Design and Performance Evaluation of LiMIC (Linux Kernel Module for MPI Intra-node Communication) on InfiniBand Cluster *

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

Department of Computer Science and Engineering
The Ohio State University
Columbus, OH 43210
{jinhy,surs,chail,panda}@cse.ohio-state.edu

Abstract

High performance intra-node communication support for MPI applications is critical for achieving the best performance out of clusters of SMP workstations. Although the performance of system area networks has improved in the recent years, intra-node communication still remains orders of magnitude faster than the network. 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 72% and the bandwidth improved by 405% for 64KB message size. In addition, LiMIC can enhance the performance of NAS IS Class B benchmark by up to 13.5%.

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) [8]. A majority of these scientific applications are written on top of the Message Passing Interface (MPI) [15]. Even though the high performance networks have evolved and have very low latencies, below 5μs, intra-node communication still remains order of magnitudes faster than the 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 the MPI applications, an efficient implementation of intra-node message passing becomes critical. Although several MPI implementations [3][16] provide intra-node communication support, the performance offerred is not optimal. This is mainly due to several message copies involved in the process of intra-node message passing. Every process has its own virtual address space and cannot directly access another process's message buffers. Consequently, explicit shared memory is used in message passing, thus introducing copies. One approach to avoid extra message copies is to use the Operating System kernel to provide a direct copy from one process to another. While some researchers have suggested this approach [17][11][19][18][16], their efforts fall short because of several design limitations and the lack of portability. Accordingly, the current generation high performance MPI implementations do not have the Operating System kernel support for high performance intra-node message passing.

In this paper, we 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 a any communication library or interconnection network and offers

*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.

portability across the Linux kernels. We call this kernel module as LiMIC (**Li**nux kernel module for **MPI** **I**ntra-node **C**ommunication).

InfiniBand [1] is a high-performance interconnect based on open standards. It offers very low message latency, less than 5 μs and high bandwidth of 880 Million Bytes per second. MVAPICH [3] is an implementation of MPI over InfiniBand. MVAPICH is based on the Abstract Device Layer of MPICH [12]. InfiniBand network cards support efficient intra-node message passing. To evaluate the impact of our design of LiMIC, we integrate it into MVAPICH. Our performance evaluation reveals that for point-to-point operations, we can achieve a *405%* benefit in bandwidth and *72%* improvement in latency for 64KB message size. Further, our application level evaluation with the NAS benchmarks [6], Integer Sort, reveals a performance benefit of up to 12.66% and 13.5% executing Class A and B respectively.

The rest of this paper organized as follows: Section 2 describes design alternatives for intra-node communication. We propose a kernel based memory mapping scheme in Section 3. Then we detail design challenges and implementation issues of LiMIC in Section 4 and present its performance evaluation results in Section 5. Section 6 addresses related works. Finally, this paper concludes in Section 7.

2 Existing Approaches

In this section we detail some of the existing approaches in current generation MPI implementations. In addition, we describe the relative benefits and drawbacks of each alternative.

2.1 NIC-Based Message Loopback

An intelligent NIC can provide a NIC based loopback. When a message transfer is initiated, the NIC can detect whether the destination is the same physical node or not. By intiating a local DMA from the NIC memory back to the host memory, we can gain achieve higher performance. Firstly, we can reduce the overhead introduced by the network components like switches. Secondly, we can avoid flooding the network by unecessary messages. Thirdly, we can save the CRC checks which are usually done on packets which are received over the network. However, there still exist two DMA operations. Although I/O buses such as PCI are getting faster, the DMA overhead for small messages is significantly high compared to User-Space Shared Memory based implementation described in Section 2.2. Additionally, the bandwith of the I/O bus is shared between the copy-out and the copy-in operations, hence this alternative can give us only half the bandwidth available for the I/O bus. Further, the DMA operations cannot utilize the cache effect for small or medium messages. Hence, this approach does not provide good benefits for smaller message sizes. Figure 1 illustrates this scheme.

InfiniHost[14] is a Mellanox's[2] second generation InfiniBand(IB) Host Channel Adapter (HCA). It provides internal loopback for packets transmitted between two Queue Pairs (connections) that are assigned to the same HCA port.

