Designing An Efficient Kernel-level and User-level Hybrid Approach for MPI Intra-node Communication on Multi-core Systems [*]

Lei Chai[1] Ping Lai[1] Hyun-Wook Jin[2]
Dhabaleswar K. Panda[1]

[1]*Department of Computer Science and Engineering*
The Ohio State University
Columbus, OH, USA
{chail, nagaraj, laipi, panda}@cse.ohio-state.edu

[2]*Department of Computer Science and Engineering*
Konkuk University
Seoul, Korea
jinh@konkuk.ac.kr

Abstract

The emergence of multi-core processors has made MPI intra-node communication a critical component in high performance computing. In this paper, we use a three-step methodology to design an efficient MPI intra-node communication scheme from two popular approaches: shared memory and OS kernel-assisted direct copy. We use an Intel quad-core cluster for our study. We first run micro-benchmarks to analyze the advantages and limitations of these two approaches, including the impacts of processor topology, communication buffer reuse, process skew effects, and L2 cache utilization. Based on the results and the analysis, we propose topology-aware and skew-aware thresholds to build an optimized hybrid approach. Finally, we evaluate the impact of the hybrid approach on MPI collective operations and applications using IMB, NAS, PSTSWM, and HPL benchmarks. We observe that the optimized hybrid approach can improve the performance of MPI collective operations by up to 60%, and applications by up to 17%.

1 Introduction

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 [6] supercomputer list published in November 2007, 77% of the sites use multi-core processors from Intel and AMD. Message Passing Interface (MPI) [18] 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.

Traditionally there have been three approaches for MPI intra-node communication: network loopback, user-level shared memory, and kernel assisted direct copy [15]. Network loopback is not commonly used in modern MPI implementations due to its higher latency. Therefore, we do not consider this approach in this paper. User-level shared memory is the most popular approach currently used in many MPI libraries because of its good performance and portability. In this approach, the communication is a two-copy process in which the sending process copies the messages into a shared buffer and the receiving process copies the messages out. Kernel assisted direct copy is a one-copy approach that takes help from the OS kernel and directly copies the messages from sender's buffer to receiver's buffer. It provides good performance by eliminating the intermediate copy. In order to obtain optimized MPI intra-node communication performance, it is important to have a comprehensive understanding of the shared memory and

[*]This research is supported in part by DOE's Grants#DE-FC02-06ER25749 and #DE-FC02-06ER25755; NSF's Grants #CNS-0403342 and #CPA-0702675; grants from Intel, Mellanox, Cisco systems, Linux Networx and Sun Microsystems; and equipment donations from Intel, Mellanox, AMD, Apple, Appro, Dell, Microway, PathScale, IBM, SilverStorm and Sun Microsystems.

kernel-assisted approaches and improve upon them. To achieve this goal, in this paper 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. We use a three-step methodology. We use MVAPICH [2] (shared memory) and MVAPICH-LiMIC2 [16] (kernel-assisted direct copy) as the MPI libraries. We start from micro-benchmarks and study the impacts of processor topology, communication buffer reuse, and process skew effects on these two approaches. We also profile the L2 cache utilization. Based on the experimental results and analysis we then propose topology-aware and skew-aware thresholds to build an efficient hybrid approach. Finally, we evaluate the impact of the hybrid approach on MPI collective operations and applications using IMB, NAS, PSTSWM, and HPL benchmarks. We observe that the hybrid approach can improve the performance of MPI collective operations by up to 60%, and applications by up to 17%.

The rest of the paper is organized as follows: We provide an overview of the background knowledge in Section 2, including multi-core systems, shared memory approach used in MVAPICH, and LiMIC/LiMIC2 approach. The results and analysis of micro-benchmarks are presented in Section 3. Section 4 illustrates the design of an efficient hybrid approach and Section 5 presents the impact of the hybrid design on MPI collective operations and applications. We discuss related work in Section 6. Finally, we conclude this paper and point out future work in Section 7.

2 Overview of Multi-core Processors and MPI Intra-node Communication

In this section, we introduce multi-core processors and MPI intra-node communication approaches. Since network loopback is not commonly used, we focus on the shared memory and kernel-assisted direct copy approaches.

