Comput Sci Res Dev (2010) 25: 3–14
DOI 10.1007/s00450-010-0115-3

SPECIAL ISSUE PAPER

Designing truly one-sided MPI-2 RMA intra-node communication on multi-core systems

Ping Lai · Sayantan Sur · Dhabaleswar K. Panda

Published online: 10 April 2010
© Springer-Verlag 2010

Abstract The increasing popularity of multi-core processors has made MPI intra-node communication, including the intra-node RMA (Remote Memory Access) communication, a critical component in high performance computing. MPI-2 RMA model includes one-sided data transfer and synchronization operations. Existing designs in popularly used MPI stacks do not provide truly one-sided intra-node RMA communication. They are built on top of two-sided send-receive operations, therefore suffering from overheads of two-sided communication and dependency on the remote side. In this paper, we enhance existing shared memory mechanisms to design truly one-sided synchronization. In addition, we design truly one-sided intra-node data transfer using two kernel based direct copy alternatives: basic kernel-assisted approach and I/OAT-assisted approach. Our new design eliminates the overhead of using two-sided operations and eliminates the involvement from the remote side. We also propose a series of benchmarks to evaluate various performance aspects over multi-core architectures (Intel Clovertown, Intel Nehalem and AMD Barcelona). The results show that the new design obtains up to 39% lower latency for small and medium messages and demonstrates 29% improvement in large message bandwidth. Moreover, it provides superior performance in terms of better scalability, reduced cache misses, higher resilience to process skew and increased computation and communication overlap. Finally, up to 10% performance benefits is demonstrated for a real scientific application AWM-Olsen.

Keywords MPI-2 RMA · Intra-node communication · Multi-core system

This research is supported in part by DOE grants #DE-FC02-06ER25749 and #DE-FC02-06ER25755; NSF grants #CNS-0403342, #CCF-0702675, #CCF-0833169, #CCF-0916302 and #OCI-0926691; grants from Intel, Mellanox, Cisco systems, QLogic and Sun Microsystems; and equipment donations from Intel, Mellanox, AMD, Appro, Chelsio, Dell, Fujitsu, Fulcrum, Microway, Obsidian, QLogic, and Sun Microsystems.

P. Lai (✉) · S. Sur · D.K. Panda
Computer Science and Engineering, Ohio State University, Columbus, USA
e-mail: laipi@cse.ohio-state.edu

S. Sur
e-mail: surs@cse.ohio-state.edu

D.K. Panda
e-mail: panda@cse.ohio-state.edu

1 Introduction

Parallel scientific computing has been growing dramatically over the past decade. It is driven by compute/communicate intensive and data hungry applications. It has led to faster development of new technologies, and massive deployment of workstation clusters coupled with revolutionary changes in programming models. Multi-core technology is an important contributor to this trend. As it becomes mainstream, more and more clusters are deploying multi-core processors. Quad-core and Hex-core processors are quickly gaining ground in many applications. In fact, more than 87% of the systems in the November 2009 ranking of the Top500 supercomputers belong to the multi-core processor family. In this scenario, it is expected that considerable communication will take place within each node. It suggests that intra-node communication design of a programming model will play a key role in overall performance of end applications.

In the last decade MPI (Message Passing Interface) [14] has evolved as one of the most popular programming models for distributed memory systems. MPI-1 specification defines message passing based on send-receive operations. It

is generally referred to as two-sided communication model, as both the sender and receiver are involved in communication. Subsequently, MPI-2 [21] standard introduced the one-sided communication model also known as Remote Memory Access (RMA) model which includes data transfer and synchronization operations. Ideally only one process participates in the communication, so it has to specify all the communication parameters including the parameters of remote side. Synchronization is done explicitly to guarantee the communication completion. Here the process initiating the communication is called *origin*, and the remote process is called *target*. MPI-2 currently supports three one-sided data communication operations, i.e., *MPI_Put*, *MPI_Get* and *MPI_Accumulate*, and two synchronization modes, i.e., active mode and passive mode.

There are different ways to design the one-sided model. One way is to implement it on top of two-sided operations. This approach has good portability, but has extra overheads. For example, it has intermediate layer handover and two-sided inherent overhead (e.g. tag matching and *rendezvous* handshake etc.). Several popular MPI implementations such as MPICH2 [3] and LAM/MPI [2] use this two-sided based approach. The second approach is to utilize special features such as RDMA operations to achieve truly one-sided communication. MVAPICH2 [4] and OpenMPI [8] use this design for inter-node RMA communication. However, all of these MPI stacks do not have truly one-sided design for intra-node case. This could significantly degrade the overall performance due to increasing importance of intra-node communication and higher overhead of the two-sided based approach. Therefore, it is necessary to design truly one-sided intra-node communication mechanisms.

In this paper we design and implement a truly one-sided model for intra-node RMA communication, and carry out comprehensive evaluations and analysis. We design truly one-sided data transfer using two alternatives. One is based on kernel-assisted direct copy and the other one utilizes I/OAT [1] technology to offload this copy. *MPI_Put* and *MPI_Get* are naturally mapped to direct copy with no interruption to *target* (support for *MPI_Accumulate* will be investigated in the future). This design eliminates two-sided operation related overhead. More importantly, since the target is not involved, its progress does not block the communication. For synchronization, as the passive mode has been investigated in [18, 22, 23], we only deal with the active mode. Shared memory mechanism is utilized to realize truly one-sided synchronization. We come up with several benchmarks running on three multi-core architectures, i.e., Intel Clovertown, Intel Nehalem and AMD Barcelona. From the experimental results we observe that our new design provides much better performance in terms of latency and bandwidth as compared to the existing two-sided based designs. Particularly, the basic kernel-assisted approach improves the latency for small and medium messages by 39%, and the I/OAT based approach yields up to 29% improvement in large message bandwidth. Furthermore, we see that the new design achieves better scalability, fewer cache misses and more computation and communication overlap. It is also more tolerant to process skew and offers more benefits in real applications.

The rest of this paper is organized as follows. In Sect. 2, we provide the introduction on MPI-2 one-sided RMA communication model and the common mechanisms for intra-node communication. Then we analyze the drawbacks of the existing designs in Sect. 3. In Sect. 4, we describe the proposed design in detail. We present and analyze the experimental results in Sect. 5, discuss the related work in Sect. 6, and summarize conclusions and possible future work in Sect. 7.

2 Background

In this section, we describe the required background knowledge for this work.

2.1 MPI-2 RMA model

MPI-2 RMA model includes the data transfer and synchronization operations. It defines that the *origin* can directly access a memory area on the *target* process. This memory is called *window* which is defined by a collective call MPI_Win_create. Ideally the *origin* specifies all the parameters including the target memory address, so the target is unaware of the on-going communication.