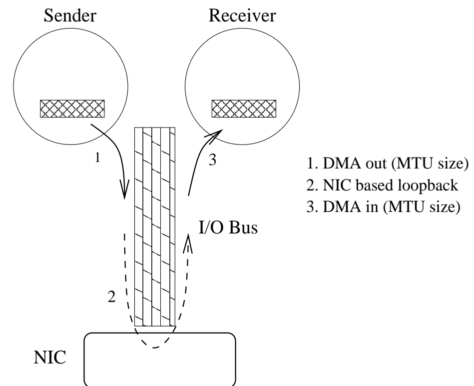


Figure 1. NIC based message loopback

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 and other control information. This shared memory based design has been used in MPICH-GM[16] and other MPI implementations such as MVAPICH over InfiniBand [3]. The sending process copies the message along with other information required for message matching to the shared memory area. The receiving process can then match the tags of the posted receives and accordingly copy over the correct message to its own buffer. We note that this approach involves minimal setup overhead for every message exchange. However, there are at least two copies involved in the message exchange. This approach might tie down the CPU with memory copy time. This method is feasible for shorter messages, where the send, receive and shared buffer are probably present in the cache. But as the size of the message grows, the performance deteriorates. Such vigorous copy-in and copy-out also destroys the cache contents for the end MPI application. Figure 2 shows the various memory transactions which happen during the message transfer. We note that for transfer-

ring large messages (greater than data cache size), we can safely assume that application buffers and shared buffers are out of cache. 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 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 operations 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 cache block replacement scheme, 7 might occur.

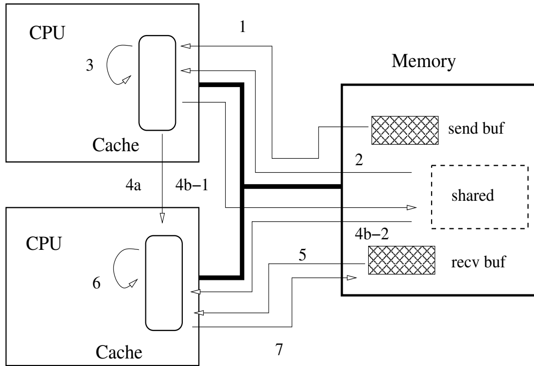


Figure 2. User-space shared memory

In addition, this approach is prone to skew between applications. If the receiving process arrives earlier than sending process, it returns to the application after posting the receive. However, the message is not copied into the receive buffer until the receiving process arrives at any other MPI call. We note that this is not the only case in which skew can affect message transfer progress.

3 Proposed Kernel-Based Memory Mapping Scheme

This section describes the data path of direct memory copy by a kernel function. Earlier attempts at such an approach are described in Section 6, however their designs lacked portability across various interconnects and different communication libraries. Our design principles and details of this approach are described in Section 4.

Kernel-Based Memory Mapping approach takes help from the operating system kernel to copy messages directly from one user process to another. The sender or the receiver process posts the message request descriptor in a message queue indicating its virtual address, tags etc. This memory is then mapped into the kernel address space. When the other process (either sender or receiver) arrives at the message exchange point, the message descriptors are matched at the kernel level and the kernel performs a direct copy from the sender buffer to the receiver application buffer. Thus this approach involves only one copy. This approach reduces the number of copies. However, there are other overheads. The overheads include time to trap into the kernel, TLB flush and locking of other kernel data structures. In addition, one CPU is still required to perform the copy operation. We note that these overheads are relatively less as compared to typical message transfer latencies of medium and large messages. Figure 3 demonstrates the memory transactions needed for copying from the sender buffer directly to the receiver buffer. In step 1, the receiving process needs to bring the sending process' buffer into its cache block. Then in step 3, the receiving process can write this buffer into its own receive buffer. This may generate step 2 based on whether the block was in cache already or not. Then, depending on the cache block replacement policy, step 4 might be generated implicitly.

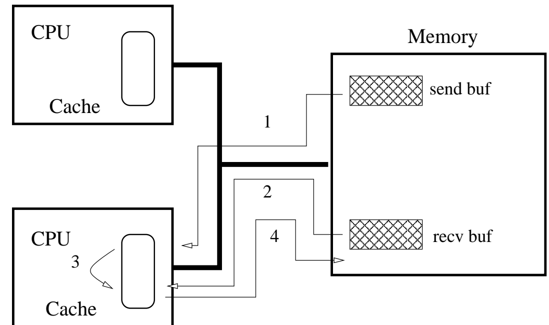


Figure 3. Kernel-Based memory mapping

It is noted to be that the number of possible memory transactions for the Kernel-Based Memory Mapping is always lesser than the number in User-Space Shared Memory approach. In addition, this approach is not prone to skew between applications. Regardless of either the sender or the receiver arriving later, the message copy is performed as soon as a message match is made. We also note that due to the reduced number of copies to and from various buffers, we can maximize the cache utilization.

It is to be noted that the Kernel-Based Memory Mapping approach has the potential to provide good benefits. In this paper we design and implement this method of intra-node message passing. We also try to generalize our approach and address portability is-

sues by implementing this scheme as a runtime loadable kernel module which does not require any kernel modifications.

4 Design Challenges and Implementation Issues

4.1 LiMIC : Portable and MPI Friendly Interface

In this section we describe in detail our design approach in coming up with a new kernel module interface for intra-node message passing. This section sharply distinguishes our approach and design philosophy from earlier research in this direction.

