Designing High Performance and Scalable MPI Intra-node Communication Support for Clusters

Lei Chai Albert Hartono Dhabaleswar K. Panda
Department of Computer Science and Engineering
The Ohio State University
chail, hartonoa, panda @cse.ohio-state.edu

Abstract

As new processor and memory architectures advance, clusters start to be built from larger SMP systems, which makes MPI intra-node communication a critical issue in high performance computing. This paper presents a new design for MPI intra-node communication that aims to achieve both high performance and good scalability in a cluster environment. The design distinguishes small and large messages and handles them differently to minimize the data transfer overhead for small messages and the memory space consumed by large messages. Moreover, the design utilizes the cache efficiently and requires no locking mechanisms to achieve optimal performance even with large system size. This paper also explores various optimization strategies to reduce polling overhead and maintain data locality. We have evaluated our design on NUMA and dual core NUMA systems. The experimental results on NUMA system show that the new design can improve MPI intra-node latency by up to 35% and bandwidth by up to 50% compared to MVAPICH. While running the bandwidth benchmark, the measured L2 cache miss rate is reduced by half. The new design also improves the performance of MPI collective calls by up to 25%. The results on dual core NUMA system show that the new design can achieve 0.48 usec in CMP latency.

Keywords: *MPI, Intra-node Communication, Non-Uniform Memory Access (NUMA), Multi-core Processor, Cluster Computing*

1. Introduction

Cluster of workstations is one of the most popular architectures in the arena of high performance computing due to its cost-effectiveness. A cluster is typically built from Symmetric Multi-Processor (SMP) systems connected by high speed networks. Traditional SMP systems depend on a shared memory bus for different processors to access the memory. Due to the scalability concern of this model, most traditional SMP systems have a small number of processors within one node. For example, the most commonly used SMPs are equipped with dual processors. However, with the emerging new technologies such as *Non-Uniform Memory Access (NUMA)* [1] and *Multi-Core Processor* [8] (also called *Chip-level MultiProcessing* or *CMP*), SMP nodes tend to have larger system sizes. 4-way and 8-way SMPs are gaining more and more popularity. SMPs with even more processors are also emerging. The increased system size indicates that a parallel application running on a cluster will have more intensive intra-node communication. With MPI being the most widely used parallel computing paradigm, it is extremely crucial to have a high performance and scalable architecture for MPI intra-node communication.

There are limitations in the current existing 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. In this paper we take on the challenges and design a *user space memory copy* based architecture that aims to improve MPI intra-node communication performance by taking advantage of the advanced features provided by the modern systems. We focus on two types of modern systems - NUMA and multi-core systems. 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

This research is supported in part by Department of Energy's Grant #DE-FC02-01ER25506; National Science Foundation's Grants #CNS-0403342 and #CCR-0509452; grants from Mellanox, Intel, Sun MicroSystems, Cisco, and Linux Networx; and equipment donations from Apple, AMD, Dell, Intel, IBM, Microway, Mellanox, PathScale, SilverStorm, and Sun MicroSystems.

1-4244-0328-6/06/$20.00 C2006 IEEE.

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, and exploiting processor affinity for better data locality.

We have evaluated our design on NUMA and dual core NUMA systems. The experimental results on the NUMA system show that the new design can improve MPI intra-node latency by up to 35% and bandwidth by up to 50% compared to MVAPICH. The L2 cache miss rate when running the bandwidth benchmark is reduced by half. The new design also improves the performance of MPI collective calls by up to 25%. We have also studied the impact of our design on the dual core NUMA system. The results show that the new design can achieve 0.48 usec in CMP latency.

The rest of the paper is organized as the following: In Section 2, we introduce some background knowledge, including advanced system architectures, MPI intra-node communication, and MVAPICH (high performance MPI over InfiniBand). We illustrate our design and the analysis of the design in Section 3. Performance evaluation and comparison are presented in Section 4. Related work is discussed in Section 5. And finally, in Section 6 we conclude and point out future work directions.

