Lightweight Kernel-Level Primitives for High-Performance MPI Intra-Node Communication over Multi-Core Systems

Hyun-Wook Jin[#], Sayantan Sur^{*}, Lei Chai^{*}, Dhabaleswar K. Panda^{*}

[#]*Department of Computer Science and Engineering, Konkuk University
Seoul 143-701, Korea
jinh@konkuk.ac.kr*

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

Abstract— Modern processors have multiple cores on a chip to overcome power consumption and heat dissipation issues. As more and more compute cores become available on a single node, it is expected that node-local communication will play an increasingly greater role in overall performance of parallel applications such as MPI applications. It is therefore crucial to optimize intra-node communication paths utilized by MPI libraries. In this paper, we propose a novel design of a kernel extension, called LiMIC2, for high-performance MPI intra-node communication over multi-core systems. LiMIC2 can minimize the communication overheads by implementing lightweight primitives and provide portability across different interconnects and flexibility for performance optimization. Our performance evaluation indicates that LiMIC2 can attain 80% lower latency and more than three times improvement in bandwidth. Also the experimental results show that LiMIC2 can deliver bidirectional bandwidth greater than 11GB/s.

I. INTRODUCTION

Commodity clusters have helped shaped the landscape of high-performance computing for more than a decade. Leveraging high volume off-the-shelf components, commodity clusters provide excellent cost-performance ratio. Of the several programming models available for scientific computing applications, MPI [1] is the most widely used. There exists a large volume of high-performance scientific applications that are written using MPI [2][3] and more continue to be written. Modern MPI libraries utilize high-speed networks for communication with remote processes and shared memory for communication with processes local to a compute node.

Recently, the architecture of the commodity CPUs has undergone a fundamental change. It has been shown that power consumption and heat dissipation issues restrict the clock frequencies in modern microprocessors. The only feasible way the computation power of a processor can keep on increasing as per Moore's Law [4], is by increasing the number of processing cores on the chip. Modern processors from Intel, AMD, Sun and IBM have multiple cores on a chip. As more and more compute cores become available on a single node, it is expected that node-local communication will play an increasingly greater role in overall performance of

MPI applications. It is therefore crucial to optimize intra-node communication paths utilized by MPI libraries.

LiMIC [5], is a Linux kernel module for intra-node communication which enables high-performance message passing for MPI libraries. LiMIC achieves very good performance since it no longer requires MPI libraries to copy messages in and out of shared memory regions. Rather, LiMIC enables messages to be copied *directly* from sending process' memory to the receiver process. Thus, LiMIC eliminates unnecessary message copies and enhances cache utilization. In this paper we present a revised design of LiMIC, called LiMIC2, for multi-core processors and a detailed performance evaluation. Our results indicate that LiMIC2 can attain 81% lower latency and 358% better bandwidth than the shared memory based intra-node communication of MVAPICH [6]. Also the measurement results show that LiMIC2 can deliver 11443 MillionBytes/Sec of bidirectional bandwidth.

The rest of the paper is organized as follows: In Section II, we overview existing intra-node communication mechanisms and MVAPICH. We discuss design alternatives of kernel-level support for intra-node communication in Section III. Section IV gives details of design and implementation of LiMIC2. Its performance is presented in Section V. Finally we conclude the paper in Section VI.

II. BACKGROUND

In this section, we describe existing intra-node communication mechanisms. We also overview the intra-node communication mechanism of MVAPICH to which our proposed scheme will be applied.

A. Existing Intra-Node Communication Mechanisms

An intelligent Network Interface Card (NIC) can provide a NIC-level loopback [7]. 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, we can eliminate overheads on the network link because the message is not injected into the network. However, there still exist two DMA operations

(i.e., from host to NIC memory and from NIC to host memory). Although I/O buses are getting faster, the DMA overhead is still high. Further, the DMA operations cannot utilize the cache effect.

Another mechanism for intra-node communication is to use user-space shared memory [8]. This mechanism 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 then copies the message to its own buffer. This approach involves minimal setup overhead for every message exchange. The CPU resource, however, is tied down with the memory copy operations. In addition, as the size of the message grows, the performance deteriorates because vigorous copy-in and copy-out also destroys the cache contents.