Traditionally, researchers have explored kernel based approaches as an extension to the features available in user-level communication libraries. As a result, most of this work has been non-portable to other user-level communication libraries or other MPI implementations. In addition, these earlier designs did not take into account MPI message matching semantics and message queues. Further, the MPI library blindly calls routines provided by the communication library. Since, some of the communication libraries are proprietary, this mechanism denies any sort of optimization-space for the MPI library developer. A high level description of these earlier methodologies is shown in Figure 4.

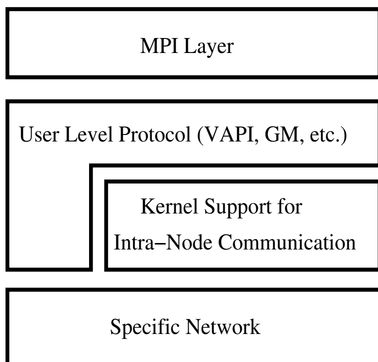


Figure 4. Earlier Design Approaches

In order to avoid the limitations of the past approaches we look towards generalizing the kernel-access interface and making it MPI friendly. We also note that such a design is readily portable across different interconnects. 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. A high level diagram showing our approach is shown in Figure 5. Our implementation of this interface is called LiMIC (**Li**nux kernel module for **M**PI **I**ntra-node **C**ommunication).

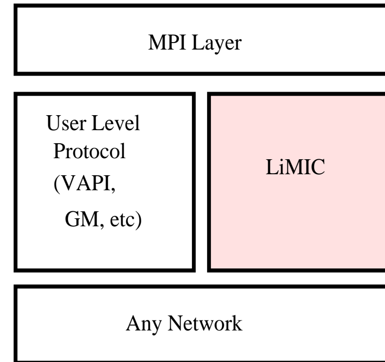


Figure 5. LiMIC Design Approaches

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 just 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. An entry is placed in the LiMIC message queue.

- `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 much like the `LiMIC_Isend` call. It is to be noted that either the sender or the receiver, whichever arrives later will have the message buffer copied to the destination buffer. Also, blocking send and receive can be easily implemented over non-blocking primitives and just blocking on `LiMIC_Wait`.

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

4.2 Memory Mapping Mechanism

This section describes in detail the design of the memory mapping mechanism in LiMIC. We discuss various issues to achieve efficient memory mapping keeping in mind of the overall design goals of LiMIC.

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 the 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 the memory mapping, we use the `kiobuf` provided by the Linux kernel since version 2.3.12. The `kiobuf` supports the abstraction that hides the complexity of the virtual memory system from device drivers. This was originally designed for I/O and other 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. Also it allows us to make a mapping between `kiobuf` and physical memory of user buffer. In addition, since the `kiobuf` internally takes care of pinning down the memory area, or locking of pages in physical memory, we can easily gurantee that the user buffer is present in the physical memory when another process tries to access it. Further, we can avoid many unstability issues come from using memory mapping functions because we are utilizing kiobuf abstraction to handle this. 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 of the kernel module. The `kiobuf` functions provide a way to map between `kiobuf` and physical memory 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 (description 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` structure pool. This pool is shared between all processes running on the same node until the kernel module is removed from the system. Further, the option to allocate additional `kiobuf`s upon exhaustion of the pre-allocated pool is provided to the MPI library.

Figure 6 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 maps user buffer to the `kiobuf` by calling `map_user_kiobuf()` (Step 2). Then, the kernel module puts this request in a 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).

4.3 Copy Mechanism

This section describes the various design issues involved in deciding when the kernel module performs the message copy. 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.

Intuitively, messages are more important to the receiver, we can make receive (i.e., `LiMIC_Irecv`) and wait (i.e., `LiMIC_Wait`) functions of receiver side perform the copy operation. However, it presents heavy dependency on the receiver to make a progress of the communication. In the case of one-copy intra-node communication, if the copy operation is performed on only one of either sender or receiver, the ability of communication progress is significantly degraded. It is mainly because there is no intermediate buffer. An example of such a degradation is when the receiver arrives first at the communication point and calls `MPI_Irecv`, then the sender arrives and calls `MPI_Send`. Then the receiver calls `MPI_Wait` after a large computation or skew. In this case, the sender has to wait for (skew + message copy time).

In order to provide better progress, we consider a design that gives the ability of copy operation to both sender and receiver. So, we consider an approach that performs the copy operation on `MPI_Wait`. Although this approach can provide better progress, this ap-