MPI-2 defines three RMA data transfer operations. MPI_Put and MPI_Get transfer the data to and from a target window. MPI_Accumulate combines the data movement to target with a reduce operation. These operations are not guaranteed to complete when the functions return. The completion must be ensured by explicit synchronization. In other words, MPI-2 allows one-sided operations only within an *epoch* which is the period between two synchronization events. Synchronization is classified as *passive* (no explicit participation from the target) and *active* (involving both origin and target). In the passive mode, the origin process uses MPI_Win_lock and MPI_Win_unlock to define an epoch. The active mode is classified into two types: a) collective MPI_Win _fence on the entire group; and b) collective on a smaller group, i.e., an origin uses MPI_Win_start and MPI_Win_complete to specify *access epoch* for a group of targets, and a target calls MPI_Win_post and MPI_Win_wait to specify *exposure epoch* for a group of origins. The origin can issue RMA operations only after the target window has been posted, and the target can complete an epoch only when all the origins in

the group have finished accessing to its window. Normally multiple RMA operations are issued in an epoch to amortize the synchronization overhead. In this paper, we primarily concentrate on active synchronization and use the post-wait/start-complete mode as the example in the following sections.

2.2 Mechanisms for intra-node communication

There are several common mechanisms for intra-node communication. The easiest one is through user space shared memory. Two processes sharing a buffer can communicate with copy-in and copy-out operations. This approach usually provides benefits for small messages, while not good for large messages due to the two copies overhead. MVAPICH2, MPICH2 [3] and OpenMPI all use this mechanism for two-sided small message passing.

The second category of mechanisms take help from the kernel to save one copy. In the kernel space, the data is directly copied from the sender's address space to the receiver's address space. Some such kernel modules have been developed for MPI two-sided large message communication. For example, LiMIC2 [19] is used in MVAPICH2 [4] and KNEM [11] is used in MPICH2 and OpenMPI. Based on this approach, another alternative is to further offload the direct copy to DMA (Direct Memory Access) engine. Intel I/OAT [1] is such a DMA engine which has multiple independent DMA channels with direct access to main memory. It copies the data asynchronously while releasing the CPU for other work. KNEM [11] has I/OAT support for very large messages. These two kernel-assisted direct copy approaches both fit the one-sided model very well. As long as the *origin* provides the kernel or I/OAT engine with the buffer information about itself and *target*, the data can be directly copied to the *target* window without interrupting it.

3 Detailed motivation

As illustrated in Fig. 1, various MPI stacks[1] design MPI-2 RMA communication using two means, i.e., based on two-sided send-receive operations[2] and truly one-sided approach. Truly one-sided approach bypasses the two-sided operations to build the RMA communication directly over the underlying remote memory access mechanisms (e.g., network RDMA mechanism, or node level RMA mechanisms). While the inter-node truly one-sided design has

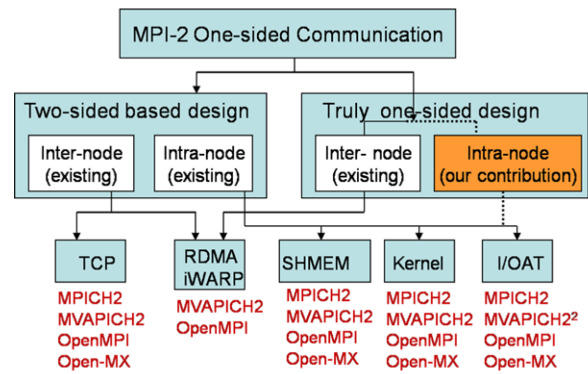


Fig. 1 MPI-2 RMA communication design in various MPI stacks

been implemented in some stacks (e.g., in MVAPICH2 and OpenMPI), to the best of our knowledge there are no existing truly one-sided designs for intra-node communication.

Two-sided based design has several drawbacks. It is unavoidable to inherit the two-sided overhead. For instance, short messages need copy-in and copy-out through the shared memory. Large messages require sending buffer information (in MPICH2) or even *rendezvous* handshake (in MVAPICH2) before the data is actually transferred. It not only adds latency, but also leads to the interactive dependency between the origin and target processes, which is contrary to the goal of one-sided model. This approach also does not provide any overlap between communication and computation. This could result in very bad performance if the origin and target processes are skewed. We experimented this using the popular MPI stacks (MVAPICH2, OpenMPI and MPICH2) and demonstrated this effect. Please refer to Sect. 5.2 for detailed results. All of these observations suggest the demand on designing a truly one-sided intra-node communication.

4 Proposed design and implementation

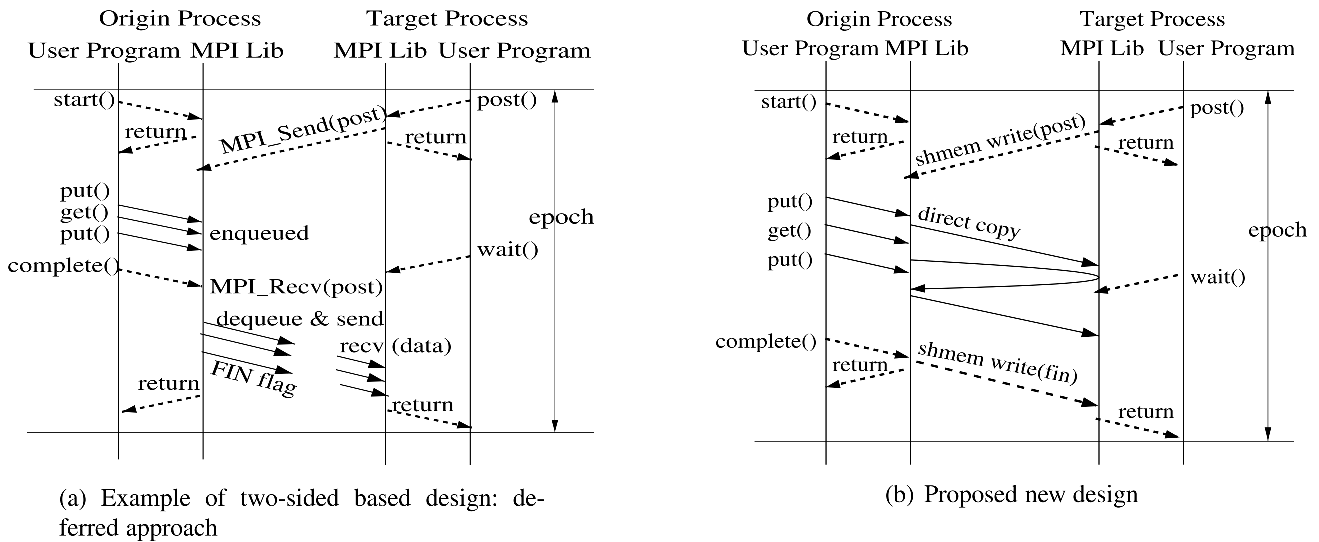
In this section, we describe the details of our design. In the following, we use post, wait, start, complete, put and get as the abbreviations for the corresponding MPI functions.

4.1 Design goals

In order to better understand the new design, first let us see an example of the existing two-sided based design. Figure 2(a) shows the approach [24] used in MPICH2 and MVAPICH2 for intra-node one-sided communication. The dotted lines represent synchronization steps and the solid lines represent data communication steps. At origin, MPI_Win_start and the put's/get's return immediately. The put's/get's are queued locally and will be issued in

[1]Strictly speaking Open-MX is not an MPI implementation, but it can be ported to several MPI implementations.