Kernel-based memory mapping approach takes help from the operating system kernel to copy messages directly from one user space to another without any additional copy operation [5][9][10]. The sender or the receiver process posts the message request descriptor in a message queue indicating its virtual address, 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's user buffer. Thus this approach involves only one data copy.

Buntinas et al. [11] have compared several different approaches of intra-node communication on an SMP system.

B. Intra-Node Communication of MVAPICH

MVAPICH is a high performance MPI implementation over InfiniBand clusters [6]. The implementation is based on MPICH [12]. MVAPICH is currently being used by more than 535 organizations world-wide.

Current MVAPICH utilizes a user-space shared memory approach for its intra-node communication [8]. A temporary file is created and all the processes attach themselves to the file using mmap and use the file (essentially shared memory) for intra-node communication.

For small and control messages, the shared memory is organized on a pair-wise basis, i.e. each pair of processes have two shared memory buffers, holding messages toward each direction. We call these buffers "receive buffers" since essentially every process has a receive buffer corresponding to every other process. The send/receive mechanism is straightforward. The sending process writes data from its source buffer into the shared buffer. Then the receiving process copies the data from the shared buffer into its destination buffer. In this paper, we call it eager protocol.

For large messages, each process maintains a pool of fixed sized buffers, we call it shared buffer pool, which is used for the process to send messages to any other processes. The benefits of using the shared buffer pool for large messages include: First, the pool size is flat. It does not increase in proportion to the number of processes. Second, the messages can be sent in a pipelined manner. And third, as soon as a buffer is cleared (data is copied by the receiving process) it

can be reused by the sending process, which may improve L2 cache utilization. In this paper, we call it rendezvous protocol.

Message matching is performed based on source rank, tag, and context id which identifies the communicator. Message ordering is ensured by the memory consistency model and use of memory barrier if the underlying memory model is not consistent.

III. DESIGN ALTERNATIVES OF KERNEL-LEVEL SUPPORT

As discussed in Section II.A, the kernel-based approach has the significant potential to provide efficient MPI intra-node communication. In this approach, to achieve direct data movement, a process should be able to access the other process' virtual address space so that the process can copy the message to/from the other's address space. This can be achieved by memory mapping mechanism that maps a part of the other process' address space into its own address space. After the memory mapping, the process can access mapped area as its own. This memory mapping should be done in the kernel context. In this section, we classify the design alternatives of the kernel-based memory mapping approach into three: i) Extension of NIC device driver, ii) Stand-alone communication module, and iii) Lightweight primitives.

A. Extension of NIC Device Driver

Traditionally, researchers have explored kernel based approaches as an extension to the features available in user-level protocols such as GM [13] and BIP [14]. In this approach, the MPI library blindly calls the interfaces of the user-level protocol without the decision of inter- or intra-node communication. Then the user-level protocol internally decides whether the call is for inter- or intra-node communication based on the source or destination address. In case of intra-node communication, the user-level protocol performs the memory mapping with the assist of the NIC device driver. Geoffray et al. [9] and Takahashi et al. [10] have suggested the NIC device drivers that support a direct memory access between processes.

It is to be noted that the most of the user-level protocols are proprietary and designed for a specific NIC/interconnect. As a result, this approach has been non-portable to other user-level protocols or other MPI implementations. Further, since the MPI library blindly calls routines provided by the user-level protocol, this mechanism denies any sort of optimization-space for the MPI library developer. For example, an MPI library developer may want to choose thresholds for the hybrid approach of two or more intra-node communication mechanisms described in Section II.A [15]. The extension of NIC device driver, however, cannot provide such flexibility to the MPI library developer.

B. Stand-Alone Communication Module

In order to avoid the limitations of the extension of NIC device driver, we can generalize the kernel-access interface and make a stand-alone communication module. The stand-alone communication module manages send, receive, and completion queues internally so that, once the MPI library decides to use intra-node communication channel, simply calls

the interface and let the communication module handle the message transportation between intra-node processes. In our previous research, we have proposed a stand-alone communication module called LiMIC (Linux kernel module for MPI Intra-node Communication) and showed that the communication latency and bandwidth can be improved significantly [5][15].