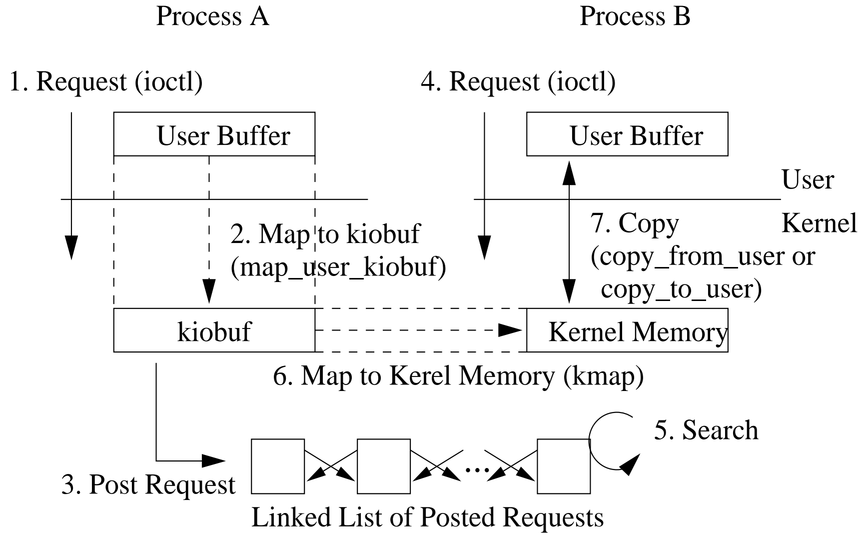


Figure 6. Memory Mapping Mechanism

proach has several drawbacks. Both buffers of sender and receiver should be mapped into `kiobuf`s, since we do not know which communication partner will call `MPI_Wait` earlier. As a result, it will increase the usage of the `kiobuf` pool. Also, both the sender and the receiver buffers will be needed to be pinned.

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. The actual copy operation is performed by the process which arrives later at the communication call. So, regardless of 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. Figure 7 shows the state transition diagram of our kernel module.

4.4 Critical Section Locking

The kernel module data structures are shared between different instances of the kernel executing on the send and the receiver processes. To gurantee consistency, we have to design and implement a locking scheme. To provide efficient message transfer, we have to avoid introducing too much overhead for the queueing and locking.

The design alternatives for implementing the locking mechanism are as follows:

- **Coarse Grained Locking:** An easy way to approach this is to think of the kernel trap as one big critical section. So, if we follow this approach, only one process, either the sender or the receiver

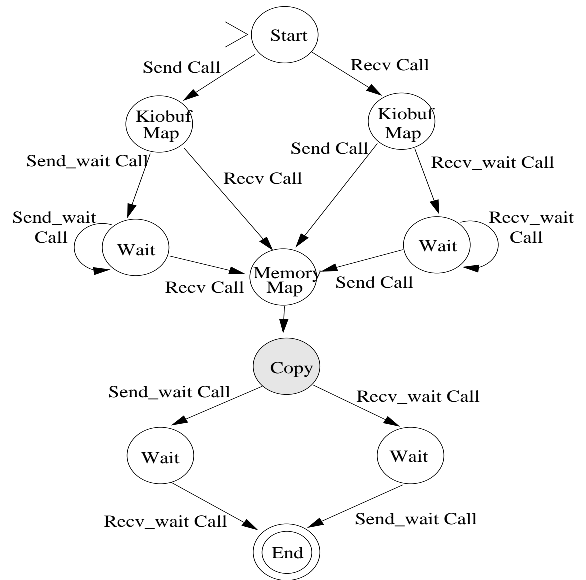


Figure 7. Copy by process arriving later

can access the kernel module functions. This approach, however, prevents an overlapping between intra-node communications, i.e., if the sender and receiver kernel modules are copying different messages, they need to be forcibly serialized.

- **Fine Grained Locking:** On the other hand, a finer grained approach is to lock every shared data structure before access. While this approach can maximize the communication overlap, this introduces a lot of overhead due to frequent locking and unlocking.
- **Medium Grained Locking:** To tackle the defeciencies of coarse and fine grained locking, we come up with a medium grained locking approach. First, we introduce a separate lock variable for managing the `kiobuf` pool. For each process we include one lock for the send and receive and also a lock for the completion queue. The copy operation and memory mapping operation is placed out of the critical section since for high performance, these operations should not be serialized.

4.5 MPI Message Matching

In this section we discuss the design alternatives of LiMIC kernel module in order to support message matching with MPI semantics. There is a separate message queue for messages sent or received through the kernel. 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 the *Context ID*. In MPICH, the *Context ID* usually refers to the MPI communicator and has information about whether the message is a point-to-point message or part of a collective operation. Message matching can also be done by using wildcards 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 not posted into the generic MPI message queue. Rather it 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 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 context id and message source.
- **`MPI_ANY_SOURCE` and `MPI_ANY_TAG`**: This case is a little more challenging than the others. 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. In this case the receive request is posted in both the message queues. In addition, a flag is associated with the request. Whenever the LiMIC finds a message which matches this request, it marks this request as matched and sets the flag. A similar approach is also followed by the user-level message matching mechanism. As mentioned in section 4.4, the update of the flag is guarded by locks to ensure atomicity.

5 Performance Evaluation

In this section we evaluate the performance of LiMIC. We compare the performance of LiMIC with MVAPICH [3] version 0.9.4. MVAPICH 0.9.4 implements a combination of User-space shared memory approach and NIC-based message loopback.

