Designing Truly One-Sided MPI-2 RMA Intra-node Communication on Multi-core Systems

PING LAI, DHABALESWAR K. PANDA

Technical Report
OSU-CISRC-9/09-TR46.

Designing Truly One-Sided MPI-2 RMA Intra-node Communication on Multi-core Systems

Ping Lai D. K. Panda
Department of Computer Science and Engineering
The Ohio State University
{laipi, panda}@cse.ohio-state.edu

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 defines one-sided communication facilitated with synchronization operations. The existing designs for intra-node RMA communication are built on top of two-sided send/recv operations. They suffer from the two-sided inherent overhead and the close dependency on the remote side. In this paper, we propose two kernel based direct copy alternatives, i.e., the basic kernel-assisted approach and the I/OAT-assisted approach, to design the truly one-sided intra-node communication. In addition, we utilize the shared memory mechanism to achieve the truly one-sided synchronization. The new design eliminates the overhead of two-sided operations and removes the involvement from the remote side. We also propose a series of benchmarks to evaluate various performance aspects over multi-core-based 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. The performance is quite stable irrespective of the intra-node communication types. Moreover, it presents superior performance in terms of better scalability, less cache misses, more resilience to process skew and higher computation and communication overlap. Finally, up to 10% performance benefits is demonstrated for a real scientific application AWM-Olsen.

1. Introduction

Parallel scientific computing has been growing dramatically these years. It drives the faster development of newer technologies, the massive deployment of workstation clusters and the revolutionary improvement of programming models.

Multi-core technology is one of the main contributors 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 June 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 a node. It suggests that the intra-node communication design of a programming model will play a key role in the overall performance.

In the last decade MPI (Message Passing Interface) [22] has evolved as one of the most popular programming models for distributed memory systems. MPI-1 specification defines the 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 the communication. The synchronization is done implicitly in progress engine. To fulfill the need of many scientific applications that have dynamically changing data distributions and data access patterns, MPI-2 [23] standard introduces the one-sided communication model also known as Remote Memory Access (RMA) model. In this model, ideally only one process participates in the communication, so it has to specify all the communication parameters including the parameters (such as memory address) on the remote side. Synchronization is done explicitly to guarantee the communication completion. Here the process initiating the communication is called *origin*, and the remote peer process is called *target*. MPI-2 currently supports three one-sided operations, i.e., *MPI_Put*, *MPI_Get* and *MPI_Accumulate*, and two synchronization modes, i.e., active mode and passive mode.

There can be different ways to design and implement the one-sided model. One way is to build it on top of the existing two-sided operations. This approach has good portability, but meanwhile suffers from some overhead. It includes intermediate layer handover, two-sided inherent overhead (including more data copies, send/recv matching and *rendezvous* negotiation etc.) and the remote process involvement. Several popular MPI implementations such as MPICH2[2] and LAM/MPI [1] use this two-sided based approach. The second approach is to utilize special functions such as RDMA operations to achieve direct one-sided communication. MVAPICH2 [21, 3] uses this design and shows better performance. However, all of these designs mainly focus on optimizing the inter-node RMA communication. Even in the second direct one-sided approach, it falls back to the two-sided based design if it is the intra-node communication. This could significantly degrade the overall performance due to the increasing importance of intra-node communication and higher overhead of the two-sided based approach. Therefore, it is necessary to design more efficient intra-node one-sided communication.

In this paper we design and implement a truly one-sided RMA model for intra-node communication, and carry out comprehensive evaluations and analysis. We propose two alternatives for truly one-sided data communication. One is based on the kernel-assisted direct copy and the other one further takes help from the modern I/OAT [17] technology to offload this copy. *MPI_Put* and *MPI_Get* are naturally mapped to the direct copy without interrupting the *target*. This design eliminates the two-sided operation related overhead. More importantly, since the target is not involved, its progress does not block the communication. [1] For synchronization, as the passive mode has been investigated in [19, 26], we only deal with the active mode. Shared memory mechanism is utilized to realize the one-sided property in synchronization. We come up with several benchmarks running on three multi-core-based 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. Also, the performance is not much affected by the intra-node communication types (i.e., inter-socket, intra-socket or shared-cache). Further, we see that the new design achieves better scalability, less cache misses and more computation and communication overlap. It is also more tolerant to the process skew and offers more benefits in real applications.

The rest of this paper is organized as follows. In Section 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 Section 3. In Section 4, we describe the proposed design in detail. We present and analyze the experimental results in Section 5, discuss the related work in Section 6, and summarize conclusions and possible future work in Section 7.

[1] Please note that currently *MPI_Accumulate* is not considered. We plan to incorporate it in the future.

2. Background

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

2.1. MPI-2 RMA Communication

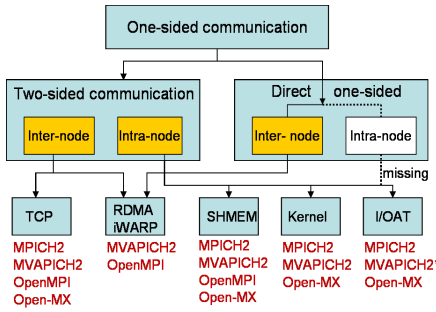
MPI-2 RMA communication (i.e., one-sided communication) model defines that the *origin* process can directly access the memory area on the *target* process. This memory area is called *window* which is defined in MPI_Win_create. Ideally the origin process specifies all the parameters including the target memory address and target data types, etc, so the target is unaware of the on-going communication.

As mentioned in Section 1, MPI-2 defines three RMA operations. MPI_Put and MPI_Get transfer the data to and from a window on a target. MPI_Accumulate combines the data movement to target with a reduce operation. The operations are not guaranteed to complete when these functions return. The completion must be ensured using explicit synchronization primitives. In other words, MPI semantics allows one-sided operations only within an *epoch* which is the period between two synchronization events. The synchronization is classified as *passive* (requiring no explicit participation of 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 further classified into two types: a) collective MPI_Win_fence on the entire group; and b) collective on a smaller user-defined group, i.e., origin process uses MPI_Win_start and MPI_Win_complete to specify an *access epoch* to a group of targets, and the target calls MPI_Win_post and MPI_Win_wait to specify an *exposure epoch* to a group of origins. The origin can issue RMA operations only when it knows that the target window has been posted, and the target can complete an epoch only when it knows that all the origins in the group have finished accessing on its window. Normally multiple one-sided operations are issued in an epoch to amortize the synchronization overhead. In this paper, we primarily concentrate on the active synchronization and use the post-wait/start-complete synchronization as the example in the following sections.

