LiMIC: Support for High-Performance MPI Intra-Node Communication on Linux Cluster [*]

Hyun-Wook Jin Sayantan Sur Lei Chai Dhabaleswar K. Panda

Department of Computer Science and Engineering
The Ohio State University
{jinhy, surs, chail, panda}@cse.ohio-state.edu

Abstract

High performance intra-node communication support for MPI applications is critical for achieving best performance from clusters of SMP workstations. Present day MPI stacks cannot make use of operating system kernel support for intra-node communication. This is primarily due to the lack of an efficient, portable, stable and MPI friendly interface to access the kernel functions. In this paper we attempt to address design challenges for implementing such a high performance and portable kernel module interface. We implement a kernel module interface called LiMIC and integrate it with MVAPICH, an open source MPI over InfiniBand. Our performance evaluation reveals that the point-to-point latency can be reduced by 71% and the bandwidth improved by 405% for 64KB message size. In addition, LiMIC can improve HPCC Effective Bandwidth and NAS IS class B benchmarks by 12% and 8%, respectively, on an 8-node dual SMP InfiniBand cluster.

1. Introduction

Cluster based computing systems are becoming popular for a wide range of scientific applications owing to their cost-effectiveness. These systems are typically built from Symmetric Multi-Processor (SMP) nodes connected with high speed Local Area Networks (LANs) or System Area Networks (SANs) [6]. A majority of these scientific applications are written on top of Message Passing Interface (MPI) [2]. Even though high performance networks have evolved and have very low latency, intra-node communication still remains order of magnitudes faster than network. In order to fully exploit this, MPI applications usually run a set of processes on the same physical node.

To provide high performance to MPI applications, an efficient implementation of intra-node message passing becomes critical. Although several MPI implementations [3, 12] provide intra-node communication support, the performance offered is not optimal. This is mainly due to several message copies involved in the intra-node message passing. 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 from one process to another. While some researchers have suggested this approach [7, 15, 12], their efforts fall short because of several design limitations and the lack of portability.

In this paper, 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 an MPI friendly interface. 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).

InfiniBand [1] is a high-performance interconnect based on open standards. MVAPICH [3] is a high-performance implementation of MPI over InfiniBand. MVAPICH is based on the Abstract Device Layer of MPICH [8]. To evaluate the impact of LiMIC, we have integrated it into MVAPICH. Our performance evaluation reveals that we can achieve a *405%* benefit in bandwidth and *71%* improvement in latency for 64KB message size. In addition, we achieve an overall improvement of 12% with HPCC Effective Bandwidth on an 8-node InfiniBand cluster. Further, our application level evaluation with the NAS benchmarks, Integer Sort, reveals a performance benefit of 10%, 8%, and 5% executing classes A, B, and C, respectively, on an 8-node cluster.

The rest of this paper is organized as follows: Section 2 describes existing mechanisms for intra-node communication. In Section 3, we discuss limitations of previous kernel-based approaches and suggest our solution, LiMIC. Then we discuss the design challenges and implementation issues of LiMIC in Section 4. We present performance evaluation results in Section 5. Finally, this paper concludes in Section 6.

∗ This research is supported in part by Department of Energy's grant #DE-FC02-01ER25506, National Science Foundation's grants #CCR-0204429 and #CCR-0311542 and a grant from Mellanox, Inc.

2. Existing Intra-Node Communication Mechanisms

2.1. NIC-Level Loopback

An intelligent NIC can provide a NIC-level 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(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 [11] 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.

2.2. 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. 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.

Figure 1(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.

This shared memory based design has been used in MPICH-GM [12] and other MPI implementations such as MVAPICH [3]. In addition, Lumetta et al. [10] have dealt with efficient design of shared memory message passing protocol and multiprotocol implementation.

3. Kernel-Based Solution, Its Limitations, and Our Approach

3.1. 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(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.