We conducted experiments on an 8 node cluster with the following configuration:

SuperMicro SUPER X5DL8-GG nodes with dual Intel Xeon 3.0 GHz processors, 512 KB L2 cache, 1 GB memory, PCI-X 64-bit 133 MHz bus. The Linux kernel version used was 2.4.22smp from kernel.org.

First we present experimental results on message transfer breakdown, descriptor posting overhead, etc. Then we proceed to Microbenchmark level evaluation and then finally application level results.

5.1 LiMIC Cost Breakdown for Message Transfer

We present results on the various relative cost breakdowns for MPI operations, such as message transfer and posting message descriptors.

5.1.1 Message Transfer Breakdown

We observe that the message copy time dominates the overall send/recv 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"). 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 is shown in Figure 8.

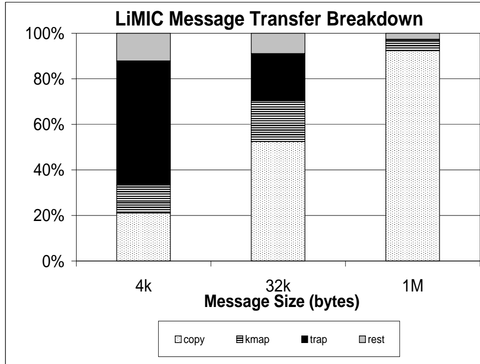


Figure 8. Message Transfer Breakdown

5.1.2 Descriptor Post Breakdown

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. This step also involves the pinning of the user buffer into physical memory. The breakdown is shown in Figure 9.

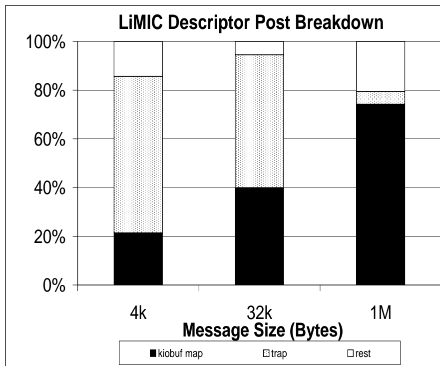


Figure 9. Descriptor Post Breakdown

5.2 Microbenchmarks

In this section, we describe our tests for microbenchmark level evaluation of our design and implementation of LiMIC.

5.2.1 Point-to-Point

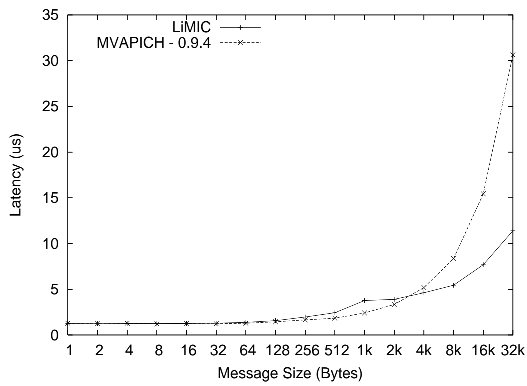
We conducted point-to-point latency and bandwidth tests with LiMIC and compared them with MVAPICH 0.9.4 performance. The latency test is carried out in the standard ping-pong fashion. The sender sends a message and waits for a reply from the receiver. The time for this is recorded by the sender and then it is divided by two to find out one-way latency. For measuring the bandwidth, a simple window based approach was followed. The sender sends $WindowSize$ number of messages and waits for a message from the receiver for every $WindowSize$ messages. The $WindowSize$ used in our experiments was 64. The latency graph is shown in Figure 10 and the bandwidth graph is shown in Figure 11. We observe an improvement of 71% for latency and 405% for bandwidth for 64KB message size.

5.2.2 `MPI_Alltoall` Performance

We conducted an evaluation of the impact of LiMIC on the collectives as well. `MPI_Alltoall` is a dense communication collective. In `MPI_Alltoall` each process exchanges a different message with every other process. The order of communication is n^2. Even though the total communication volume contains a major fraction of inter-node communication, we can see a positive benefit of optimizing intra-node communication from the results. We conduct the experiments in two configurations, (2x2) and (2x4). (2x2) implies two process on two physical nodes. Similarly, (2x4) implies two processes on four physical nodes. It is to be noted that each node in our cluster is a dual node. The performance results can be seen in Figures 12 and 13. We observe an improvement of 24% for 64KB message size on (2x2) configuration.