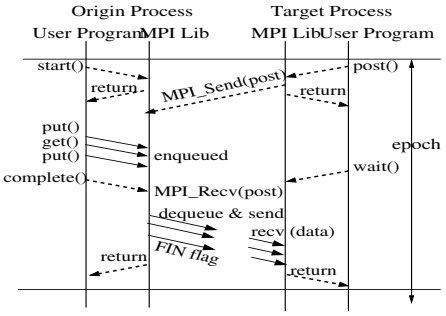
2.2. Generic Mechanisms for Intra-node Communication

There are several common mechanisms for intra-node communication. The easiest one is user space shared memory approach. Two processes sharing a buffer can communicate with copy-in (from sender buffer to shared memory) and copy-out (from shared memory to receiver buffer) operations. This approach usually provides benefits for small messages, while not for large messages due to the two copies overhead. As it must involve two parties, normally it is not a good candidate for designing one-sided communication.

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 [20] is used in MVAPICH2 [21] and KNEM [9] is used in MPICH2 [2]. Based on this approach, another alternative is to further offload the direct copy to DMA (Direct Memory Access) engine. I/OAT [17] developed by Intel is such a DMA engine. It is a PCI resource having multiple independent DMA channels with direct access to main memory. It copies the data asynchronously while releasing the CPU for other work. 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 with the buffer information about itself and *target*, the data can be directly copied to the *target* without interrupting it. On the other hand, they have the cost of trapping into the kernel, pinning down the memory, and polling the completion (for I/OAT), etc.



(a) MPI-2 one-sided communication design in various implementations



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

Figure 1. Analysis of Existing Designs

3. Analysis of Existing Designs

As illustrated in Figure 1(a), various MPI implementations [2] design MPI-2 RMA communication using two means, i.e., based on two-sided send/recv operations [3] and direct one-sided. Direct one-sided approach bypasses the two-sided operations to directly build the RMA communication over the underlying remote memory access mechanisms (e.g., RDMA mechanism available on some modern interconnects, or node level RMA mechanisms). This approach is truly one-sided and usually offers more benefits. While the inter-node direct one-sided design has been incorporated into some implementations, to the best of our knowledge the intra-node truly one-sided design is still missing.

Regarding the existing two-sided based designs for intra-node RMA, let us see an example in MPICH2. It employs a deferred method [27] as shown in Figure 1(b) where the dotted lines represent synchronization steps and the solid lines mean data communication steps. At the origin, MPI_Win_start and the put's/get's return immediately. The put/get operations are just queued locally and will be issued in 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 as compared to other two-sided based designs, it is not one-sided in nature and still has several drawbacks.

Since all the synchronization and data communication go through the send/recv path, it is unavoidable to inherit the two-sided overhead. For instance, short messages will 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 direct dependency on the target process's progress, which is contrary to the goal of one-sided model. This together with the fact that all the one-sided operations are deferred to be issued could result in very bad performance if the origin and target processes are skewed. We experimented this with a benchmark in which some amount of computation (in the form of matrix multiplication) is inserted between MPI_Win_post and MPI_Win_wait on the target to emulate the skew. The origin performs a sequence of MPI_Win_start, 16 back-to-back MPI_Put operations and MPI_Win_complete. We measure the time to finish these operations when running two processes within a node. It is essentially the latency before the origin can proceed with other work. Table 1 lists the results for *put* message size of 256 KB using several popular MPI-2 implementations. The basic latency without any process skew is also presented for reference. We can see that as the computation time at the target increases (more skew), the

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

[3] Please note that I/OAT support for two-sided communication is not officially included in the current MVAPICH2 release. It is on the scheduled plan.

execution time at the origin shoots up quickly. This is resulted from its dependency on the target. All of these observations suggest the great demand on designing a truly one-sided intra-node communication.

Table 1. Execution time (usec) of 16 *put* with increasing process skew

Matrix size	0x0(base)	32x32	64x64	128x128	256x256
MVAPICH2	3404	3780	6126	27023	194467
MPICH2	4615	4675	4815	24906	192848
OpenMPI	3804	3898	6563	27381	194560

4. Proposed Design and Implementation

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

4.1. Goal of the New Design

First we present the design goal and overall architecture. Figure 2 shows the new design we intend to target. Basically we aim to realize the truly one-sided nature for both synchronization operations (post and complete) and the communication operations (put and get), thereby removing the two-sided related overhead and alleviating the impact of process skew. All the functions should be able to proceed as soon as they are called, independent of the remote side response. ”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 Section 2.2) mechanisms as the basis for the design.

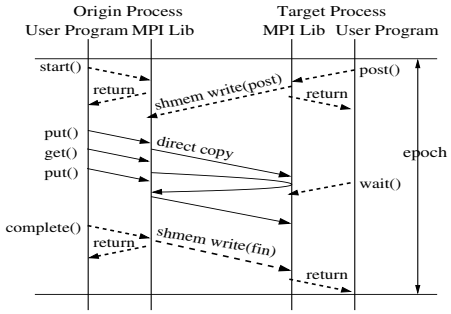


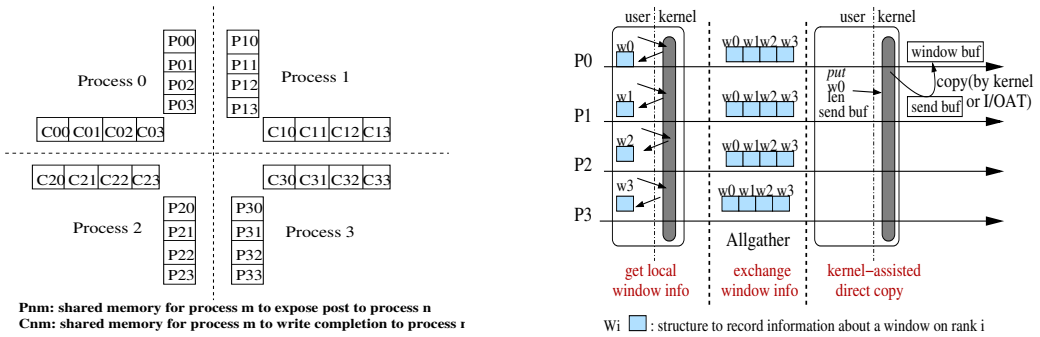
Figure 2. Goal of the new design

Our design is implemented in MVAPICH2[21] which currently uses the send/recv based deferred approach as described in Section 3. We make changes on CH3 layer to design a customized interface for intra-node direct one-sided communications. It utilizes the underlying intra-node communication means that are specially designed for this situation.