2. Background

2.1. Advanced System Architectures

Figure 1 illustrates a typical cluster that is built from SMPs. A parallel application running on a cluster can exchange data among processes through either intra- or inter-node communication. Traditionally, within an SMP node processors access the main memory through a shared bus. In recent years, new architectures are emerging to improve both the performance and scalability of SMP systems. In this section we briefly describe two of the major technologies: Non-Uniform Memory Access (NUMA) and multi-core processor.

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. 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 [18]. Linux also started to be NUMA-aware from 2.6 kernel. In this paper we focus on Linux.

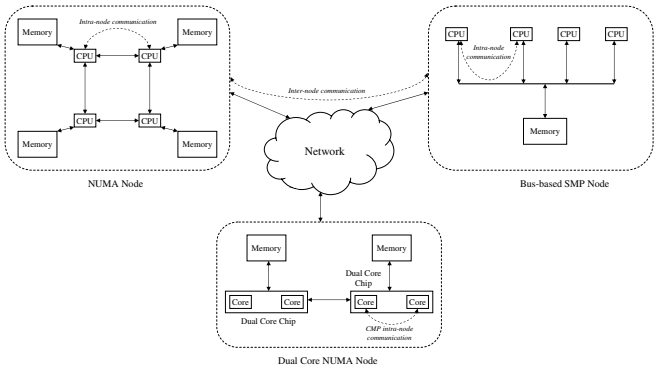


Figure 1. An Illustration of a Cluster Built From SMPs

Multi-core processor technology is proposed to achieve higher performance without driving up power consumption and heat. It is also called *Chip-level MultiProcessing (CMP)*. Both AMD [2] and Intel [3] have multi-core products. In this architecture, two or more processor cores are integrated into a single chip. The processor cores on the same chip can have separate L2 caches, or they can share the same L2 cache. Cache-to-cache transfer between two processors on the same chip is much faster than that between different chips. Therefore, one of the key issues in taking advantage of the multi-core architecture is to efficiently utilize the L2 cache. Of course, efficient cache utilization is essentially a key issue to all kinds of platforms, including NUMA system.

2.2. MPI Intra-node Communication

There exist various mechanisms for MPI intra-node communication. These mechanisms can be broadly classified into three categories, namely *NIC-based loopback, kernel-assisted memory mapping,* and *user space memory copy* [11, 15].

The NIC-based loopback approach does not distinguish intra-node traffic from inter-node traffic. It depends on the NIC to detect the fact that the source and the destination processes are on the same node. The NIC then loopbacks the message instead of injecting it into the network. The kernel-assisted memory mapping approach takes help from the operating system kernel to copy the messages directly from one process' memory space to another. This approach can also deploy the copy-on-write optimization to reduce the number of copies. The third approach, user space memory copy, involves a shared memory region that the processes can attach to and use it as a communication chan-

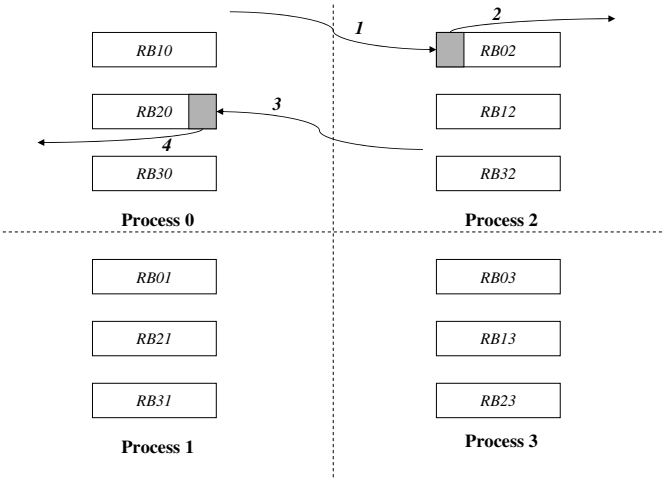


Figure 2. Original Design of MVAPICH Intra-node Communication

nel. The sending process copies the messages to the shared memory region, and the receiving process copies the messages from the shared memory region to its own buffer.