[2]Please note that I/OAT support for two-sided communication is not included in the current MVAPICH2 release. It will be available in future releases.



(a) Example of two-sided based design: deferred approach

(b) Proposed new design

Fig. 2 Comparison of old design and new design

MPI_Win_complete after it checks that the target window has been posted (by calling MPI_Recv which matches with MPI_Send issued in MPI_Win_post by the target). The completion is marked by adding a special flag in the last put/get packet. Even though this design has minimized synchronization cost, it is not truly one-sided and has the drawbacks mentioned in the last section.

Figure 2(b) shows our proposed new design. Basically we aim to realize the truly one-sided nature for both synchronization operations (post and complete) and the data transfer operations (put and get), thereby removing the two-sided related overhead and alleviating the impact of process skew. "start" operation still returns without doing anything. A "put/get" can be issued immediately if the "post" has been there, or be issued in later functions as soon as "post" is detected. As the communication is within a node, we utilize the aforementioned (in Sect. 2.2) mechanisms as the basis for our design.

Our design is implemented in MVAPICH2 [4] which currently uses two-sided based approach as shown in Fig. 2(a). We make changes on CH3 layer to design a customized interface for intra-node truly one-sided communication.

4.2 Design of truly one-sided synchronization

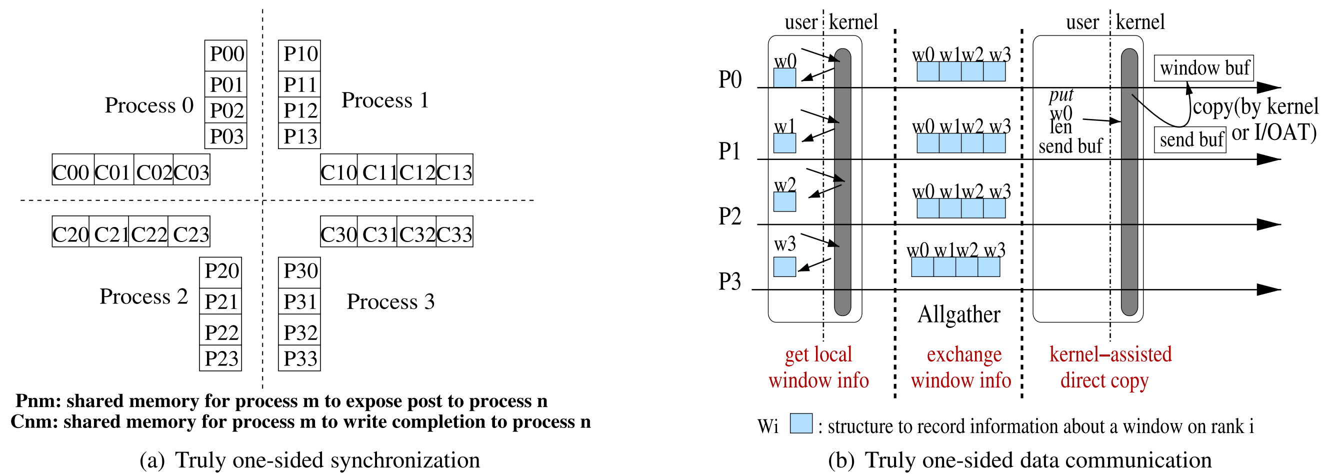
As mentioned earlier, we use the post-wait/start-complete synchronization as the example. At the beginning, the target informs the origin that its window has been posted for access, and at the end the origin notifies the target that all the one-sided operations on the window have finished.

We utilize shared memory for truly one-sided design, as illustrated in Fig. 3(a) using 4 processes for instance. Every process creates two shared memory buffers used by others to write "post" and "complete" notifications, respectively. Then each process attaches to the shared memory of other processes within the same node. Shared memory creation, information exchange and attachment operations take place in MPI_Win_create which is not in the communication critical path. Also, since these buffers are actually bit vectors whose size is very small, the scalability will not be an issue. Using this structure, a target can directly write "post" into the corresponding shared memory. When an origin starts an epoch, it checks its post buffer and knows which processes have posted their windows. Consequently, it can immediately initiate the put/get on those windows instead of queuing them. Similarly, upon finishing all the operations, the origin simply writes a "FIN" message to the completion shared memory where the corresponding target checks for the completion. It is to be noted that for the processes in the group that do not end up being the real target, the origin still needs to notify the completion to them so that those processes will not be blocked in this epoch.

The advantages of this design are two folds. First it does not need send and receive for "post" step. The other is that it is truly one-sided with no dependency on remote side's participation. Additionally, the put/get operations are not deferred.

4.3 Design of truly one-sided data communication

Different from the synchronization operations, the one-sided data communication (put/get) cannot make use of shared memory mechanism, because the buffers where these operations perform are passed from user programs. Usually they are not shared memory. Although MPI standard defines a function MPI_Alloc_mem allowing users to allocate special memory that is shared by other processes on a SMP node,



(a) Truly one-sided synchronization

(b) Truly one-sided data communication

Fig. 3 Truly one-sided intra-node RMA design

we should not assume that the users always use it. Henceforth, we take advantage of the kernel-assisted means.

With the help of kernel, put or get operations can directly transfer the data to or from the target window. Operations are transparent to the target. Figure 3(b) presents our design. Every process traps into kernel to extract the information about its own window and maps this back to a user space structure. Then all the intra-node processes exchange this information. These two steps happen in MPI_Win_create. When an origin tries to issue a get/put (e.g., process 1 issues a *put* to process 0 in Fig. 3(b)), it only needs to retrieve the target window information (w0), thereby providing both this information and its local buffer information to the kernel for performing data copy. Regarding the data copy, as mentioned in Sect. 2.2, there are two direct copy approaches, i.e., the basic kernel-assisted approach and the I/OAT-assisted approach. We implement both the versions. It is to be noted that in some of the existing designs, the data transfer uses send-receive operations which also employ the basic kernel-assisted one-copy method for large messages [11, 19], but every time it has to go through *rendezvous* handshake prior to copy.

4.4 Other design issues

We have to address several additional issues to obtain good performance with kernel-assisted approach.

First, during the copy operation, the buffer pages should be locked in main memory to avoid being swapped to disk. This is mandatory for I/OAT based copy, because DMA engine directly deals with physical addresses. Thus, both the buffers at origin and the window at target are locked. While for the basic kernel-assisted approach, only the target window buffer is locked. We use the kernel API *get_user_pages* for this locking step.