4.2. Design of Truly One-Sided Synchronization

As mentioned earlier, we use the post-wait and 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 the direct one-sided design, as illustrated in Figure 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



(a) Truly one-sided synchronization (b) Truly one-sided data communication

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

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” in 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 will check 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. One is that it requires only one copy for ”post” step. The other is that it is truly one-sided without depending 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, we should not assume that the users will 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 calls kernel module function 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. After target posts the window and an origin tries to issue a get/put (e.g., process 1 issues a *put* to process 0 in Figure 3(b)), it only needs to find the target window information (w0), thereby providing both this information and its local buffer information to the kernel. Regarding the actual data copy, as mentioned in Section 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 the original design, the data transfer uses two-sided send/recv operations which also employs the basic kernel-assisted one-copy method for large messages [20], 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 them to be 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 first 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. Then 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, considering that 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 the 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 after the last put/get operation and before the origin writes completion to the target.

The last issue is that I/OAT copies the data from page to page. If the buffers are not aligned, the number of copy operations could be two times of the number of pages. Thus, we can expect much better performance using aligned buffers. In our experimental evaluations, all of the microbenchmarks have aligned memory while the real application is not guaranteed on this.

5. Experimental Evaluation

In this section, we present comprehensive experimental evaluations and analysis. We first present the basic put/get latency and bandwidth. Then we characterize the scalability performance, the cache effect, the impact of process skew, and the computation and communication overlap. Finally, we use a real application AWM-Olsen [12] to show the application level performance. In all of the following 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 Section 5.1, 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 Opteron 8350 processor having exclusive L2 cache. Based on these different multi-core architectures, there are various kinds of intra-node communication. Type A node has inter-socket (processes are on different sockets), intra-socket (two processes on the same socket without shared L2 cache) and shared-cache communication. Nodes of type B and C only have inter-socket and intra-socket communication. Most of our experiments are carried out using type A nodes.

5.1. Latency and Bandwidth Performance

We use the RMA microbenchmarks in OMB [24] suite to measure the latency and bandwidth of intra-node one-sided put/get by running two processes on a single node. The performance with MPICH2 (using their latest Nemesis channel) and OpenMPI (using the best run-time tuning we can get for their one-sided communication-OSC component) is also measured for comparison.

In the ping-pong latency test, one process performs a sequence of post-wait-start-get(put)-complete (two epochs). Correspondingly, the other process performs start-get(put)-complete-post-wait. This pattern repeats for many iterations. The benchmark reports the averaged latency for one epoch. We experiment with all aforementioned intra-node communication types. Figures 5(a), (b) and (c) show the results for intra-socket *get* on an Intel Clovertown host. Comparing with the existing designs, our basic kernel-assisted design greatly reduces the latency for small and medium messages by more than 39% and 30%, respectively, while showing similar performance as MVAPICH2 for large messages. This is because that for small and medium messages, our design saves one copy of synchronization and data messages. The locked page caching also improves the performance. However, for large messages, the data communication dominates the latency, and the send/recv in MVAPICH2 also uses kernel-assisted direct copy underneath. MPICH2 and OpenMPI show worst performance starting from medium messages, because they use 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 it yields much better performance (by up to 29%) for very large messages (beyond 1 MB). We believe it is because that I/OAT copies data in blocks larger than that in the general copy. We see the similar results for *put* latency which is not presented in this paper due to the space limit. For inter-socket and shared-cache communication, the performance trends remain the same, except that MPICH2 and OpenMPI perform better in shared-cache case while degrade in inter-socket case as compared to their performance in the intra-socket case, which is mainly due to the different data access time. The results are shown in Figures 7 and Figures 9.

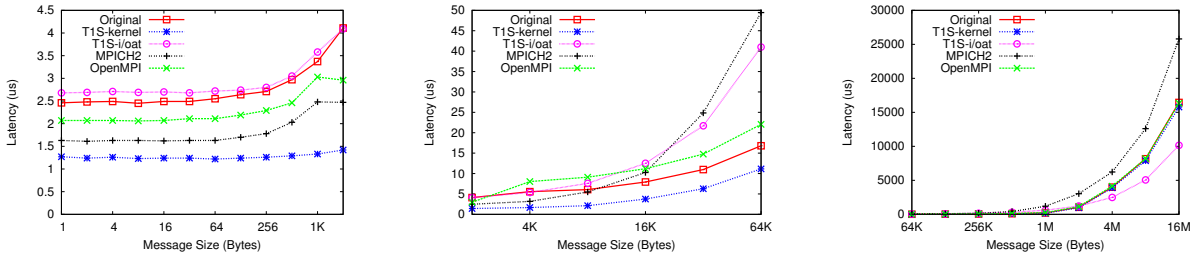


Figure 4. Latency of *get*: (a) small messages, (b) medium messages and, (c) large messages (Intel Clovertown, Intra-socket)

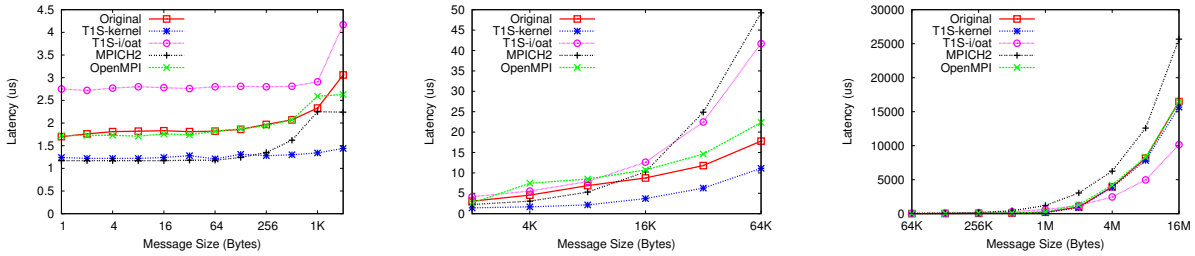


Figure 5. Latency of *put*: (a) small messages, (b) medium messages and, (c) large messages (Intel Clovertown, Intra-socket)