The user space memory copy scheme has several advantages. It provides much higher performance compared to NIC-based loopback. In addition, it is more portable than the kernel-assisted memory mapping approach across different operating systems and versions, because it does not require any service directly from the kernel. Due to these advantages, many MPI implementations choose to use the user space memory copy approach for intra-node communication, such as MVAPICH [5], MPICH-MX [6], and Nemesis [10]. Our design is also based on user space memory copy. The drawback of the user space memory copy approach is that it usually depends on the processor to perform memory copy and can achieve less computation and communication overlap compared with the NIC-based loopback approach.

2.3. MVAPICH

We incorporated our design into MVAPICH [5], a high performance MPI implementation over InfiniBand clusters. The implementation is based on MPICH [14]. MVAPICH is currently being used by more than 340 organizations across 28 different countries to extract the benefits of Infiniband for MPI applications.

Current MVAPICH utilizes a user space shared memory approach for its intra-node communication. 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 2 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*.

The send/receive mechanism is straightforward as illustrated in Figure 2, 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).

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.

3. Proposed Design

In this section, we provide a detailed illustration of our proposed design. 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.

3.1 Overall Architecture

Throughout this paper, 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 (RB)*, one *Send Buffer Pool (SBP)*, and a collection of $P - 1$ *Send Queues (SQ)*. Figure 3 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

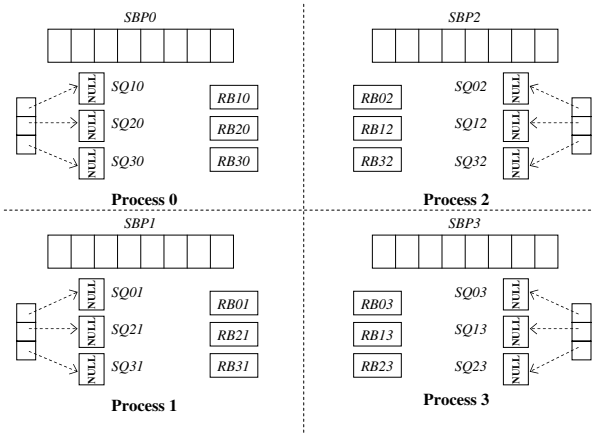


Figure 3. Overall Architecture of the Proposed Design

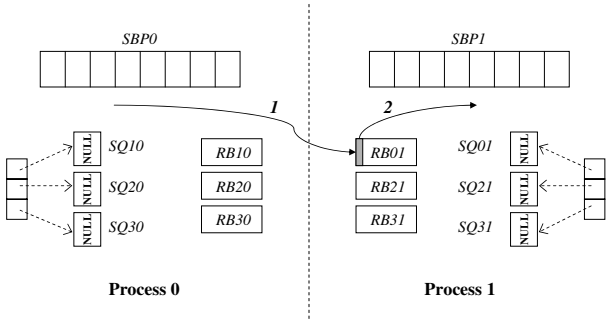


Figure 4. Send/Receive Mechanism for a Small Message

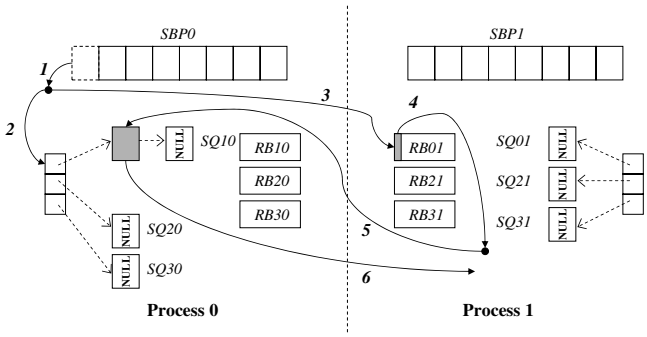


Figure 5. Send/Receive Mechanism for a Large Message

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 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.