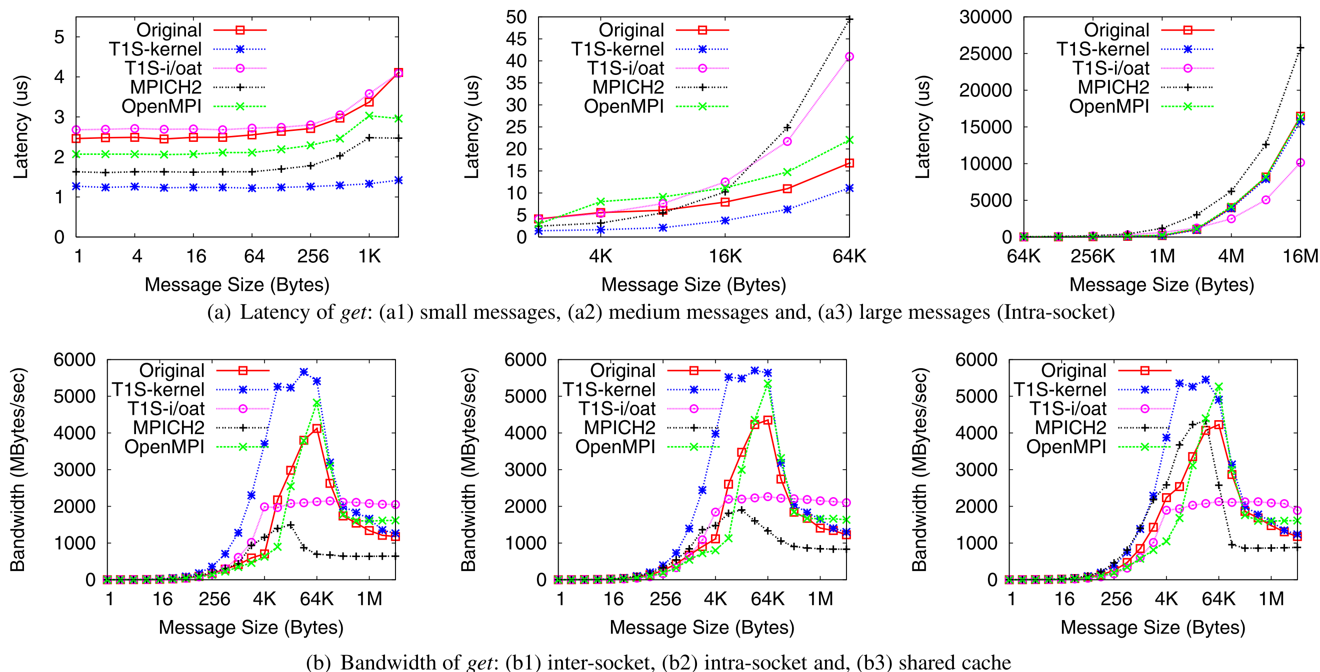
The high cost of locking pages may degrade the performance if every time the buffers are locked before the copy. In order to alleviate this, the locked pages are cached inside the kernel module upon being added for the first time. Next time the same buffer is not locked again. However, only the pages of window memory are cached. The local sending or receiving buffer are not cached, because they usually change as the application proceeds. For the memory allocated by *malloc*(), the cached pages must be released before the memory is freed, so we simply do not cache these pages.

Another issue about I/OAT is completion notification. After issuing copy requests, I/OAT returns cookies that can be polled for completion. Frequent polling is not desirable, so polling is performed only when the origin needs to write completion to the target after it issues all data transfer operations.

5 Experimental evaluation

In this section, we present comprehensive experimental evaluations and analysis. In all of the results figures, “Original” represents the existing design in MVAPICH2, “T1S-kernel” and “T1S-i/oat” represent the basic kernel-assisted version and the I/OAT-assisted version of our truly one-sided design. We primarily compare the performance of these three designs. In Sects. 5.1 and 5.2, we also show the results of MPICH2 and OpenMPI for more comparative study.

Experimental test bed We use three types of multi-core hosts. Type A is Intel Clovertown node with dual-socket quad-core Xeon E5345 processor (2.33 GHz). It has shared L2 cache between each pair of two cores. Type B is Intel Nehalem node with dual-socket quad-core Xeon E5530 processor (2.40 GHz) which has exclusive L2 cache for each core. Type C is AMD Barcelona host with quad-socket quad-core



(a) Latency of *get*: (a1) small messages, (a2) medium messages and, (a3) large messages (Intra-socket)

(b) Bandwidth of *get*: (b1) inter-socket, (b2) intra-socket and, (b3) shared cache

Fig. 4 Basic performance on Intel Clovertown

Opteron 8350 processor having exclusive L2 cache. There are different kinds of intra-node communication types. Type A node has inter-socket (two processes are on different sockets), intra-socket (two processes on the same socket with no shared L2 cache) and shared-cache (two processes on the same socket with shared L2 cache) communication. Nodes of type B and C only have inter-socket and intra-socket communication.

5.1 Improved latency and bandwidth

We use the RMA microbenchmarks in OMB [5] suite to measure the intra-node latency and bandwidth. Note that this benchmark uses aligned buffers for better performance. The performance with latest MPICH2 trunk and OpenMPI trunk is also measured for comparison. Please note that MVAPICH2, MPICH2 and OpenMPI all use two-sided based design. MVAPICH2 uses LiMIC2 underneath. Similarly, OpenMPI uses KNEM underneath. However, MPICH2 does not have KNEM on its RMA communication path, although uses it for the generic send-receive operations.

The ping-pong latency test measures the average one-way latency of an epoch. We experiment with all aforementioned intra-node communication types. Figures 4(a) show the results for intra-socket *get* on an Intel Clovertown host. Comparing with the existing designs, our kernel-assisted design greatly reduces the latency for small and medium messages by more than 39% and 30%, respectively. This is because that our design removes the inter-dependency between origin and target, and saves one copy of synchronization and data communication messages. However, for large messages, the data communication dominates the latency. Since MVAPICH2 and OpenMPI also use kernel-assisted copy underneath (although in an indirect manner), they have the similar performance as our design. MPICH2 shows worst performance starting from medium messages, because it uses the two-copy shared memory mechanism. On the other hand, our I/OAT-assisted design performs the worst for small and medium messages due to the high start up cost, but yields up to 38% better performance for very large messages (beyond 1 MB). It is because that I/OAT copies data in larger blocks, and has less CPU consumption and less cache pollution. We see similar results for *put* latency which is not presented here due to the space limit. For inter-socket and shared-cache communication, the performance trends remain the same. Please find more details in our technical report [20].