2.1 Multi-core Processors

"Multi-core" means to integrate two or more complete computational cores within a single processor chip. It speeds 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 paper we use *processor*, *CPU*, and *core* interchangeably.

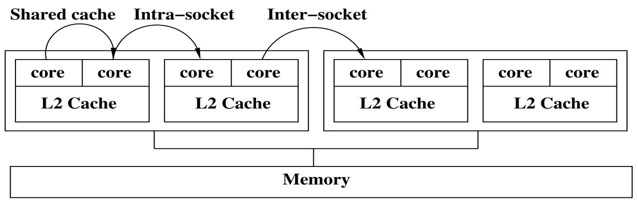


Figure 1. Illustration of Intel Clovertown Processor

There are various alternatives in designing cache hierarchy organization and memory access model for multi-core processors. In this paper, we use an Intel quad-core cluster as the experimental platform. This processor also has the code name Clovertown. Figure 1 illustrates the high-level architecture of the processor. In our system, each node is equipped with two Intel Clovertown sockets. On each socket there are four cores, and two of them share a 4MB L2 cache. Therefore, there are three cases of intra-node communication, and in this paper we refer to them as *shared cache*, *intra-socket*, and *inter-socket*, respectively (See Figure 1).

2.2 Shared Memory Based Approach in MVAPICH

MVAPICH [2] is a high-performance MPI implementation over InfiniBand clusters, used by more than 700 organizations world-wide. MVAPICH currently uses a user-space shared memory approach for intra-node communication.

For small and control messages, each pair of processes has two shared memory buffers, holding messages in each direction. Each send/receive involves two copies. The sending process writes data from its source buffer into the shared buffer. The receiving process copies the data from this shared buffer into its destination buffer. The protocol used for small and control messages is *eager* protocol.

For large messages, each process maintains a pool of fixed sized buffers, which is used by the process to send messages to any other process. There are three benefits of using the shared buffer pool for large messages. First, the pool size does not increase in proportion to the number of processes. Second, the messages can be sent in a pipelined manner. Third, as soon as the data is copied by the receiving process, the buffer can be reused by the sending process, which may improve L2 cache utilization. The protocol used for large messages is *rendezvous* protocol.

The detailed design is described in [13].

2.3 LiMIC/LiMIC2

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 [15] is a stand-alone library that provides MPI-like interfaces, such as LiMIC_send and LiMIC_recv. The second generation, LiMIC2 [16], 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. In this paper, we use MVAPICH-LiMIC2, which integrates MVAPICH with LiMIC2 for intra-node communication.

3 Performance Evaluation and Analysis: Micro-Benchmarks

In this section we study the performance of shared-memory and LiMIC2 approaches using micro-benchmarks. To clearly see the trend, we set the eager threshold to 0 for MVAPICH-LiMIC2 to force all the messages to go through LiMIC2, regardless of their size.

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.

3.1 Impact of Processor Topology

As described in Section 2.1, 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 [2] instead of single-pair because usually all the cores are activated when applications are running. On our system there are 8 cores per node, 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 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 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 2(b) and 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.

3.2 Impact of Buffer Reuse

Figure 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 3.1 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.

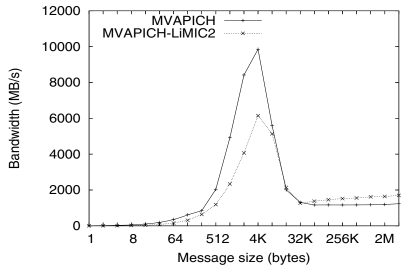
The intra-socket results are shown in Figure 3. The shared cache and inter-socket results follow the same trend. From Figure 3 we can see that the performance of both MVAPICH and MVAPICH-LiMIC2 improves with buffer reuse. This is mainly due to cache effect: starting from the second iteration, the buffers 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 3.3. From the results shown in this section we conclude that applications that have more buffer reuse potentially benefit more from MVAPICH-LiMIC2.