3.2.1. Small Message Transfer Procedure. Figure 4 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.

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

3.2.2. Large Message Transfer Procedure. Figure 5 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.

3.3 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.

3.3.1. 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 already explained in Section 3.2.2.

3.3.2. 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.

3.3.3. Efficient Memory Usage. We first illustrate the scalability issue in the current MVAPICH intra-node communication support. As we mentioned in Section 2.3, the current MVAPICH allocates a shared memory region of size $P \quad P \quad 1 \quad BufSize$, 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.

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 $P \quad P \quad 1$ number of receive buffers. Despite such polynomial complexity, the

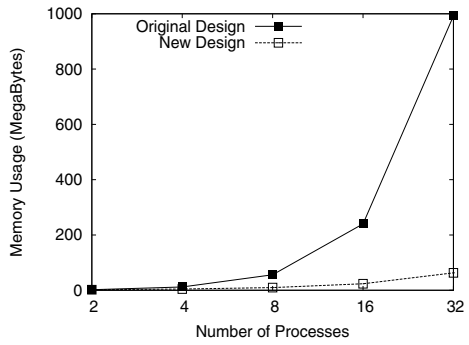


Figure 6. Memory Usage of the Proposed New Design Within a Node

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 MVAPICH 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 6, 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.

3.4 Optimization Strategies

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

3.4.1. 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.

3.4.2. Reducing Indirection Overhead. Utilizing the indirection technique, which is explained in Section 3.2, 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.

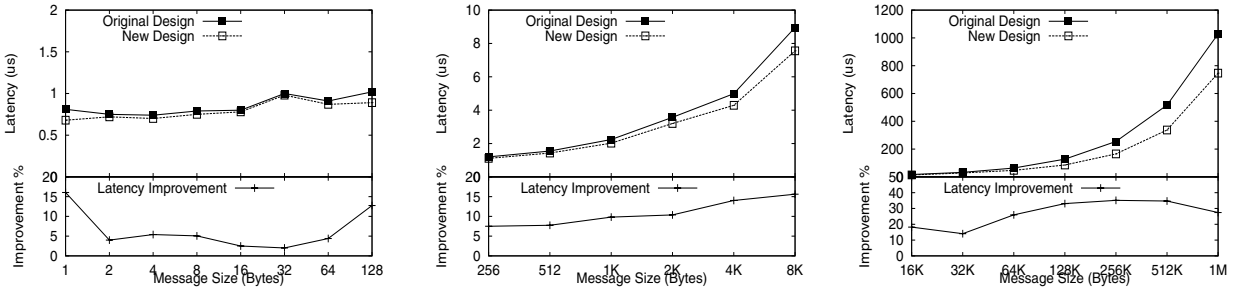
3.4.3. Exploiting Processor Affinity. As we mentioned in Section 2.1, NUMA-aware operating systems always try to allocate memory pages local to processors where they are first referenced. However, the operating system may migrate a process to some other processor at a later stage due to the reason of 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 [12]. 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.

4. Performance Evaluation

In this section, we present the performance evaluation of the proposed intra-node communication design on both NUMA and dual core NUMA clusters.

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 design in MVAPICH as the *Original Design*, and the design proposed in this paper as the *New Design*. Latency is measured in unit of *micro second (us)*, and bandwidth is measured in *million bytes per second (MB/sec)*.



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

Figure 7. Latency on NUMA Cluster

4.1. 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 7 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 8. The peak bandwidth is raised from 1200 MB/sec to 1650 MB/sec.