Figures 4(b) shows the bandwidth of *get* (*put* presents similar comparison) of different intra-node communication types. In this test, the target calls post and wait, while the origin process calls start-get's(put's)-complete. 32 get's/put's are issued back-to-back on non-overlapped locations of the target window. Use the inter-socket case as an example (Fig. 4(b1)). We see that the basic kernel-assisted design improves the bandwidth dramatically for medium messages where the I/OAT-assisted design performs badly. However, beyond the message size of 256K, I/OAT-assisted design performs the best with the improvement up to 29%. This suggests us design a hybrid method which can switch between these two alternatives as a future work. MPICH2

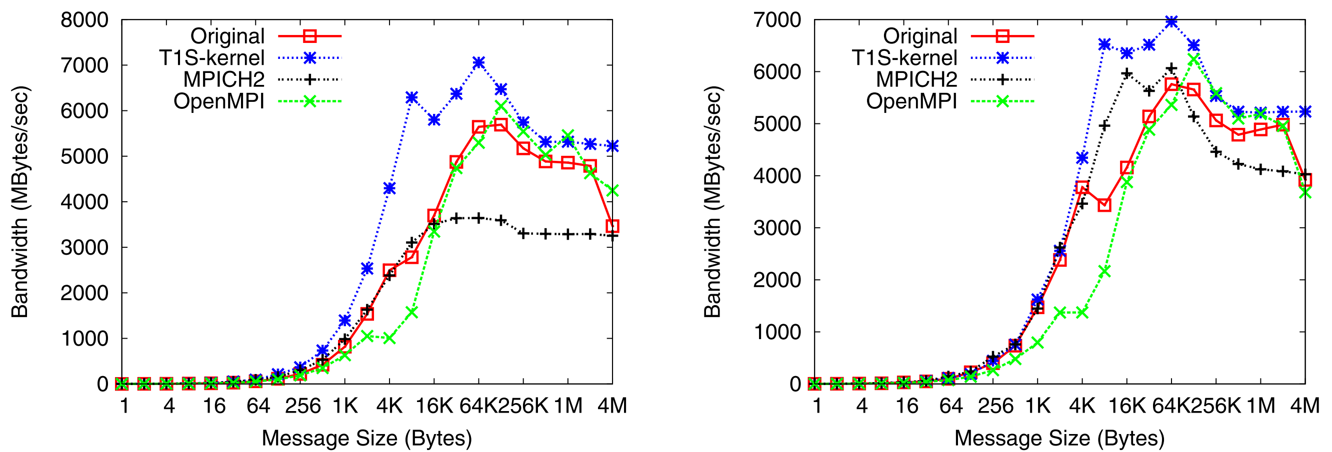


Fig. 5 Bandwidth of one-sided *get* of (**a**) Inter-socket and (**b**) Intra-socket (Intel Nehalem)

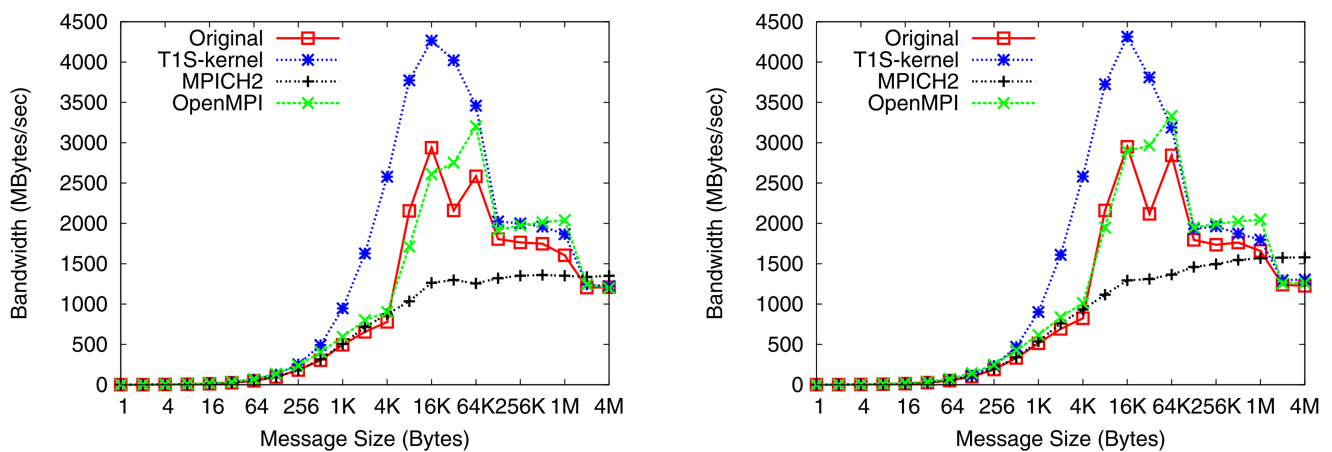


Fig. 6 Bandwidth of one-sided *get* of (**a**) Inter-socket and (**b**) Intra-socket (AMD Barcelona)

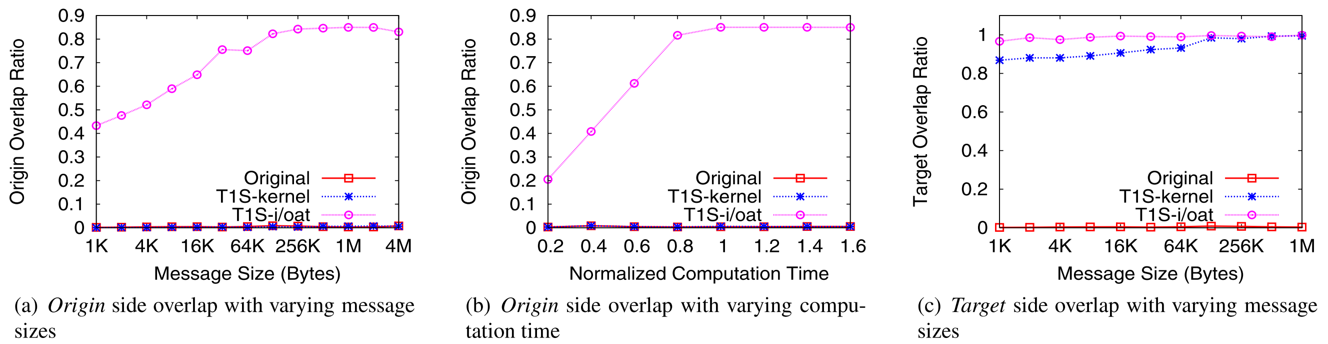
has very low bandwidth limited by the two-copy overhead. We observe very similar comparison in the inter-socket and shared-cache cases. However, in shared-cache case, MPICH2 has similar bandwidth as the MVAPICH2 and OpenMPI, mainly due to the greatly reduced data copy time within the cache.

To examine the impact of various multi-core architectures, we also measured the performance on Type B and C nodes. The chipsets on these two kinds of nodes do not support I/OAT, so I/OAT-assisted design is not shown. Here we only present part of the results. Please refer to [20] for details. Figures 5(a) and (b) show the *get* bandwidth in inter-socket and intra-socket cases on Intel Nehalem host. Similarly, Figs. 6(a) and (b) show the bandwidth on AMD Barcelona node. We find that although the absolute bandwidth varies with different multi-core processors and intra-node communication types, the new design always performs the best for a range of medium messages.

All of the above results demonstrate that our new design can obtain significantly improved latency and bandwidth on various multi-core architectures. We have seen that MPICH2 does not perform well and OpenMPI basically has the similar design as the current MVAPICH2. Since our new design is implemented in MVAPICH2, we only compare our new design ("T1S-kernel", "T1S-i/oat") with the "Original" MVAPICH2 design for the most part of the remaining results.