5.3 NAS Benchmarks

We conducted performance evaluation of LiMIC on the IS benchmark in the NAS Parallel Benchmark suite [6]. IS is an integer sort benchmark kernel that stresses the communication aspect of the network. We conducted experiments with the classes A and B on the configurations (2x1), (2x2), (2X4), and (2X4). We not only measured the benchmark results but also profiled the time spent in MPI calls. This profiling was made possible by a lightweight profiling library called mpiP [4]. The time spent in MPI is reported as the average time spent by every process inside MPI calls. The results are shown in Figures 14 and 15. Further, we profile the number of intra-node messages larger than 1KB and their sizes being used by IS across different classes and problem sizes. They are shown in Table 1 and 2. We can see from this message distribution that as the system size increases, the size of the messages for each class reduces. Since our LiMIC performs well for medium and larger message sizes, we see overall less impact of LiMIC. However, this is not



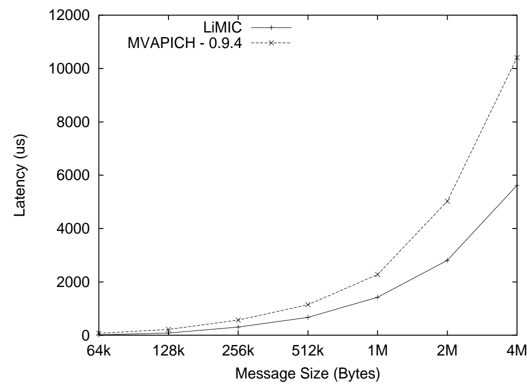
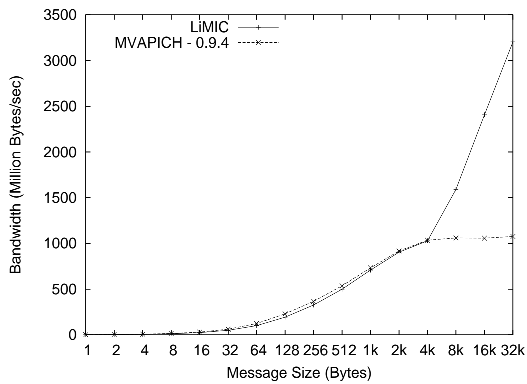


Figure 10. MPI level Latency



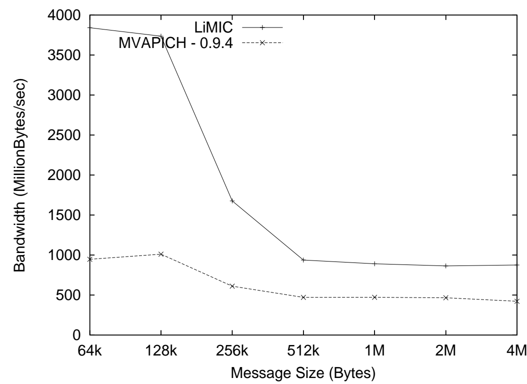
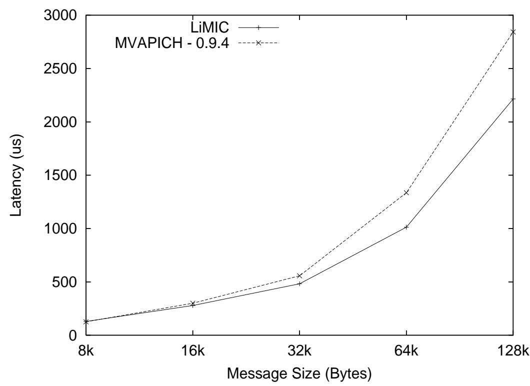


Figure 11. MPI Level Bandwidth



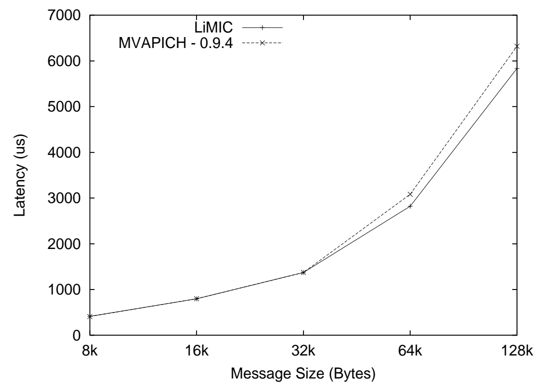
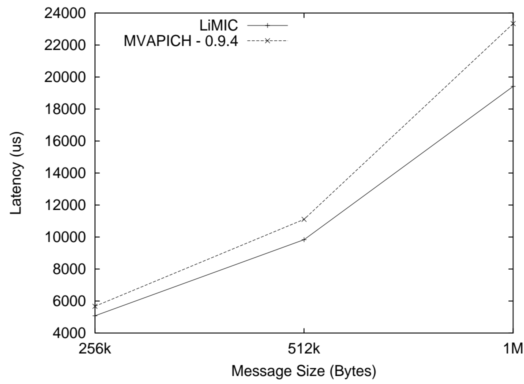


Figure 12. Alltoall for small messages on (2x2) and (2x4) configurations



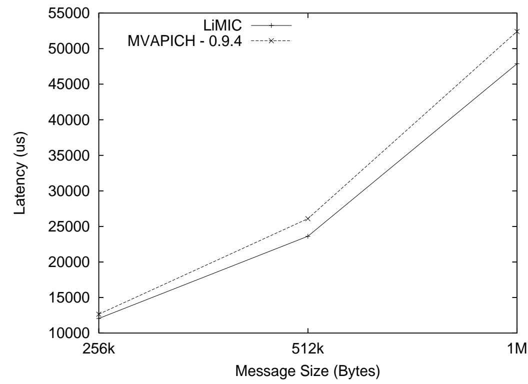


Figure 13. Alltoall for large messages on (2x2) and (2x4) configurations

a limitation imposed by LiMIC, rather a characteristic of the IS benchmark.