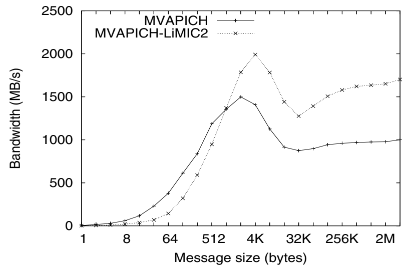
A similar trend can be observed with multi-pair latency test too. The results are not shown here to avoid redundancy.

3.3 L2 Cache Utilization

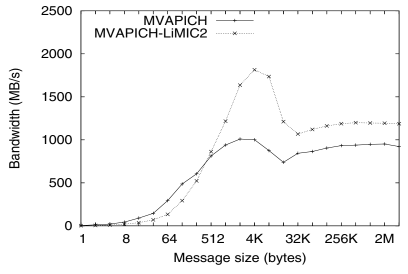
In this section, we analyze the cache effect in the buffer reuse experiment.



(a) Shared Cache



(b) Intra-socket



(c) Inter-socket

Figure 2. Multi-pair Bandwidth

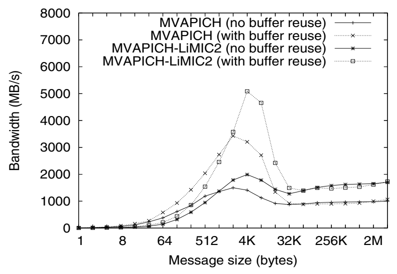


Figure 3. Impact of Buffer Reuse (Intra-socket)

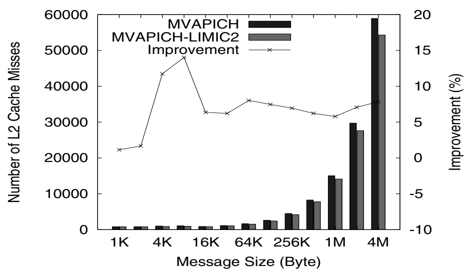


Figure 4. L2 Cache Misses

We use the same benchmark as in Section 3.2, and use *OProfile* [4] 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 4. We start from 1KB since MVAPICH-LiMIC2 shows better performance starting from 1KB in Figure 3. As expected, we see that cache 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 3. This further explains the benefits obtained by MVAPICH-LiMIC2 and demonstrates our conclusion in Section 3.2.

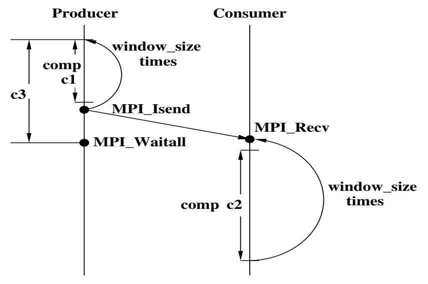


Figure 5. Process Skew Benchmark

3.4 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 we described in Section 2.3, 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. 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, as we described in Section 2.2. 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 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

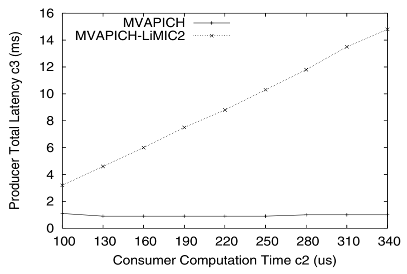


Figure 6. Impact of Process Skew

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 total amount of time that the producer needs to complete this process, shown as *c3* in Figure 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. 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, 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 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 4.2 to alleviate process skew effect.

4 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.

4.1 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 3.1, 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 [19]. 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 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 [2].

Based on the results in Figure 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 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.

4.2 Skew Aware Thresholds

We have seen from Section 3.4 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 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

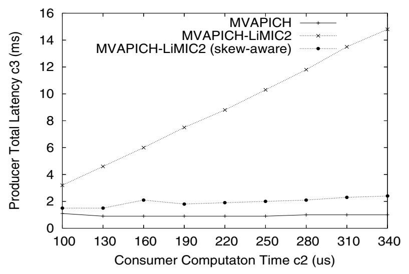


Figure 7. Impact of Skew Aware Thresholds

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 7. We used the same benchmark with the same set of parameters as described in Section 3.4. 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.