5.2 Reduced impact of process skew

In this benchmark, some amount of computation (in the form of matrix multiplication) is inserted between *post* and *wait* on the target to emulate the skew. The origin process performs *start*, 16 back-to-back *put* operations and *complete*. The time to finish these operations is measured. It is essentially the latency before the origin can proceed with other work. Please note that since we study the intra-node communication, inserting computation on the target not only introduces process skew but also adds background workload.



(a) *Origin* side overlap with varying message sizes

(b) *Origin* side overlap with varying computation time

(c) *Target* side overlap with varying message sizes

Fig. 7 Computation and communication overlap ratio

Table 1 Latency (μsec) of 16 *put* with process skew

Matrix size	0×0	32×32	64×64	128×128	256×256
MVAPICH2	3404	3780	6126	27023	194467
MPICH2	4615	4675	4815	24906	192848
OpenMPI	3804	3898	6563	27381	194560
T1S-kernel	3365	3333	3398	3390	3572
T1S-i/oat	2291	2298	2310	2331	2389

As representative examples, we list the results for *put* message of 256 KB in Table 1. The basic latency with minimum process skew (corresponding to the matrix size of 0×0) is also presented for reference. For the existing designs in MVAPICH2, MPICH2 and OpenMPI, we see that the latency shoots up as two processes become more skewed. This is because of the dependency on the target. Contrarily, our new design is more robust. The basic kernel-assisted design only has small degradation, and the I/OAT-assisted design has even less change. It means that the origin can proceed with the followed work irrespective of whether the target is busy or not. It is because our design is truly one-sided in which the computation on the target does not block the progress on the origin. Furthermore, I/OAT based design offloads the data copy so that the background workload has little impact. We also measure the time on the target [20] which still shows that our design has better performance.

5.3 Increased computation and communication overlap

Latency hiding and computation/communication overlap are one of the major goals in parallel computing. We investigate this through a put/get bandwidth test. At the origin, some computation is added after 16 back-to-back get/put operations for overlapping purpose. For a particular message size, the latency of 16 put/get is first measured. This basic latency is used as the reference for the inserted computation time. For example, if the basic latency is *Tcomm*, the computation time *Tcomp* should be equal or larger than *Tcomm* to achieve good overlap. The actual total latency is reported as *Ttotal*. We tested this on a type A host for example. The overlap is defined as:

$$Overlap = (Tcomm + Tcomp - Ttotal)/Tcomm$$

If the computation and communication are completely overlapped, the overlap should be 1 (because *Tcomp*=*Ttotal* in this case). Generally, the smaller the value is, the less overlap it has.

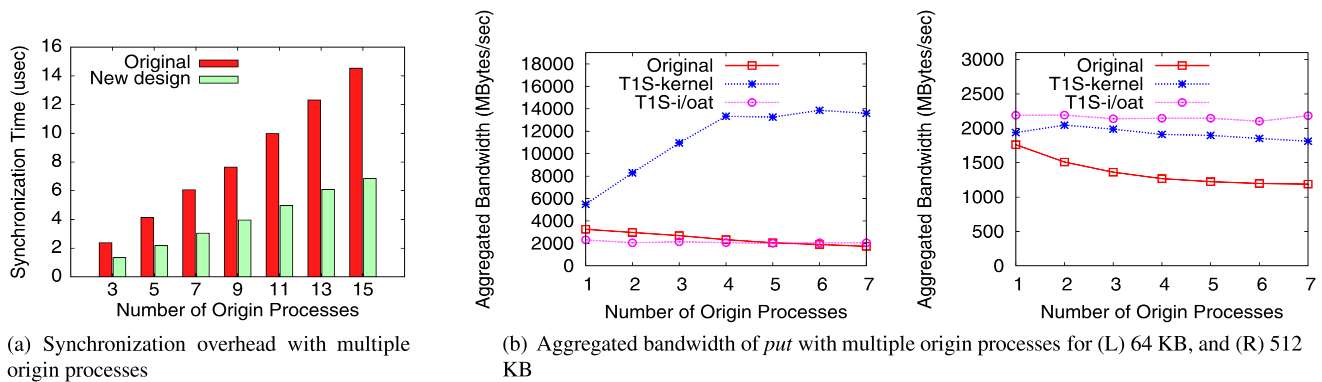
Figure 7(a) compares the origin side overlap with varying messages and $Tcomp = 1.2{*}Tcomm$. It clearly shows that the I/OAT based design provides close to 90% overlap, but the original design and the basic kernel-assisted design have no overlap at all. The reason is that I/OAT offloading releases the CPU so that the computation and communication can be executed simultaneously. Figure 7(b) illustrates the overlap change with increasing *Tcomp* for message of 1 MB. It conveys the same information that only I/OAT based design provides the origin side overlap.

Similarly, to examine the target side overlap, computation is inserted between post and wait just as we did in Sect. 5.2, but here we measure the time on the target. The overlap with $Tcomp = 1.2{*}Tcomm$ is shown in Fig. 7(c). We find that both versions of our new design can achieve almost full overlap, while the original design has no overlap. This is expected as our design aims at truly one-sided where the target can do its own computation while the communication is going on simultaneously.

5.4 Improved scalability

In some applications, multiple origin processes communicate with one target process. It is very important for a design to provide scalable performance in this situation. We use two experiments to evaluate this aspect.

The first experiment is to measure the synchronization overhead. One target process creates different windows for different origin processes and issues post and wait. Each origin issues start and complete without any put/get in between. We measure the average synchronization time at target as presented in Fig. 8(a) (this is on one AMD Barcelona host). Because the two versions of our new design have



(a) Synchronization overhead with multiple origin processes

(b) Aggregated bandwidth of *put* with multiple origin processes for (L) 64 KB, and (R) 512 KB

Fig. 8 Scalability performance

the same synchronization mechanism, we use "new design" to represent both. We see that the new design has much lower synchronization overhead. The improvement consistently remains about 50% with increasing number of origin processes. The truly one-sided nature decouples the origin and the target and reduces the work on the target, so it is more capable of handling multiple processes. We observe similar behavior using multiple targets and one origin.

The second experiment is similar, but now each origin issues 16 put operations and the aggregated bandwidth is reported. It is tested on a type A node. Figures in 8(b) ((L) and (R)) illustrate the results for the message size of 64 KB and 512 KB, respectively. We find that the existing design actually has decreasing bandwidth as the number of origin processes increases. It is because that both the synchronization and data communication require the participation from the target. As the number of origin processes increases, the target becomes the bottleneck. On the contrary, the kernel-assisted truly one-sided design provides increasing aggregate bandwidth until it reaches the peak. After that, it also tends to decrease because of the cache and memory contention. The I/OAT based design has the consistently low (for 64 KB) or high (for 512 KB) bandwidth. I/OAT copy does not consume many CPU cycles and does not pollute cache, so its performance is not disturbed by the number of origin processes. In the same way, we experimented with multiple targets and one origin in which the new design also shows better scaling.

