2006.

[54] J. Liu, J. Wu, S. P. Kini, P. Wyckoff, and D. K. Panda. High Performance RDMA-Based MPI Implementation over InfiniBand. In *SC*, June 2003.

[55] J. Liu, J. Wu, and D. K. Panda. High performance RDMA-based MPI implementation over InfiniBand. *Int'l Journal of Parallel Programming*, In Press, 2005.

[56] S. S. Lumetta, A. M. Mainwaring, and D. E. Culler. Multi-Protocol Active Messages on a Cluster of SMP's. In *SC '97*, 1997.

[57] S. Makineni and R. Iyer. Architectural Characterization of TCP/IP Packet Processing on the Pentium Microprocessor. In *High Performance Computer Architecture, HPCA-10*, 2004.

[58] Amith R Mamidala, Rahul Kumar, Debraj De, and Dhabaleswar K Panda. MPI Collectives on Modern Multicore Clusters: Performance Optimizations and Communication Characteristics. In *Proceedings of IEEE International Sympsoium on Cluster Computing and the Grid*, 2008.

[59] Mellanox Technologies. Mellanox InfiniBand InfiniHost MT23108 Adapters. http://www.mellanox.com, July 2002.

[60] Mellanox Technologies. Mellanox VAPI Interface, July 2002.

[61] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, Mar 1994.

[62] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*, Jul 1997.

[63] Myricom Inc. Portable MPI Model Implementation over GM, March 2004.

[64] Fabrizio Petrini, Wu chun Feng, Adolfy Hoisie, Salvador Coll, and Eitan Frachtenberg. The Quadrics Network: High Performance Clustering Technology. *IEEE Micro*, 22(1):46–57, January-February 2002. Available from `http://www.c3.lanl.gov/~fabrizio/papers/ieemicro.pdf`.

[65] J. C. Phillips, G. Zheng, S. Kumar, and L. V. Kale. NAMD: Biomolecular Simulation on Thousands of Processors. In *SuperComputing*, 2002.

[66] R. Rabenseifner and A. E. Koniges. The Parallel Communication and I/O Bandwidth Benchmarks: beff and beffio. http://www.hlrs.de/organization/par/services/models/mpi/b_eff/.

[67] R. Recio, P. Culley, D. Garcia, and J. Hillard. IETF Draft: RDMA Protocol Specification, November 2002.

[68] G. Regnier, S. Makineni, R. Illikkal, R. Iyer, D. Minturn, R. Huggahalli, D. Newell, L. Cline, and A. Foong. TCP Onloading for Data Center Servers. In *IEEE Computer*, Nov 2004.

[69] SGI. Message passing toolkit (mpt) user guide.

[70] Suresh Siddha. Multi-core and Linux Kernel. http://oss.intel.com/pdf/mclinux.pdf.

[71] Sun Microsystems Inc. Memory Placement Optimization (MPO). www.opensolaris.org/os/community/performance/ mpo_overview.pdf.

[72] Toshiyuki Takahashi, Shinji Sumimoto nd Atsushi Hori, Hiroshi Harada, and Yutaka Ishikawa. PM2: High Performance Communication Middleware for Heterogeneous Network Environments. In *SuperComputing (SC)*, 2000.

[73] Tian Tian and Chiu-Pi Shih. Software Techniques for Shared-Cache Multi-Core Systems. http://www.intel.com/cd/ids/developer/asmo-na/eng/recent/286311.htm?page=1.

[74] Li Zhao, Ravi Iyer, Srihari Makineni, Laxmi Bhuyan, and Don Newell. Hardware Support for Bulk Data Movement in Server Platforms. In *Proceedings of International Conference on Computer Design*, 2005.