4.2. L2 Cache Miss Rate

To further analyze the reason of the performance gain presented in Section 4.1, 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* [4], and the benchmarks are the same as used in Section 4.1. The results are shown in Figure 9. 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 3.3.2.

4.3. 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 10 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 11 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.

4.4. 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 12 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 12 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

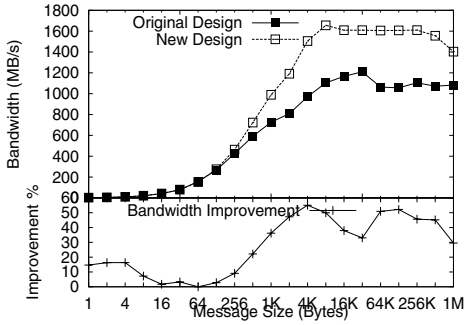


Figure 8. Bandwidth on NUMA Cluster

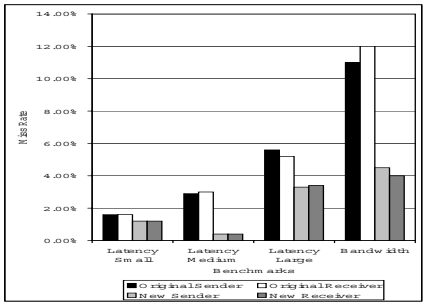


Figure 9. L2 Cache Miss Rate

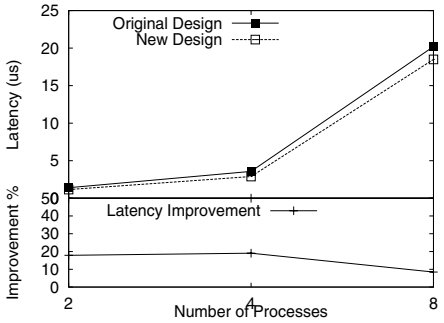


Figure 10. MPI_Barrier Performance

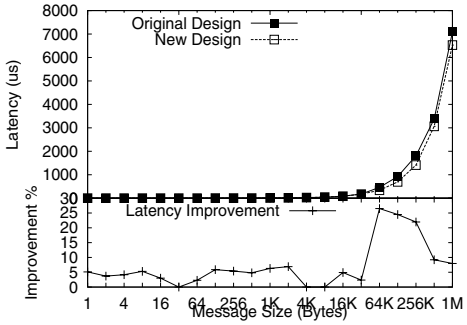
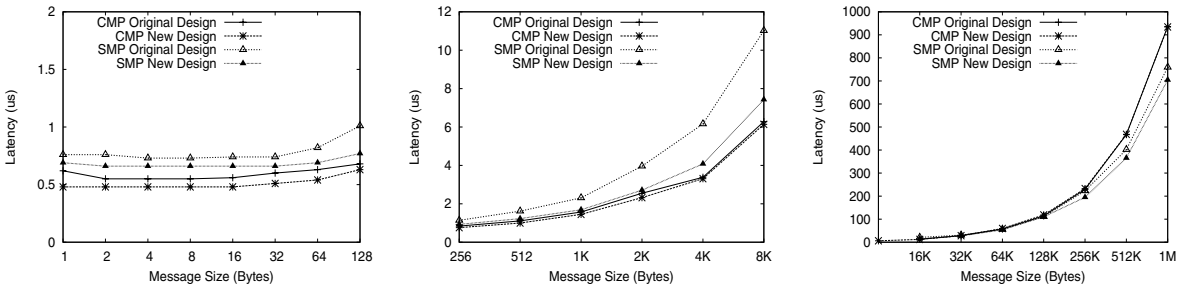


Figure 11. MPI_Alltoall Performance



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

Figure 12. Latency on Dual Core NUMA Cluster

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 12 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 13, indicate the same trend. Again, the new design improves SMP bandwidth for all message sizes, and CMP bandwidth for small and medium messages.