This approach 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. However, having the separate message queues from the MPI library brings on many tricky issues. Since the internal message queues are shared between intra-node processes, the stand-alone communication module has to take care of synchronization of accessing the queues. This synchronization overhead can increase as proportional to the number of intra-node processes/cores. In addition to the synchronization, the MPI message matching (i.e., source, tag and context id matching) also has to be done by itself.

C. Lightweight Primitives

In this approach, the kernel extension provides lightweight primitives to perform the memory mapping between different processes. The kernel module exposes the interface for memory mapping and data copy to the MPI library, while the stand-alone communication module described in Section III.B provides more communication-friendly interface. The lightweight primitives do not need to have any internal queues and data structures shared between intra-node processes. Therefore, the lightweight primitives can avoid the synchronization and MPI message matching, which can result in lower overhead than the stand-alone communication module. More importantly, it may increase the parallelism of local MPI processes.

The approach of lightweight primitives has the potential of achieving the best performance compared to the other kernel-level approaches. This alternative also preserves the advantages of the stand-alone communication module such as portability across different interconnects and flexibility to optimize the performance.

IV. DESIGN AND IMPLEMENTATION OF LIGHTWEIGHT KERNEL-LEVEL PRIMITIVES (LiMIC2) FOR MVAPICH

As we have discussed in Section III, the lightweight primitives can provide better performance and benefits than the others. To the best of our knowledge, this approach has not been explored in the literature yet for MPI intra-node communication. In this section, we propose a new design for lightweight kernel-level primitives called LiMIC2. To distinguish with our previous work of stand-alone communication module approach [5], we use the version number ‘2’. We also modify MVAPICH to exploit LiMIC2 at the MPI level.

A. LiMIC2: Lightweight Kernel-Level Primitives

LiMIC2 consists of a runtime loadable kernel module and user library. The kernel module implements the kernel-level memory mapping between different processes without any kernel modification. The user library provides the interface to the kernel module functions for the MPI library.

There are three key functions provided by the LiMIC2 kernel module. The first function extracts the information of the sending process. The user buffer of the sender is mapped into the address space of the receiver, which is done in the context of the receiver. Therefore, to perform the memory mapping, the receiver needs information about the sender’s memory space. That is, we need to keep the information required for the memory mapping when the sender is the current process. The function provides such information, which includes the access point of the virtual memory page table and the `task_struct` data structure. Another key function returns the page table entries of a given user buffer using the information extracted by the first function as input parameter. The obtained page table entries are used when the memory mapping is performed. The last function maps the user buffer of the sender into the kernel memory space of the receiver and copies the data from the mapped buffer into the destination user buffer.

LiMIC2 provides the user-level library so that the MPI library can easily utilize the kernel-level functions of LiMIC2. The library functions are as follows:

- `int limic_open(void)`: This call returns the file descriptor to use LiMIC2. This is used in `MPI_Init()`.
- `void limic_close(int fd)`: This call simply close the LiMIC2 virtual device. This is used in `MPI_Finalize()`.
- `int limic_tx_init(int fd, void *buf, int len, limic_user *lu)`: This call saves the process information into the `limic_user` structure of which details are hidden from the MPI library.
- `int limic_rx_comp(int fd, void *buf, int len, limic_user *lu)`: This call performs actual memory mapping and direct data movement, where the `limic_user` structure has the information obtained from `limic_tx_init()`.

As we can see, the library functions are easy to use. The `limic_tx_init()` function calls the first kernel-level function described earlier on the sender side. The `limic_rx_comp()` function utilizes the rest of the kernel-level functions on the receiver side. The library functions internally calls the `ioctl()` system call to invoke the kernel-level functions of LiMIC2.

In this way, LiMIC2 provides the primitives for the memory mapping and data copy operations without significant additional overheads. LiMIC2 does not have internal message queues but lets the MPI library take care of the actual message exchange. As a consequence, LiMIC2 does not involve the MPI message matching. In addition, it suggests simple library

interfaces, which lead to the portability across various MPI implementations and interconnects.