Figures 6 (a) and (b) show the bandwidth of *get* and *put* in the intra-socket case. In the bandwidth test, one process is always the target who calls post and wait, while the other process calls start-get's(put's)-complete. In each iteration the get/put are issued for multiple times (named as burst size) on non-overlapped locations in the target window, e.g., here we use 32 continuous put/get. We see that the basic kernel-assisted design improves the bandwidth dramatically for small to medium message range

where the I/OAT-assisted design performs the worst. However, beyond the message size of 256K, I/OAT-assisted design performs the best with the improvement up to 64%, thanks to the multiple channels in I/OAT for parallel copies. It is also because I/OAT has less CPU consumption and it operates directly on memory without being affected by cache contention, but other designs will have a lot of cache misses with messages larger than 128 KB (128KB*32=4MB is the L2 cache size shared by two cores). MPICH2 and OpenMPI have very low bandwidth limited by the two-copy overhead and cache-cache transaction time. We observe very similar comparison in the inter-socket case in Figures 8. However, the shared-cache case displays some differences as shown in Figures 10 (a) and (b). Two versions of our new design and MVAPICH2 still remain the similar performance. MPICH2 and OpenMPI get higher bandwidth than that they obtain in the inter-socket case, mainly due to the greatly reduced data copy time inside cache.

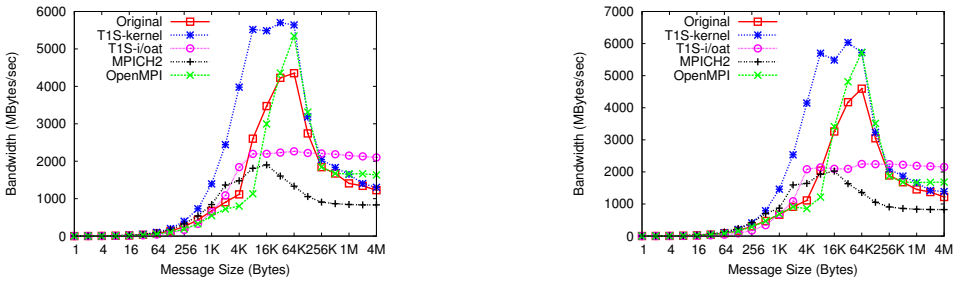


Figure 6. Bandwidth of one-sided operations: (a) *get*, and (b) *put* (Intel Clovertown, Intra-socket)

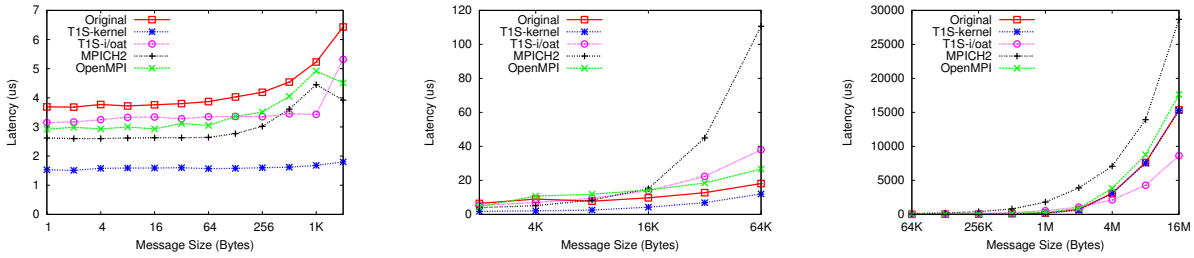


Figure 7. Latency of *get*: (a) small messages, (b) medium messages and, (c) large messages (Intel Clovertown, Inter-socket)

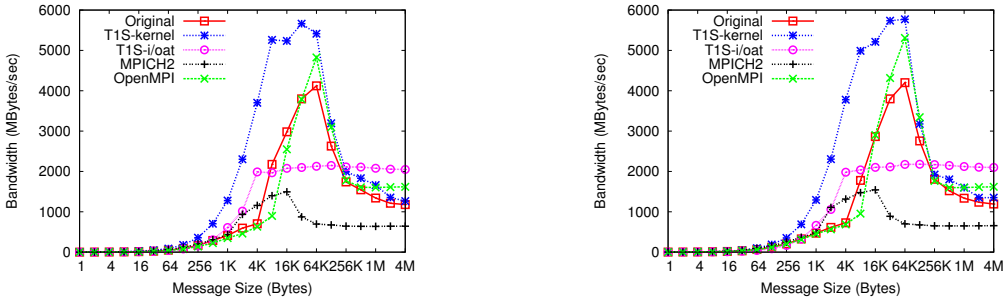


Figure 8. Bandwidth of one-sided operations: (a) *get*, and (b) *put* (Intel Clovertown,Inter-socket)

To examine the impact of different multi-core architectures, we also measured the latency and bandwidth on Type B and C nodes under inter-socket and intra-socket situations. The chipsets on these two nodes do not support I/OAT, so we do not examine the performance of the I/OAT-assisted design. Here

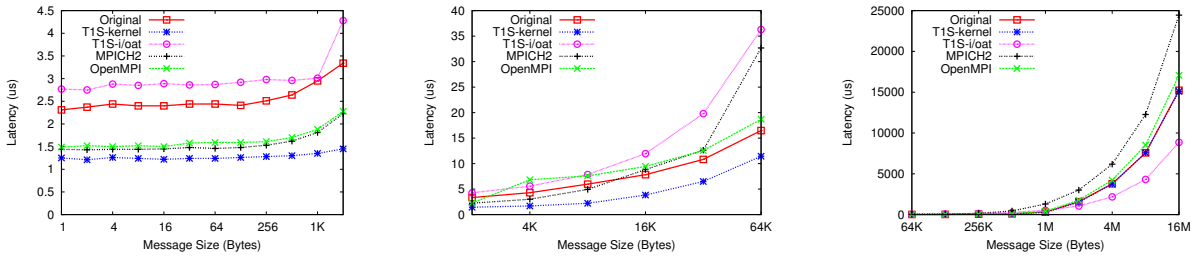


Figure 9. Latency of *get*: (a) small messages, (b) medium messages and, (c) large messages (Intel Clovertown, Shared-cache)