5.5 Decreased cache misses

Our work emphasizes on the intra-node communication, so the cache effect also plays an important role. We used the Linux *oprofile* tool (with sampling rate of 1:500) to measure the L2 cache misses during the aggregated bandwidth test used in the last section. The test has seven origin processes and one target to occupy all the cores. Figure 9 compares the L2 cache miss samples with varying *put* message sizes.

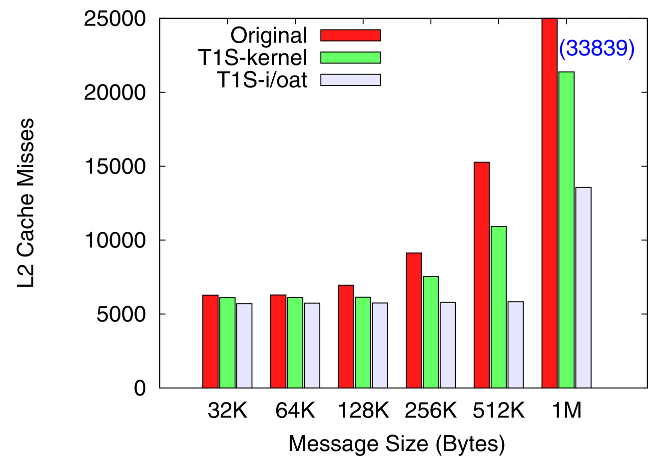


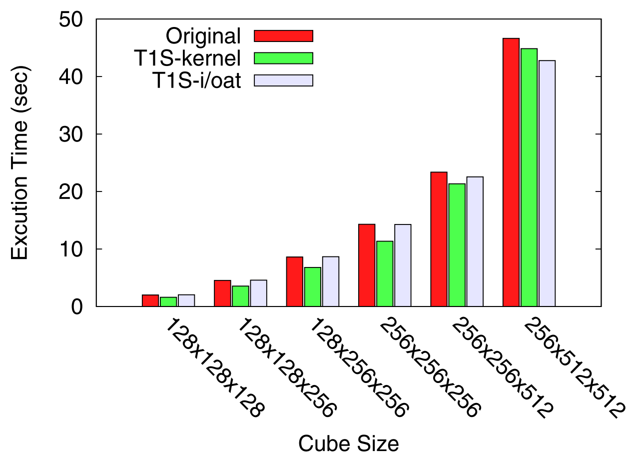
Fig. 9 L2 Cache misses

Obviously, the I/OAT-assisted design has the least cache misses, because it greatly reduces the cache pollution. The basic kernel-assisted design reduces the copies in synchronization, caches the locked pages and removes the interdependent interaction between the origin and target, therefore it also has much less cache misses. Note that for 1 MB messages, we use the label instead of a full bar for original MVAPICH2 design, as the number is too large.

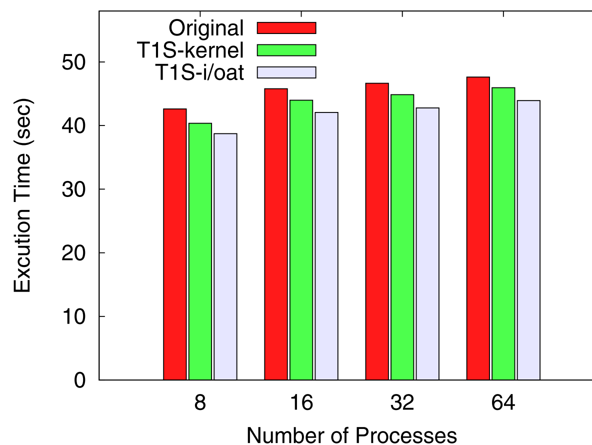
5.6 Improved application performance

We use a real scientific application AWM-Olsen to evaluate the design. AWM-Olsen is stencil-based earthquake simulation from the Southern California Earthquake Center [13]. Processes are involved in nearest-neighbor communication followed by a global sum and computation. They execute on a three-dimensional data cube. AWM was originally written in MPI send-receive. We modified it to use MPI-2 one-sided semantics and arranged the computation and communication for higher overlap potential.

We run the application on Clovertown hosts with 8 processes on each node, and measure the execution time



(a) Performance with varying data cube



(b) Weak scaling performance

Fig. 10 Performance of AWM-Olsen application

of the main step. Figure 10(a) shows the performance of 32 processes with varying data cube sizes. Our kernel-assisted design outperforms the original design about 15% for medium range of data cube, while the I/OAT-assisted version provides around 10% benefits for very large data cube. Figure 10(b) shows the weak scaling performance with varying process counts. The data cube increases as the processes increase such that the data grid per process remains $128 \times 128 \times 128$ elements. We see our new design provides stable improvement as the system size increases.

6 Related work

Ever since RMA communication was introduced into MPI-2 standard, many implementations have incorporated the complete or partial design. MPICH2 [24] and LAM/MPI [2] designed the RMA communication on top of two-sided operations. MVAPICH2 has implemented a truly one-sided design [17] with special optimization on passive synchronization [18, 23], but it only applies to the inter-node communication. In [22], a true one-sided passive synchronization scheme is designed for multi-core based systems, but it does not deal with active synchronization and data transfer operations. OpenMPI also exploits alternate ways including send-receive, buffered and RMA [8], but again it also does not provide truly one-sided intra-node communication. SUN MPI provides the SMP based one-sided communication [10], but it requires all the processes be on the same node and use MPI_Alloc_mem. NEC-SX MPI [16] implements truly one-sided communication specially making use of global shared memory over Giganet cluster. There are some other works exploiting the truly RMA possibilities on particular platforms [7, 9].

The papers [11, 12, 15, 19] present different approaches including the kernel-assisted and I/OAT-assisted one copy approaches to design two-sided intra-node communication. It is to be noted that although KNEM is used in some two-sided functions in MPICH2, those functions are not used for implementing the one-sided communication.

Our work in this paper differentiates from these previous works by focusing on designing intra-node truly one-sided communication for both synchronization and data communication over the commodity multi-core architecture.

7 Conclusions and future work

In this paper, we proposed the design of truly one-sided communication within a node. We first analyzed the inadequacy of the existing two-sided based design, based on which, we designed and implemented truly one-sided data communication using two alternatives (the basic kernel-assisted and the I/OAT-assisted truly one-sided approach), and enhanced existing shared memory mechanisms for truly one-sided synchronization. The new design eliminates the overhead related with two-sided operations. We evaluated our design on three multi-core systems. The results show that our new design greatly decreases the latency by over 39% for small and medium messages and increases the large message bandwidth by up to 29%. We further designed a series of experiments to characterize the resilience to process skew, computation and communication overlap, the scalability, and the L2 cache effect. In all of these experiments, our new design presents superior performance than the existing designs. Finally, we used a real scientific application AWM-Olsen to demonstrate its application level benefits.