B. MVAPICH over LiMIC2

To see the benefit of LiMIC2 we apply it to an MPI implementation, MVAPICH. Figure 1 shows how the modified MVAPICH sends and receives a message using LiMIC2. On the sender side, if MVAPICH decides to send an intra-node message using LiMIC2, it calls the LiMIC2 library function, `limic_tx_init()`. This function returns the pointers to the kernel-level process information but MVAPICH does not need to know the details of this information and even how to use. This process information is blindly sent to the receiver as a LIMIC_POST control message. The control messages are exchanged through the user-level shared memory communication channel of MVAPICH. On the receiver side, the MVAPICH calls `limic_rx_comp()` with the process information piggybacked in the LIMIC_POST control message. The `limic_rx_comp()` library function invokes the memory mapping and data copy operations of LiMIC2 kernel module and returns the length copied/received successfully. Then MVAPICH sends the LIMIC_COMP control message to the sender to allow the sender to modify or free the send buffer.

Fig. 1 Steps of message transmission of MVAPICH over LiMIC2

V. PERFORMANCE MEASUREMENT

In this section, we evaluate the MPI-level communication performance with LiMIC2. We use the OSU benchmarks [6] to measure the latency and bandwidth. We measure the performance for both CMP (Chip-level MultiProcessing) and non-CMP cases. The CMP case represents the communication between two processes running on two cores on the same chip (i.e., single package) while the non-CMP case represents the communication between two cores on different chips. It is to be noted that, in the CMP case, the cores share the L2 cache but, in the non-CMP case, each core uses its own L2 cache.

Table 1 shows the experimental system setup. We use MVAPICH (version 0.9.9) for an MPI implementation. The MVAPICH supports the single node system case that does not have any interconnection network but use only intra-node communication channel. As described in Section II.B, MVAPICH implements the user-level shared memory for

intra-node communication. It also use different protocols based on the message size; eager protocol for short messages and rendezvous protocol for large messages. In our experimental system, the threshold of message size to switch between these protocols is 32KB. Since LiMIC2 is beneficial to the large messages, the modified MVAPICH also uses the existing eager protocol for small messages (i.e., up to 32KB message size).

TABLE I
EXPERIMENTAL SYSTEM

Processor	2-Way Intel Dual-Core Xeon 2.66GHz (Woodcrest)
Memory	2GB (DDR2-533)
System Bus	1333MHz
Chipset	Intel 5000P Chipset
Operating Systems	Linux (kernel version 2.6.15)

Fig. 2 Latency for non-CMP case

Fig. 3 Bandwidth for non-CMP case

A. Non-CMP Case

Figure 2 shows the results of latency measurement for non-CMP case. As mentioned earlier, both MVAPICH-0.9.9 and MVAPICH-LiMIC2 use the eager protocol of user-level shared memory for messages smaller than 32KB. As a consequence, we see that the lines in the graph are overlapped up to 32KB message size, which is the same in the other graphs also. In Figure 2, we can observe that LiMIC2 reduces the latency up to 81% compared with the original MVAPICH implementation. This is due to the fact that LiMIC2 reduces the number of data copy into only one while the original

MVAPICH performs two data copy operations. In addition, since LiMIC2 does not use additional buffers, it can benefit more cache effect.

Figure 3 shows the bandwidth results for non-CMP case. The figure shows that LiMIC2 improves the bandwidth up to 358%. In the figure, LiMIC2 reports the 5155MillionBytes/Sec for 1MB message. Since there is a high jump of bandwidth in between 32KB and 64KB message sizes, we believe that if we tune the threshold of eager protocol we are able to get even better bandwidth for smaller messages.

Fig. 4 Latency for CMP case

Fig. 5 Bandwidth for CMP case

Fig. 6 Bidirectional bandwidth for CMP case

B. CMP Case

Figures 4 and 5 present the latency and bandwidth measurement results for CMP case, respectively. Compared to the non-CMP case, we see smaller improvement with LiMIC2.

This is because the data movement between cores in the same chip (CMP case) is much faster than between cores of different chips (non-CMP case). Thus, the benefit of reducing number of data copies is not shown much in the latency and bandwidth tests. We, however, still see 47% and 24% improvement of latency and bandwidth with LiMIC2, respectively.