5 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) [1] for collectives, and NAS [8], PSTSWM [5] and HPL from HPCC benchmark suite [14] for applications. To better understand the application behaviors and relationship with MPI implementations we have also done profiling to the applications.

5.1 Impact on Collectives

We show the results of three typical collective operations, MPI_Alltoall, MPI_Allgather, and MPI_Allreduce, in Figure 8. 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 [17]. 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.

Table 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

5.2 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 9 and 10 (Class B for NAS and small problem size for PSTSWM). The corresponding message size distribution is shown in Table 1. The cluster-mode results are shown in Figure 11 (Class C for NAS and medium problem size for PSTSWM), in which we use 8 nodes and 8 processes per node (8x8).

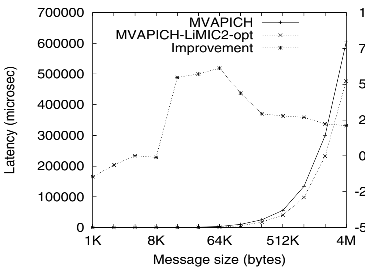
From Figure 9(a) we see that MVAPICH-LiMIC2-opt can improve the performance of FT, PSTSWM, and IS significantly. The improvement is 8% for FT, 14% for PSTSWM, and 17% for IS, respectively. If we look at Figure 10(a) we find that MVAPICH-LiMIC2-opt has better cache utilization for these benchmarks. Most messages used in these benchmarks are large as shown in Table 1. This means that applications that use large messages will potentially benefit from MVAPICH-LiMIC2-opt.

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.

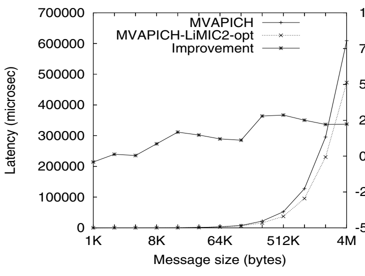
From Figure 11 we see that in cluster mode where there is a mix of intra-node and inter-node communication, applications can still benefit from using MVAPICH-LiMIC2-opt, e.g. PSTSWM performance improves by 6%, which suggests that MVAPICH-LiMIC2-opt is a promising approach for cluster computing.

6 Related Work

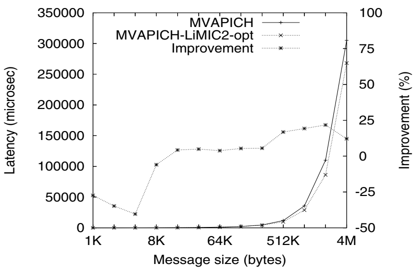
Buntinas et al. have evaluated five data transfer methods between processes and their applications to MPI on a single-



(a) MPI_Alltoall

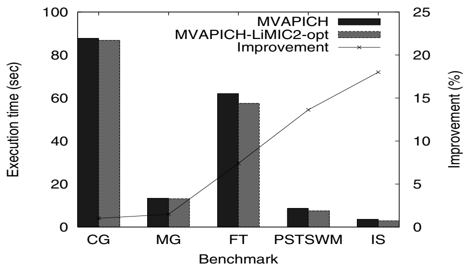


(b) MPI_Allgather

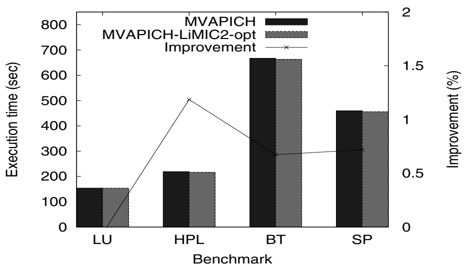


(c) MPI_Allreduce

Figure 8. Collective Results (Single Node 1x8)



(a)



(b)

Figure 9. Application Performance (Single Node 1x8)

core based 2-way SMP system [9]. However, in this paper, we focus on the shared memory and the OS kernel-assisted direct copy approaches on a multi-core cluster. In addition to performance comparison and analysis we have proposed optimizations to build a hybrid approach and evaluated its benefits on applications.