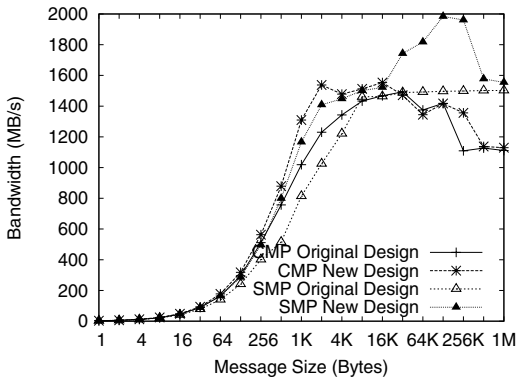


Figure 13. Bandwidth on Dual Core NUMA Cluster

5. Related Work

Some earlier work has been done to improve the performance of intra-node communication in cluster architectures. The work in [9] presents different mechanisms for data transfer in an SMP machine (i.e. copy through shared memory buffer, using message queues, the Ptrace system call, kernel module based copy, and a high speed network card) and has shown the performance evaluations for each technique. The intra-node communication design proposed in [15] makes use of Linux kernels to allow a kernel module to copy messages directly from one process's address space to another. [11] studied the polling overhead associated with MPI intra- and inter-node communication, and proposed an adaptive scheme for efficient polling. In [10], the authors implemented a new message passing communication subsystem for MPICH2 [7], called Nemesis, which makes use of copy through shared message queues as its intra-node data transfer, and uses atomic operations for process synchronizations. Our design model proposed in this paper is different from the Nemesis work in the sense that it transfers messages through both message queues and shared memory buffers, coupled with the indirection technique to avoid the use of locking and atomic operations. Moreover, our proposed design also features efficient cache utilization. In addition to Nemesis, many open source MPI projects have intra-node communication support, such as MPICH-MX [6], MPICH-GM [16], LAM/MPI [17], and Open MPI [13].

6. Conclusions and Future Work

In this paper, we have designed and implemented a high performance and scalable MPI intra-node communication scheme that uses the system cache efficiently, requires no locking mechanisms, and has low memory usage. Our experimental results show that the proposed design can improve MPI intra-node latency by up to 35% compared to MVAPICH 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 also done initial study on the interaction between multi-core systems and MPI. From the experimental results we see that the design proposed in this paper can also improve intra-node communication performance for multi-core systems.

We plan to do application level evaluation on large-scale clusters in the future to study how the new design helps the MPI application performance. We also plan to do more in-depth investigation on MPI intra-node communication design for multi-core systems.

Software Distribution: *The design proposed in this paper will be available for downloading in upcoming MVAPICH releases.*

References

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

[2] http://www.amd.com/.

[3] http://www.intel.com/.

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

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

[6] MPICH-MX Software. http://www.myri.com/scs/download-mpichmx.html.

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

[8] Multicore Processor Technology. http://www.dell.com/downloads/global/power/ps2q05-20050103-Fruehe.pdf.

[9] Darius Buntinas, Guillaume Mercier, and William Gropp. Data Transfers Between Processes in an SMP System: Performance Study and Application to MPI. In *International Conference on Parallel Processing*, 2006.

[10] 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.

[11] Lei Chai, Sayantan Sur, Hyun Wook Jin, and D. K. Panda. Analysis of Design Considerations for Optimizing Multi-Channel MPI over InfiniBand. In *Workshop on Communication Architecture on Clusters*, 2005.

[12] 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.

[13] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J.Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, Concept, and Design of a Next Generation MPI. *Euro PVM/MPI*, September 2004.

[14] 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.

[15] 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.

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

[17] Jeffrey M. Squyres and Andrew Lumsdaine. A Component Architecture for LAM/MPI. In *Proceedings, 10th European PVM/MPI Users' Group Meeting*, number 2840 in Lecture Notes in Computer Science, Venice, Italy, September / October 2003. Springer-Verlag.

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