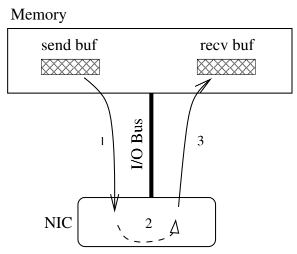
It is to be noted that the number of possible memory transactions for the Kernel-based memory mapping is always less than the number in User-space shared memory approach. We also note that due to the reduced number of copies to and from various buffers, we can maximize the cache utilization. However, 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.

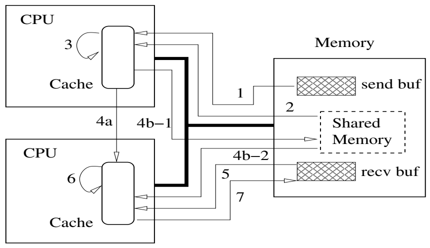
There are several previous works that adopt this approach, which include [7, 15]. However, their designs lack portability across different networks and deny flexibility to the MPI library developer. To the best of our knowledge, no other current generation MPI implementations provide such a kernel support.

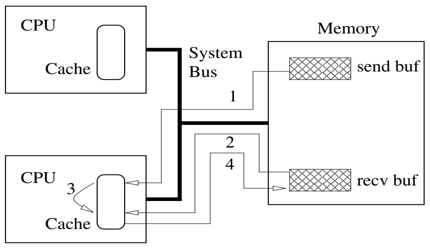
3.2. Our Approach: LiMIC

It is to be noted that the kernel-based approach has the potential to provide efficient MPI intra-node communication. In this paper 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.

Traditionally, researchers have explored kernel based approaches as an extension to the features available in user-



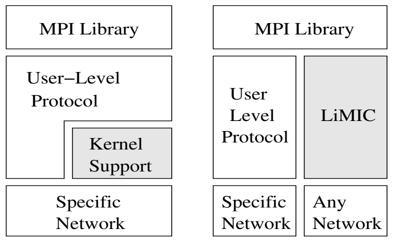




(a) NIC-Level Loopback (b) User-Space Shared Memory (c) Kernel-Based Memory Mapping

Figure 1. Memory Transactions for Different Intra-Node Communication Schemes

level protocols. A high level description of these earlier methodologies is shown in Figure 2(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.



(a) Earlier Design Approach (b) LiMIC Design Approach

Figure 2. Kernel Support Design Approaches

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 2(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 [5]. As a result, LiMIC can provide portability on different interconnects and flexibility for MPI performance optimization.

4. Design and Implementation Issues

4.1. 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 3.2, 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. 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 5) 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.

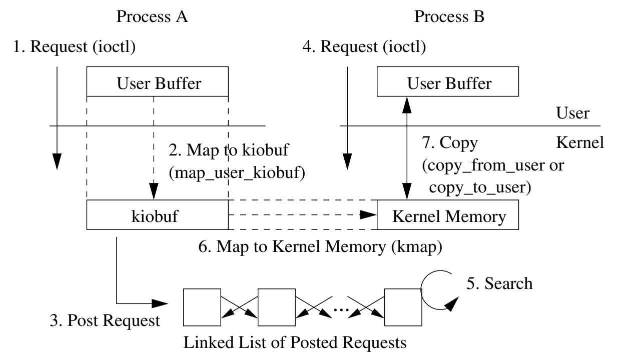

Process A
Process B
1. Request (ioctl)
4. Request (ioctl)
User Buffer
User Buffer
User
Kernel
2. Map to kiobuf (map_user_kiobuf)
7. Copy (copy_from_user or copy_to_user)
kiobuf
Kernel Memory
6. Map to Kernel Memory (kmap)
5. Search
3. Post Request
Linked List of Posted Requests

Figure 3. Memory Mapping Mechanism

Figure 3 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.

4.3. 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.

In this paper 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.

4.4. 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 [13]. The LiMIC might need an extended interface for such MPI implementations while most of MPI implementations can easily employ LiMIC.

5. Performance Evaluation

In this section we evaluate various performance characteristics of LiMIC. As described in section 2, there are various design alternatives to implement efficient intra-node message passing. MVAPICH [3] 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 5.1. However, it is to be noted that each application can set a different threshold as discussed in Section 3.2. 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.