As multi-core processors emerge, research has been done to understand the application performance on multi-core systems. Work done in [7] presents a scientific workloads characterization on AMD Opteron based multi-core systems. In our previous work [12], we have done a case study on an Intel dual-core cluster that analyzes impact of multi-core architecture on high performance computing.

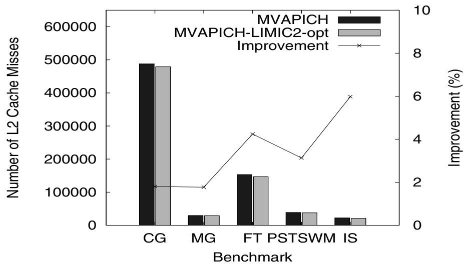
Researchers have explored OS kernel-assisted approaches for MPI intra-node communication. Besides LiMIC and LiMIC2 discussed in Section 2.3, there are Kaput [11] and I/OAT based approaches [21, 20].

Various MPI implementations besides MVAPICH use the shared memory approach for intra-node communication, such as MPICH2 Nemesis [10], MPICH-MX [3], etc.

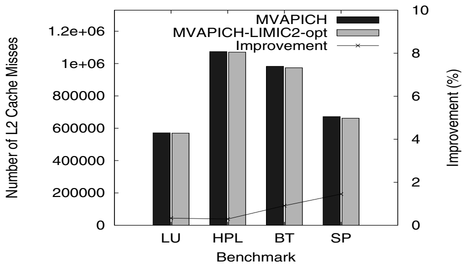
7 Conclusions and Future Work

In this paper, 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. 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 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.

In the future we plan to study on algorithms to optimize LiMIC2 based collective operations. In addition, we plan to do evaluation and analysis on other platforms, such as AMD dual-core and quad-core systems. We also plan to do in-depth studies on how the improvements in MPI intra-node communication benefit the application performance.



(a)



(b)

Figure 10. L2 Cache Misses in Applications (Single Node 1x8)

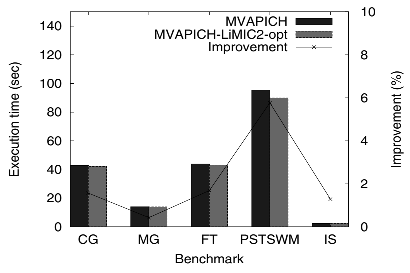


Figure 11. Application Performance on 8 Nodes (8x8)

Acknowledgements

We would like to thank Deepak Nagaraj for carrying out some of the experiments and participating in the discussions.

References

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

[2] MPI over InfiniBand and iWARP Project. http://mvapich.cse.ohio-state.edu.

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

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

[5] PSTSWM. http://www.csm.ornl.gov/chammp/pstswm/.

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

[7] S. R. Alam, R. F. Barrett, J. A. Kuehn, P. C. Roth, and J. S. Vetter. Characterization of Scientific Workloads on Systems with Multi-Core Processors. In *International Symposium on Workload Characterization*, 2006.

[8] D. H. Bailey, L. Dagum, E. Barszcz, and H. D. Simon. The NAS parallel benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

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

[10] D. Buntinas, G. Mercier, and W. 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] P. Carns. A Kernel Module for Copying Data Between Processes, 2004.

[12] 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. In *Proceedings of IEEE International Sympsoium on Cluster Computing and the Grid*, 2007.

[13] 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.

[14] Innovative Computing Laboratory. HPC Challenge Benchmark. http://icl.cs.utk.edu/hpcc/.

[15] 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.

[16] H.-W. Jin, S. Sur, L. Chai, and D. 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.

[17] A. R. Mamidala, R. Kumar, D. De, and D. 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.

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

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

[20] K. Vaidyanathan, L. Chai, W. Huang, and D. K. Panda. Efficient Asynchronous Memory Copy Operations on Multi-Core Systems and I/OAT. In *IEEE International Conference on Cluster Computing*, 2007.

[21] K. Vaidyanathan, W. Huang, L. Chai, and D. K. Panda. Designing Efficient Asynchronous Memory Operations Using Hardware Copy Engine: A Case Study with I/OAT. In *International Workshop on Communication Architecture for Clusters*, 2007.