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-8M	11	0	0	0
8M-16M	11	0	0	0
16M-32M	0	0	0	0
32M-64M	0	0	0	0

Table 1. 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	0
256k-1M	0	0	0	22
1M-4M	0	0	22	0
4M-8M	0	0	0	0
8M-16M	0	22	0	0
16M-32M	11	0	0	0
32M-64M	11	0	0	0

Table 2. Intra-node message size distribution for IS Class B

5.4 High Performance Linpack

In order to evaluate the importance of intra-node communication on the high performance benchmarks, we run the well known HPL [7] benchmark with LiMIC. Further, we also profile the MPI communication time of Linpack with mpiP. The results are shown in Figure 16. We note that LiMIC can improve the overall computational capacity of each node with the HPL benchmark. We intend to carry out our evaluation of HPL on a larger system size in the final version and technical report of this paper.

6 Related Work

Several researchers have proposed different designs for efficiently implementing intra-node communication in clusters. In this section, we describe in brief their contributions. Geoffray, Tourancheau, Prylli et al [17][11][10] suggest a BIP-SMP multi-protocol layer for intra-node and inter-node communication. Takahashi et al [19][18] provide a device driver called PM/Shmem that supports a direct memory access between processes. MPICH-GM[16] provides a User level shared memory approach for MPI. It also provided a kernel level approach but the support is withdrawn currently to the best of our knowledge. LiMIC is distinguished with previous works in the sense, that it provides a MPI friendly interface and is independent of interconnection networks. Moreover, LiMIC has no limitation that only one of sender or receiver can do the copy operation. The kernel module of MPICH-BPI/SMP allows only receiver to copy. With respect to heterogeneous computing environment such as Grid computing, MPICH-Madeleine[5] and MPICH-G2[9][13] support multiprotocol communication.

7 Conclusions and Future Work

In this paper, we have proposed a design for high performance intra-node communication and implemented a Linux kernel module called LiMIC, which provides an MPI friendly interface and independency on communication libraries or interconnection networks. The LiMIC can be applied to any MPI implementations with only minor modifications regardless of interconnection networks. In addition, it can be easily integrated with a device driver of an interconnection network.

To measure the performance of LiMIC, we have integrated it with MVAPICH, an open source MPI implementation for InfiniBand, and performed various benchmarks. Through the benchmark results, we could observe that LiMIC improved the point-to-point latency and bandwith up to 71% and 405%, respectively. In addition, we could see its benefit on collective communications. Moreover, LiMIC improved NAS IS benchmark by 13.5%.

As a future work, we plan to exploit the memory-to-memory DMA engine instead of using CPU resource to copy the message. Also we intend to study on SMP-aware collectives by utilizing LiMIC. We would like to optimize the LiMIC on NUMA architecture and modify the `kiobuf` manipulation part of LiMIC for Linux kernel version 2.6.

References

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

[2] Mellanox Technologies. http://www.mellanox.com.