In case of the original MVAPICH implementation, both cores of sender and receiver are busy to copy the message to/from the shared memory area. On the other hand, in case of LiMIC2, only the receiver is busy for the copy operation. In order to see this benefit of saving the core resource on the sender side, we run the bidirectional bandwidth test. Figure 6 shows the measurement results for CMP case. We can observe that LiMIC2 improves the bidirectional bandwidth up to 282%. It is to be noted that the maximum bidirectional bandwidth achieved is 11443MillionBytes/Sec.

VI. CONCLUSIONS AND FUTURE WORK

In this paper we have proposed a novel design of a kernel extension, called LiMIC2, for high-performance MPI intra-node communication over multi-core systems. We have showed that LiMIC2 can minimize the communication overheads by implementing lightweight primitives and provide portability across different interconnects and flexibility for performance optimization. Our performance evaluation indicates that, in non-CMP case, LiMIC2 can attain 81% lower latency and 358% better bandwidth than the shared memory based communication of MVAPICH. In CMP case, LiMIC2 has achieved 282% improvement of bidirectional bandwidth. The maximum bidirectional bandwidth achieved by LiMIC2 is 11443MillionBytes/Sec.

As future work, we plan to measure the MPI application-level performance. We also intend to extend LiMIC2 for one-sided communication of MPI-2.

ACKNOWLEDGMENT

The KU part of this research is supported in part by the Faculty Research Fund of Konkuk University (2006) and Electronics and Telecommunications Research Institute (ETRI) grant #6010-2007-0022. The OSU part is supported in part by Department of Energy's grants #DE-FC02-06ER25749 and #DE-FC02-06ER25755 and NSF grants #CNS-0403342 and #CCF-0702675.

REFERENCES

- [1] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard*, June 1995.
- [2] G. Grell, J. Dudhia, and D. Stauffer, "A Description of the Fifth-Generation Penn State/NCAR Mesoscale Model (MM5)," *Tech. Rep. NCAR/TN-398+STR*, National Center for Atmospheric Research, 1994.
- [3] H. Berendsen, D. van der Spoel, and R. van Drunen, "GROMACS: A Message-Passing Parallel Molecular Dynamics Implementation," *Computer Physics Communications*, 91(1), 1995.
- [4] G. Moore, "Cramming More Components Onto Integrated Circuits," *Electronics Magazine*, (38):8, April 1965.
- [5] H.-W. Jin, S. Sur, L. Chai, and D. K. Panda, "LiMIC: Support for High-Performance MPI Intra-Node Communication on Linux Cluster," in *Proc. ICPP-05*, pp. 184-191, June 2005.

- [6] MVAPICH: MPI over InfiniBand and iWARP, <http://mvapich.cse.ohio-state.edu/>.
- [7] Mellanox Technologies, *Mellanox InfiniBand InfiniHost MT23108 Adapters*, <http://www.mellanox.com>, July 2002.
- [8] L. Chai, A. Hartono, and D. K. Panda, "Designing High Performance and Scalable MPI Intra-node Communication Support for Clusters," in *Proc. Cluster 2006*, September 2006.
- [9] 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.
- [10] T. Takahashi, S. Sumimoto, A. Hori, H. Harada, and Y. Ishikawa, "PM2: High Performance Communication Middleware for Heterogeneous Network Environments," in *Proc. SC2000*, 2000.
- [11] D. Buntinas, G. Mercier, and W. Gropp, "Data Transfers Between Processes in an SMP System: Performance Study and Application to MPI," in *Proc. ICPP-06*, August 2006.
- [12] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "High-Performance, Portable Implementation for the MPI Message Passing Interface Standard," *Parallel Computing*, 22(6):789-828, 1996.
- [13] Myricom Inc., *The GM Message Passing System*, <http://www.myri.com>, January 2000.
- [14] L. Prylli and B. Tourancheau, "BIP: a new protocol designed for high performance networking on myrinet," in *Proc. PC-NOW*, 1998.
- [15] L. Chai, S. Sur, H.-W. Jin, and D. K. Panda, "Analysis of Design Considerations for Optimizing Multi-Channel MPI over InfiniBand," in *Proc. CAC 2005*, April 2005.