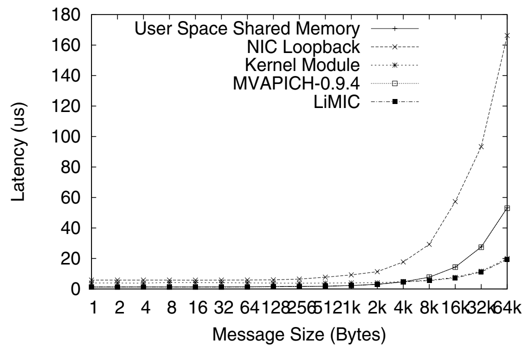
First, we evaluate our designs at microbenchmarks level. Second, we present experimental results on message transfer and descriptor post breakdown. Then we evaluate the scalability of performance offered by LiMIC for larger clusters. Finally, we evaluate the impact of LiMIC on NAS Integer Sort application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

5.1. Microbenchmarks

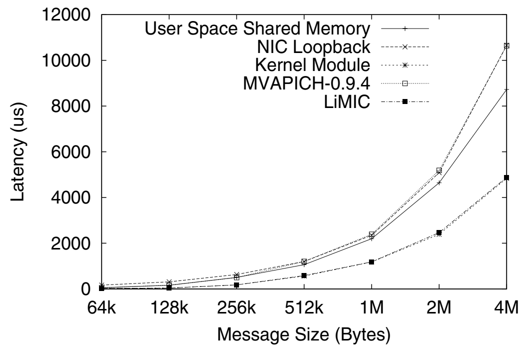
In this section, we describe our tests for microbenchmarks such as point-to-point latency and bandwidth. The tests were conducted on Cluster A.

The latency test is carried out in a standard ping-pong fashion. The latency microbenchmark is available from [3]. The results for one-way latency is shown in Figures 4(a) and 4(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.

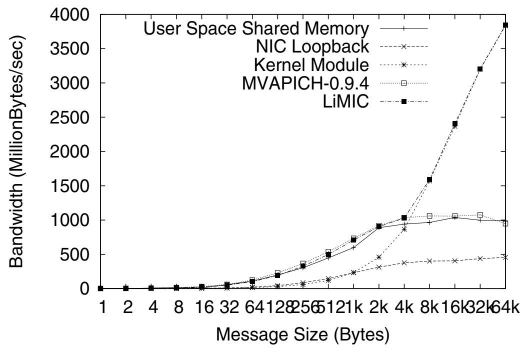
For measuring the point-to-point bandwidth, a simple window based communication approach was used. The bandwidth microbenchmark is available from [3]. The bandwidth graphs are shown in Figures 4(c) and 4(d). We observe an improvement of 405% for bandwidth for 64KB message size as compared to MVAPICH-0.9.4. 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.