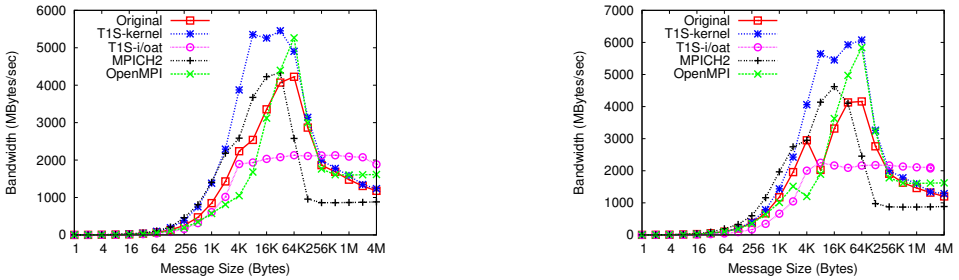


Figure 10. Bandwidth of one-sided operations: (a) *get*, and (b) *put* (Intel Clovertown, Shared-cache)

we only present part of the results. Figures 12 (a), (b) and (c) present the inter-socket *get* latency on Nehalem host. Our design still exhibits lower latency. Owing to the improved multi-core architecture, the absolute latency is much lower than that on the Clovertown architecture. Figures 13 (a) and (b) show the *get* bandwidth in inter-socket and intra-socket cases, respectively. As the tests on type A host, our new design has stable highest throughput. Similarly, Figures 15 (a), (b) and (c) show the *get* latency, and Figures 16 (a) and (b) show the bandwidth on AMD Barcelona node. We also find that our kernel-assisted direct one-sided design performs the best for most cases.

All of the above results demonstrate that our new design can obtain significantly improved latency and throughput on various multi-core architectures, and the performance stays quite stable irrespective of the intra-node communication types.

Our new design is implemented in MVAPICH2. For the remaining part of the paper, we mainly compare our new design (”T1S-kernel”, ”T1S-i/oat”) with the ”Original” MVAPICH2 design.

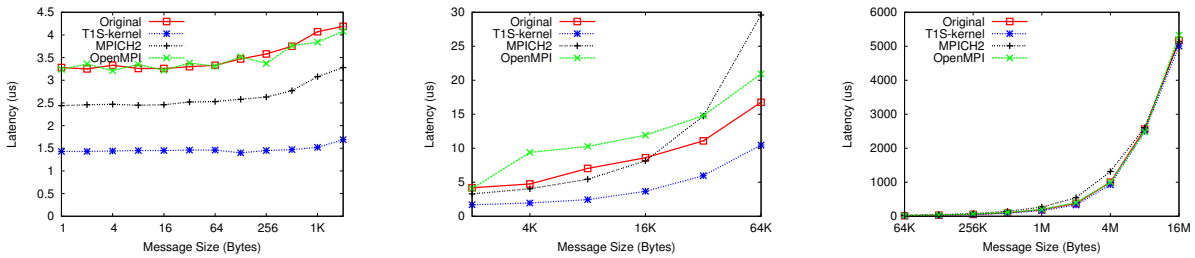


Figure 11. Latency of *get*: (a) small messages, (b) medium messages and, (c) large messages (Intel Nehalem, Inter-socket)

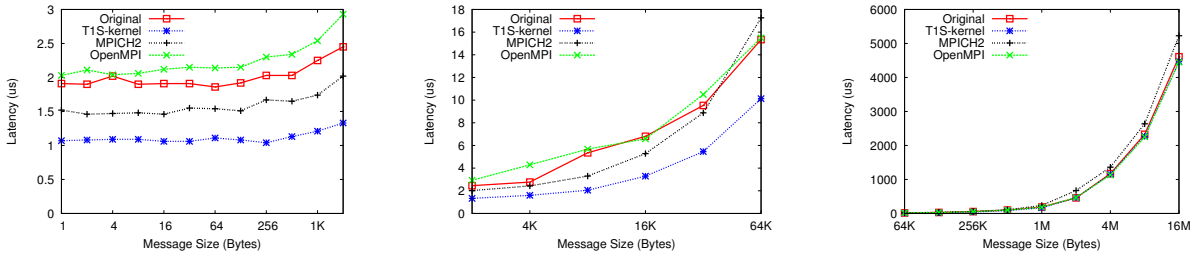


Figure 12. Latency of *get*: (a) small messages, (b) medium messages and, (c) large messages (Intel Nehalem, Intra-socket)

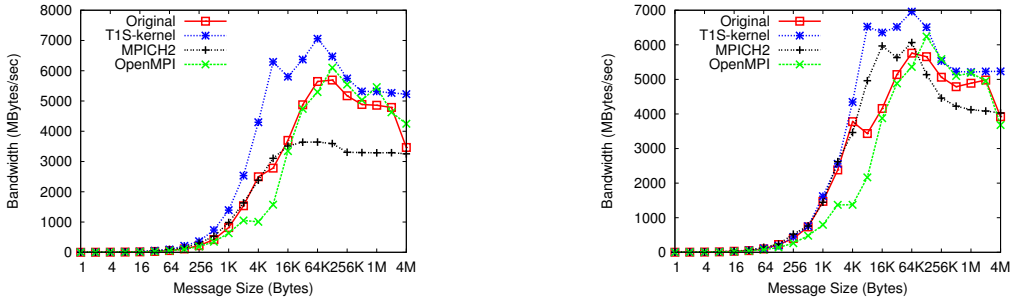


Figure 13. Bandwidth of one-sided *get* of (a) Inter-socket and (b) Intra-socket. (Intel Nehalem)