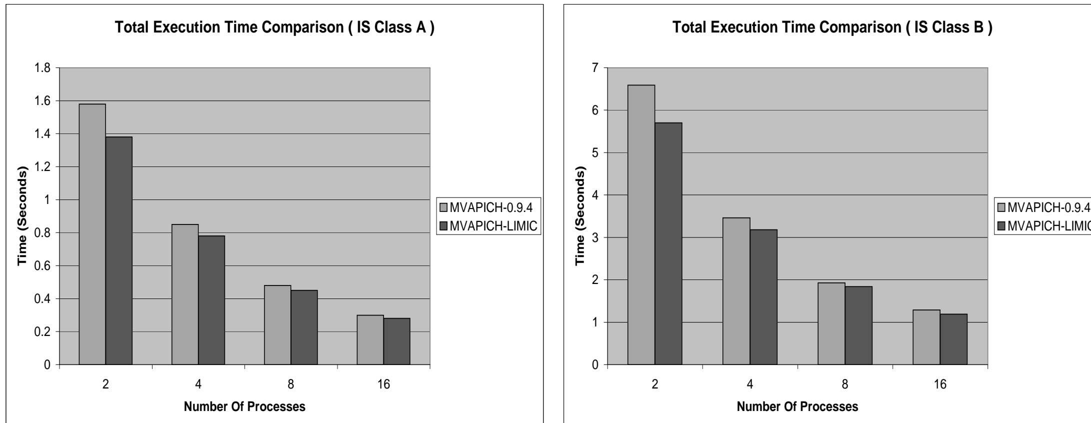


Figure 14. IS Class A & B total execution time

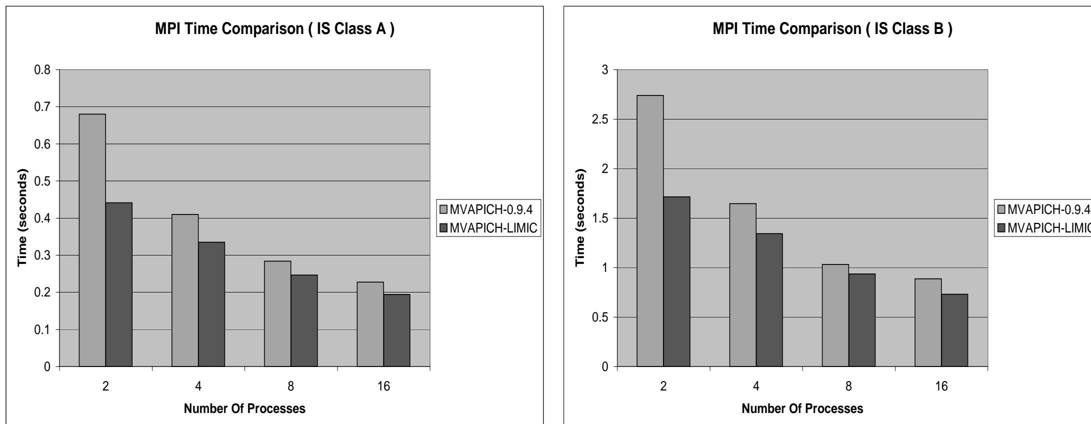


Figure 15. IS Class A & B average time spent in MPI calls

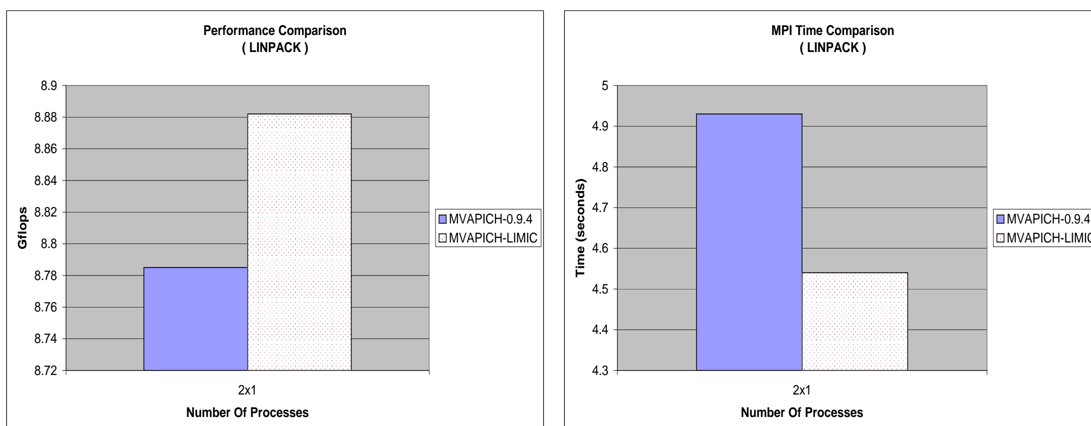


Figure 16. Performance benefits in Linpack with LiMIC: (left) overall Gflops achieved, and (right) Average time spent in MPI calls

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

[4] mpiP: Lightweight, Scalable MPI Profiling. http://www.llnl.gov/CASC/mpip/.

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

[6] 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. volume 5, pages 63–73, Fall 1991.

[7] J. Dongarra. Performance of Various Computers Using Standard Linear Equations Software. Technical Report CS-89-85, University of Tennessee, 1989.

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

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

[10] 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.

[11] Patrick Geoffray, Loic Prylli, and Bernard Tourancheau. BIP-SMP: High Performance Message Passing over a Cluster of Commodity SMPs. In *SuperComputing (SC)*, 1999.

[12] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-Performance, Portable Implementation of the MPI, Message Passing Interface Standard. Technical report, Argonne National Laboratory and Mississippi State University.

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

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

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

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

[17] Loc Prylli and Bernard Tourancheau. BIP: A New Protocol Designed for High Performance Networking on Myrinet. *In Proceedings of the International Parallel Processing Symposium Workshop on Personal Computer Based Networks of Workstations*, 1998. http://lhpca.univ-lyon1.fr/.

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

[19] Toshiyuki Takahashi, Francis O'carroll, Hiroshi Tezuka, Atsushi Hori, Shinji Sumimoto, Hiroshi Harada, Yutaka Ishikawa, and Peter H. Beckman. Implementation and Evaluation of MPI on an SMP Cluster. *Lecture Notes in Computer Science*, 1586, April 1999.