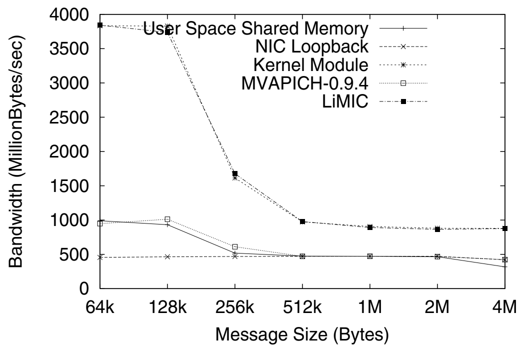
(a) Small Message Latency



(b) Large Message Latency

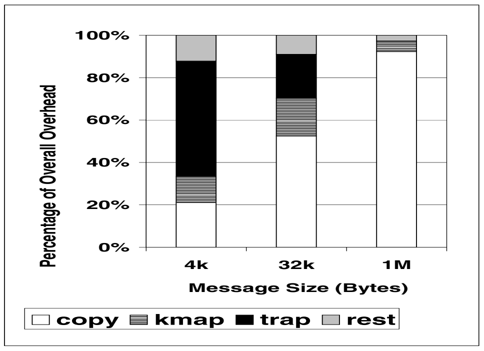


(c) Small Message Bandwidth

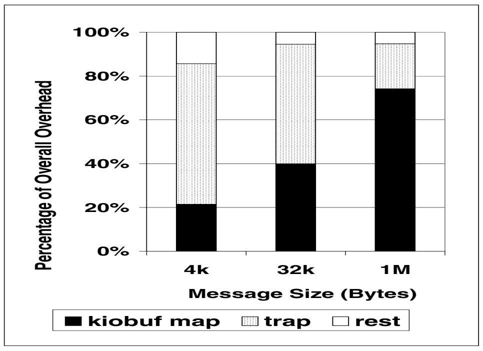


(d) Large Message Bandwidth

Figure 4. MPI Level Latency and Bandwidth



(a) Message Transfer Breakdown



(b) Descriptor Post Breakdown

Figure 5. LiMIC Cost Breakdown (Percentage of Overall Overhead)

5.2. 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 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 5(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 5(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 5(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.

5.3. 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 [14] 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) [9].

Table 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 messages 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 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%

5.4. NAS Integer Sort

We conducted performance evaluation of LiMIC on IS in NAS Parallel Benchmark suite [4] 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 6. 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 7.

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 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 7.

6. Conclusions and Future Work

In this paper 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.

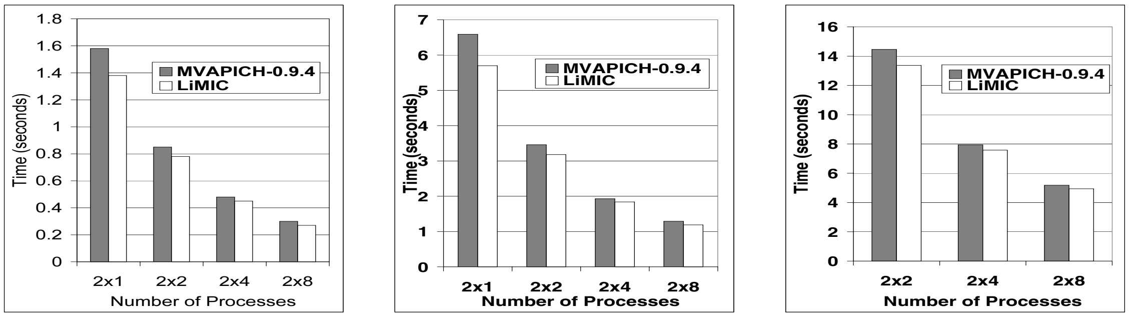


Figure 6. IS Total Execution Time Comparisons: (a) Class A, (b) Class B, and (c) Class C

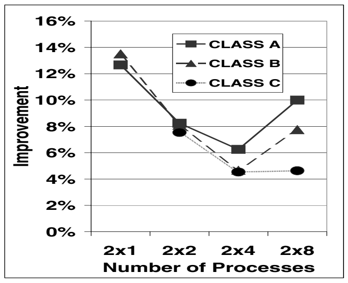


Figure 7. IS Performance Improvement

Table 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

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 effective bandwidth by 12%. Also, our experiments on a larger 16-node cluster revealed that the improvement in 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.

As future work, we plan to enhance LiMIC to achieve zero-copy intra-node communication by using the copy-on-write and memory mapping mechanisms.

References

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

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

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

[4] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[5] 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.

[6] J. Duato, S. Yalamanchili, and L. Ni. *Interconnection Networks: An Engineering Approach*. The IEEE Computer Society Press, 1997.

[7] 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.

[8] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. High-performance, portable implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, 1996.

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

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

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

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

[13] F. Petrini, W. Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The Quadrics Network: High Performance Clustering Technology. *IEEE Micro*, 22(1):46–57, January-February 2002.

[14] 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/.

[15] T. Takahashi, S. Sumimoto, A. Hori, H. Harada, and Y. Ishikawa. PM2: High Performance Communication Middleware for Heterogeneous Network Environments. In *SC 2000*, 2000.