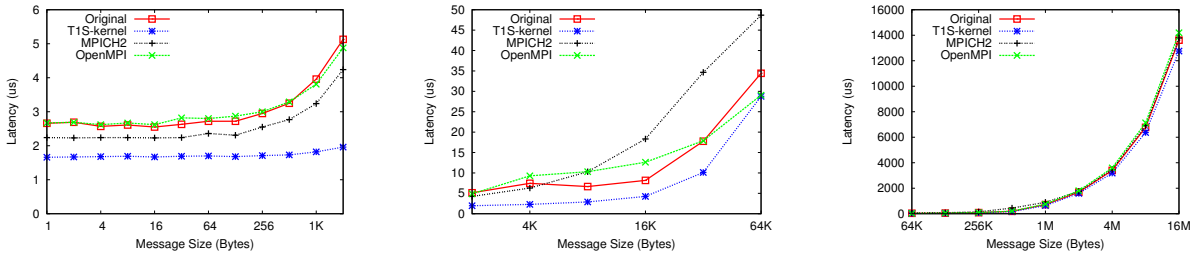


Figure 14. Latency of *get*: (a) small messages, (b) medium messages and, (c) large messages (AMD Barcelona, Intra-socket)

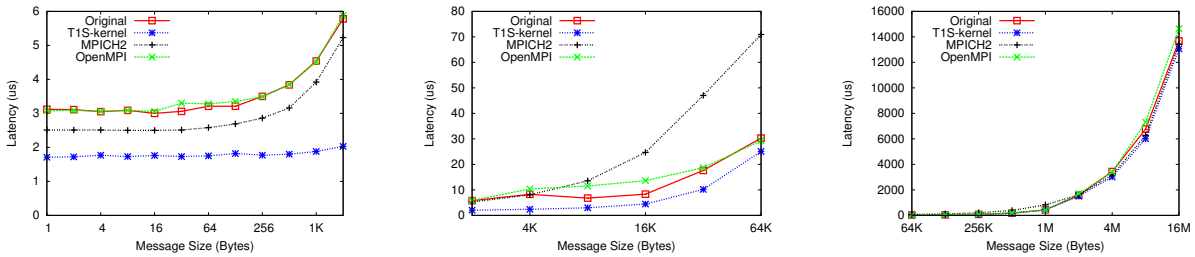


Figure 15. Latency of *get*: (a) small messages, (b) medium messages and, (c) large messages (AMD Barcelona, Inter-socket)

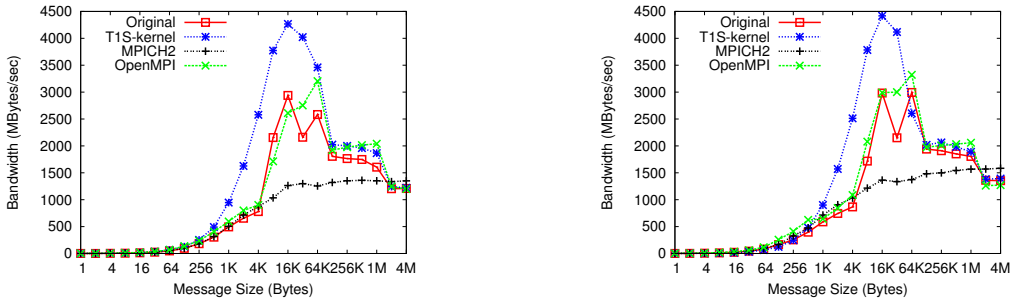


Figure 16. Bandwidth of one-sided *get* of (a) Inter-socket and (b) Intra-socket.(AMD Barcelona)

5.2. Scalability Performance

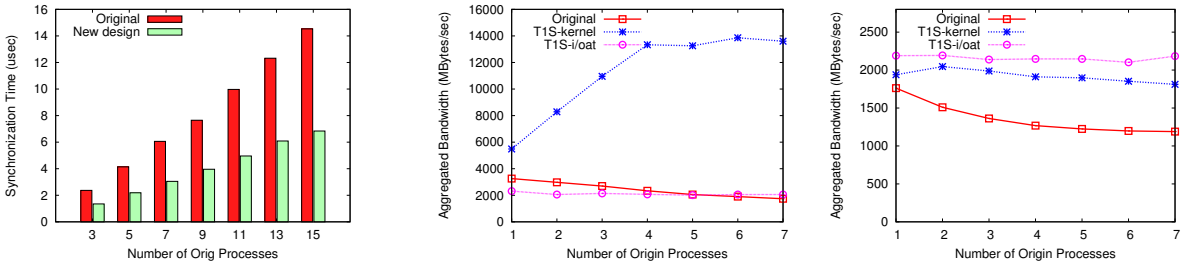
In some applications, multiple origin processes communicate with one target process. It is very important for a design to provide salable good performance in this situation. We use two experiments to evaluate this aspect.

The first experiment is to measure the synchronization overhead. One target process creates different windows for different origin processes and performs post and wait. Each origin process issues start and complete without any put/get in between. This test runs on one AMD Barcelona host. We measure the average time at target process as presented in Figure 17(a). Because the two versions of our new design have the same synchronization mechanism, we use ”new design” to represent both of them. 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 also consists of multiple origin processes and one target process. The difference is that now each origin issues a burst of 16 put operations to the target and the aggregated bandwidth is reported. It is tested on a type A node. Figures in 17(b) ((L) and (R)) illustrate the results for the message size of 64 KB and 512 KB, respectively. We find that the original 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 direct 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. It is due to the reason that I/OAT copy does not consume many CPU cycles and does not pollute cache, so its performance is not disturbed as the number of origin processes increases. These results prove that the new design offers more scalable performance.

5.3. Cache Effect

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 runs with seven origin processes and one target to occupy all the cores. Figure 18 compares the L2 cache miss samples with varying *put* message sizes. Obviously, the I/OAT-assisted design has the least cache misses, because it greatly decreases the cache pollution. The basic kernel-assisted design reduces the copies in synchronization, caches the locked pages and removes the inter-dependent interaction between the origin and target, therefore it also has much less cache misses. Note that for the 1 MB message, we use the label instead



(a) Synchronization overhead with multiple origin processes

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

Figure 17. Scalability performance

of a full bar for original MVAPICH2 design, as the number is too large.