In the future we plan to investigate more efficient hybrid design (as mentioned in Sect. 5.1) and to study more

aspects of one-sided communication (i.e., truly one-sided MPI_Accumulate). In addition, we plan to do evaluation and analysis on other platforms and do large-scale evaluations. We also plan to use other applications to carry out in-depth studies on how the improvements in intra-node one-sided communication can benefit the application performance. Finally, we would also like to examine the similarity and difference between MPI-2 one-sided communication and other one-sided models such as UPC [6] languages.

Software distribution The proposed new design will be available to community in the next MVAPICH2 [4] release.

References

1. I/OAT Acceleration Technology. http://www.intel.com/network/connectivity/vtc_ioat.htm
2. LAM/MPI Parallel Computing. http://www.lam-mpi.org/
3. MPICH2: High Performance and Widely Portable MPI. http://www.mcs.anl.gov/research/projects/mpich2/
4. MVAPICH2: MPI over InfiniBand and iWARP. http://mvapich.cse.ohio-state.edu/
5. OSU Microbenchmarks. http://mvapich.cse.ohio-state.edu/benchmarks/
6. Unified Parallel C. http://en.wikipedia.org/wiki/Unified_Parallel_C
7. Asai N, Kentemich T, Lagier P (1999) MPI-2 implementation on Fujitsu generic message passing kernel. In: Proceedings of the ACM/IEEE conference on supercomputing (CDROM)
8. Barrett BW, Shipman GM, Lumsdaine A (2007) Analysis of implementation options for MPI-2 one-sided. In: EuroPVM/MPI
9. Bertozzi M, Panella M, Reggiani M (2001) Design of a VIA based communication protocol for LAM/MPI suite. In: Euromicro workshop on parallel and distributed processing
10. Booth S, Mourao E (2000) Single sided MPI implementations for SUN MPI. In: Proceedings of the 2000 ACM/IEEE conference on supercomputing
11. Buntinas D, Goglin B, Goodell D, Mercier G, Moreaud S (2009) Cache-efficient, intranode, large-message MPI communication with MPICH2-Nemesis. In: International conference on parallel processing (ICPP)
12. Chai L, Lai P, Jin H-W, Panda DK (2008) Designing an efficient kernel-level and user-level hybrid approach for MPI intra-node communication on multi-core systems. In: International conference on parallel processing (ICPP)
13. Cui Y, Moore R, Olsen K, Chourasia A, Maechling P, Minster B, Day S, Hu Y, Zhu J, Jordan T Toward petascale earthquake simulations. In: Special issue on geodynamic modeling, vol. 4, July 2009
14. Forum M (1993) MPI: a message passing interface. In: Proceedings of supercomputing
15. Goglin B (2009) High throughput intra-node MPI communication with Open-MX. In: Proceedings of the 17th Euromicro international conference on parallel, distributed and network-based processing (PDP)
16. Träff JL, Ritzdorf H, Hempel R (2000) The implementation of MPI-2 one-sided communication for the NEC SX-5. In: Proceedings of the 2000 ACM/IEEE conference on supercomputing
17. Jiang W, Liu J, Jin H, Panda DK, Gropp W, Thakur R (2004) High performance MPI-2 one-sided communication over infiniband. In: IEEE/ACM international symposium on cluster computing and the grid (CCGrid 04)
18. Jiang W, Liu JX, Jin H-W, Panda DK, Buntinas D, Thakur R, Gropp W (2004) Efficient implementation of MPI-2 passive one-sided communication on infiniBand clusters. In: EuroPVM/MPI
19. Jin H-W, Sur S, Chai L, Panda DK (2008) Lightweight kernel-level primitives for high-performance MPI intra-node communication over multi-core systems. In: IEEE international sympsoium on cluster computing and the grid
20. Lai P, Panda DK (2009) Designing truly one-sided MPI-2 RMA intra-node communication on multi-core systems. In: Technical Report OSU-CISRC-9/09-TR46, Computer Science and Engineering, The Ohio State University
21. Message Passing Interface Forum. MPI-2: extensions to the message-passing interface, July 1997
22. Santhanaraman G, Balaji P, Gopalakrishnan K, Thakur R, Gropp WD, Panda DK (2009) Natively supporting true one-sided communication in MPI on multi-core systems with infiniband. In: IEEE international sympsoium on cluster computing and the grid
23. Santhanaraman G, Narravula S, Panda DK (2008) Designing passive synchronization for MPI-2 one-sided communication to maximize overlap. In: Int'l Parallel and Distributed Processing Symposium (IPDPS)
24. Thakur R, Gropp W, Toonen B (2005) Optimizing the synchronization operations in message passing interface one-sided communication. Int J High Perform Comput Appl

Ping Lai is a Ph.D. candidate in Computer Science & Engineering Department at the Ohio State University. Her primary research interests include high performance computing, communication protocols and high performance data-centers. She has published (including co-authored) about 10 papers in journals and conferences related to these research areas. She is a member of the Network-Based Computing Laboratory lead by Professor D.K. Panda.

Sayantan Sur is a Research Scientist at the Department of Computer Science at The Ohio State University. His research interests include high speed interconnection networks, high performance computing, fault tolerance and parallel computer architecture. He has published more than 18 papers in major conferences and journals related to these research areas. He is a member of the Network-Based Computing Laboratory lead by Dr. D.K. Panda. He is currently collaborating with National Laboratories and leading InfiniBand and iWARP companies on designing various subsystems of next generation high performance computing platforms. He has contributed significantly to the MVAPICH/MVAPICH2 (High Performance MPI over InfiniBand and 10GigE/iWARP) open-source software packages. The software developed as a part of this effort is currently used by over 1075 organizations in 56 countries. In the past, he has held the position of Post-doctoral researcher at IBM T. J. Watson Research Center, Hawthorne and Member Technical Staff at Sun Microsystems. Dr. Sur received his Ph.D. degree from The Ohio State University in 2007.

Dhabaleswar K. Panda is a Professor of Computer Science at the Ohio State University. His research interests include parallel computer architecture, high performance computing, communication protocols, files systems, network-based computing, Virtualization and Quality of Service. He has published over 260 papers (including multiple Best Paper Awards) in major journals and international conferences related to these research areas. Dr. Panda and his research group members have been doing extensive research on modern networking technologies including InfiniBand, 10GigE/iWARP and RDMA over Ethernet (RDMAoE). His research group is currently collaborating with National Laboratories and leading InfiniBand and iWARP companies on designing various subsystems of next generation high-end systems. The MVAPICH/MVAPICH2 (High Performance MPI over InfiniBand and 10GigE/iWARP) open-source software packages, developed by his research group (http://mvapich.cse.ohio-state.edu), are currently being used by more than 1075 organizations worldwide (in 56 countries). This software has enabled several InfiniBand clusters (including 5th ranked Tianhe-1 in China and 9th ranked TACC Ranger) to get into the latest TOP500 ranking. These software packages are also available with the Open Fabrics stack for network vendors (InfiniBand, iWARP and RDMAoE), server vendors and Linux distributors. Dr. Panda's research is supported by funding from US National Science Foundation, US Department of Energy, and several industry including Intel, Cisco, SUN, Mellanox, and QLogic. Dr. Panda is an IEEE Fellow and a member of ACM.