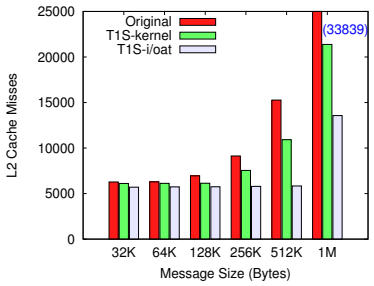


Figure 18. L2 Cache misses

5.4. The Impact of Process Skew

We have presented in Section 3 that the existing designs perform bad under the process skew situation. We ran the same benchmark on a type A host to examine this effect on the new design. Please note that since we study the intra-node communication, the method of inserting computation on the target not only introduces process skew but also adds more background workload.

As representative examples, we list the results for *put* message of 256 KB in Table 2. For the original design, earlier we have seen that the latency shoots up as two processes become more skewed. Here we see that our new design is more robust against the process skew. 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 that our design is truly one-sided in which the delay on the target does not block the progress on the origin. Furthermore, I/OAT based design offloads the data copy so that increasing background workload has little impact. It shows larger opportunity to hide the latency which will be detailed in the next section. We also measure the time on the target (included in Table 3) which still shows that our design has better performance.

Table 2. *Put* time (usec) with increasing process skew

Matrix size	0x0(base)	32x32	64x64	128x128	256x256
MVAPICH2	3404	3780	6126	27023	194467
T1S-kernel	3365	3333	3398	3390	3572
T1S-i/oat	2291	2298	2310	2331	2389

Table 3. *Put* time (usec) with increasing process skew at *target*

Matrix size	0x0(base)	32x32	64x64	128x128	256x256
MVAPICH2	3400	3775	6121	27018	194461
T1S-kernel	3369	3236	3401	23570	190663
T1S-i/oat	2294	2288	2696	23328	190740

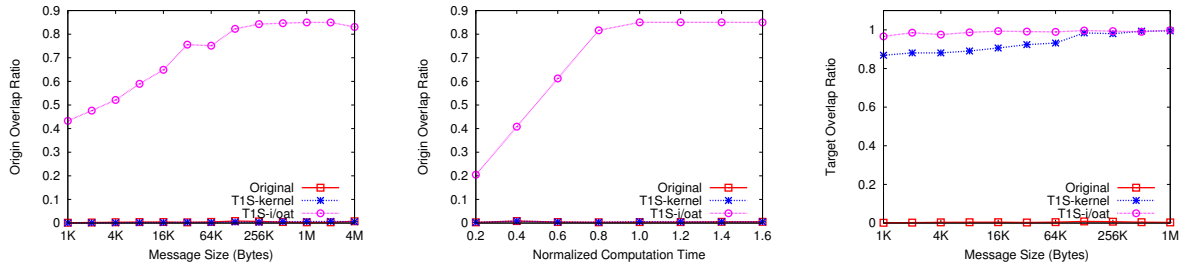
5.5. Computation and Communication Overlap

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 side, some amount of computation is added after a burst of 16 continuous 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 experiment on a type A host. The origin side overlap is defined as:

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

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

Figure 19(a) compares the overlap efficiency with varying messages and *Tcomp*=1.2**Tcomm*. It clearly shows that the I/OAT based design can provide 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 the copy can be executed simultaneously. Figure 19(b) illustrates the overlap ratio change with the increasing *Tcomp* for message of 1 MB. It conveys the same information that only I/OAT based design provides the origin side overlap.



(a) *Origin* side overlap with varying message sizes (b) *Origin* side overlap with varying computation time (c) *Target* side overlap with varying message sizes

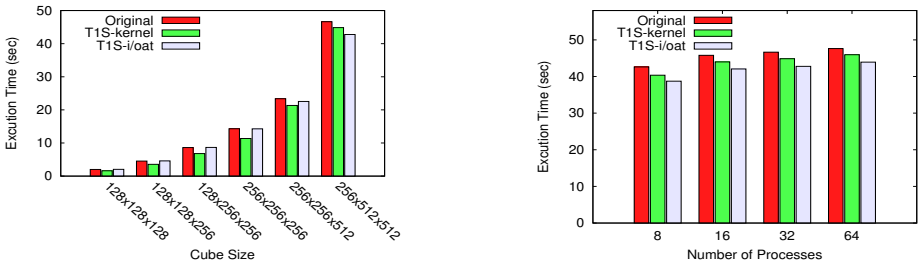
Figure 19. Computation and communication overlap ratio

Similarly, to examine the target side overlap, some computation is inserted between post and wait just as we did in Section 5.4, but here we measure the time on the target. The overlap with *Tcomp*=1.2**Tcomm* is shown in Figure 19(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 implicitly.

5.6. 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 [12]. 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.



(a) Performance with varying data cube (b) Weak scaling performance

Figure 20. Performance of AWM-Olsen application

We run the application on Clovertown hosts with 8 processes on each node, and measure the execution time of the main step. Figure 20(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 20(b) shows the weak scaling performance with varying process counts. The data cube increases as the the processes increase such that the data grid per process remains 128x128x128 elements. We see our new design provides stable improvement as the system size increases.

6. Related Work

Ever since the one-sided RMA communication was introduced into MPI-2 standard, many implementations have incorporated the complete or partial design. MPICH2 [27] and LAM/MPI [1] designed the RMA communication on top of two-sided operations. MVAPICH2 has implemented a direct one-sided design [18] with special optimization on passive synchronization [19, 26], but it only applies to the inter-node communication. Open MPI also exploits alternate ways including send/recv, buffered and RMA [5], but again it also focuses on the inter-node communication. SUN MPI provides the SMP based one-sided communication [8], but it requires all the processes be on the same node and use MPI_Alloc_mem. NEC-SX MPI [28] implements truly one-sided communication specially making use of global shared memory over Giganet cluster. There are some other works exploiting the direct RMA possibilities on particular platforms [6, 4]. Researchers in [13] compare some existing implementations. Besides MPI, there are other one-sided programming models such as ARMCI[25], GASNET[7] and BSP[16].

The papers [11, 10, 20, 9, 15] present different approaches including the kernel-assisted approaches to design two-sided intra-node communication. Authors study various uses of I/OAT technology in [29, 30, 14].

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 generic architecture.

7. Conclusions and Future Work

Ever increasing popularity of multi-core processors has demonstrated the importance of intra-node communication in MPI design, which is also true for MPI-2 one-sided RMA communication. In this paper, we propose 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 two alternatives (the basic kernel-assisted direct copy approach and the I/OAT-assisted direct copy approach)

for truly one-sided data communication, and utilized the shared memory mechanism for truly one-sided synchronization. The new design eliminates the overhead related with two-sided operations. It also realizes the truly one-sided property by removing the interactive dependency between origins and targets. We evaluated our design over three multi-core-based architectures. The results show that our new design greatly decreases the latency over 39% for small and medium messages and increases the large message bandwidth by up to 29%. The performance is stable irrespective of the processes distribution within a node. We further designed a series of experiments to characterize the scalability performance, the L2 cache effect, the resilience to process skew, and the computation and communication overlap. In all of these experiments, our new design presents superior performance than the existing designs. Finally, we use a real scientific application AWM-Olsen to demonstrate its application level benefits.

In the future we plan to study more aspects of one-sided communication (i.e., design truly one-sided MPI_Accumulate) and to investigate more efficient hybrid design. In addition, we plan to do evaluation and analysis on other platforms and do large-scale evaluations. We also plan to use some other applications to carry out more in-depth studies on how the improvements in intra-node one-sided communication can benefit the application performance.

Software Distribution: The proposed new design is planned to be made available to community in the next MVAPICH2 [3] release.

References

[1] LAM/MPI Parallel Computing. http://www.lam-mpi.org/.

[2] MPICH2: High Performance and Widely Portable MPI. http://www.mcs.anl.gov/research/projects/mpich2/.

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

[4] N. Asai, T. Kentemich, and P. Lagier. MPI-2 implementation on Fujitsu generic message passing kernel. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, 1999.

[5] B. W. Barrett, G. M. Shipman, and A. Lumsdaine. Analysis of Implementation Options for MPI-2 One-Sided. In *EuroPVM/MPI*, 2007.

[6] M. Bertozzi, M. Panella, and M. Reggiani. Design of a VIA based communication protocol for LAM/MPI Suite. In *Euromicro Workshop on Parallel and Distributed Processing* , 2001.

[7] D. Bonachea. GASNet Specification v1.1. In *Technical Report UCB/CSD-02-1207, Computer Science Division, University of California at Berkeley*, 2002.

[8] S. Booth and E. Mourao. Single Sided MPI Implementations for SUN MPI. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, 2000.

[9] D. Buntinas, B. Goglin, D. Goodell, G. Mercier, and S. Moreaud. Cache-Efficient, Intranode, Large-Message MPI Communication with MPICH2-Nemesis. In *International Conference on Parallel Processing (ICPP)*, 2009.

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

[11] L. Chai, P. Lai, H.-W. Jin, and D. K. Panda. 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)*, 2008.

[12] Y. Cui, R. Moore, K. Olsen, A. Chourasia, P. Maechling, B. Minster, S. Day, Y. Hu, J. Zhu, and T. Jordan. Toward Petascale Earthquake Simulations. volume 4, July 2009.

[13] E. Gabriel, G. E. Fagg, and J. J. Dongarra. Evaluating Dynamic Communicators and One-Sided Operations for Current MPI Libraries. In *The International Journal of High Performance Computing Applications*, 2005.

[14] T. Gangadharappa, G. Santhanaraman, K. Gopalakrishnan, S. Potluri, and D. K. Panda. Improving MPI One-sided Passive Communication Using I/OAT Offload Engines. In *EuroPVM/MPI*, 2009.

[15] B. Goglin. 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)*, 2009.

[16] M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, and T. Tsantilas. Portable and Efficient Parallel Computing Using the BSP Model. In *IEEE Transactions on Computers*, 1999.

[17] A. Gover and C. Leech. Accelerating Network Receiver Processing. http://www.kernel.org/doc/ols/2005ols2005v1-pages-289-296.pdf/.

[18] W. Jiang, J. Liu, H. Jin, D. K. Panda, W. Gropp, and R. Thakur. High Performance MPI-2 One-Sided Communication over InfiniBand. In *IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 04)*, Chicago, IL, April 2004.

[19] W. Jiang, J. X. Liu, H.-W. Jin, D. K. Panda, D. Buntinas, R. Thakur, and W. Gropp. Efficient Implementation of MPI-2 Passive One-sided Communication on InfiniBand Clusters. In *EuroPVM/MPI*, 2004.

[20] H.-W. Jin, S. Sur, L. Chai, and D. K. Panda. Lightweight Kernel-level Primitives for High-Performance MPI Intra-Node Communication over Multi-Core Systems. In *IEEE International Sympsoium on Cluster Computing and the Grid*, 2008.

[21] J. Liu, W. Jiang, P. Wyckoff, D. K. Panda, D. Ashton, D. Buntinas, W. Gropp, and B. Toonen. Design and Implementation of MPICH2 over InfiniBand with RDMA Support. In *Proceedings of Int'l Parallel and Distributed Processing Symposium (IPDPS '04)*, April 2004.

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

[23] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*, Jul 1997.

[24] O. Microbenchmarks. http://mvapich.cse.ohio-state.edu/benchmarks/.

[25] J. Nieplocha and B. Carpenter. ARMCI:A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-Time Systems. In *Lecture Notes in Commputer Science*, 1999.

[26] G. Santhanaraman, S. Narravula, and D. K. Panda. Designing Passive Synchronization for MPI-2 One-Sided Communication to Maximize Overlap. In *Int'l Parallel and Distributed Processing Symposium (IPDPS)*, 2008.

[27] R. Thakur, W. Gropp, and B. Toonen. Optimizing the Synchronization Operations in Message Passing Interface One-Sided Communication. In *International Journal of High Performance Computing Applications*, 2005.

[28] J. L. Traff, H. Ritzdorf, and R. Hempel. The Implementation of MPI-2 One-sided Communication for the NEC SX-5. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, 2000.

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

[30] K. Vaidyanathan, W. Huang, L. Chai, and D. K. Panda. Designing Efficient Asynchronous Memory Operations Using Hardware Copy Engine: A Case Study with I/OAT. In *CAC*, 2007.