Noname manuscript No.
(will be inserted by the editor)

ProOnE: A General-Purpose Protocol Onload Engine for Multi- and Many-Core Architectures

P. Lai · P. Balaji · R. Thakur · D. K. Panda

Received: date / Accepted: date

Abstract Modern high-end computing systems utilize specialized offload engines to enhance various aspects of their processing. For example, high-speed networks such as InfiniBand, Quadrics and Myrinet utilize specialized hardware to offload network processing to help improve performance. However, such hardware units are expensive, and their manufacturing complexity increases exponentially depending on the number and complexity of tasks they offload. On the other hand, the proliferation of multi- and many-core processors into the general desktop and laptop markets is increasingly driving their cost down due to the economies of scale. To take advantage of the obvious benefits of multi/many-core architectures, we propose, design and evaluate **ProOnE**, a general purpose Protocol Onload Engine. ProOnE utilizes a small subset of the available cores on a multi-core CPU to "onload" various tasks in a dedicated manner instead of "offloading" them to specialized hardware. The general purpose processing capabilities of multi-core architectures allow ProOnE to be designed in a flexible, extensible and scalable manner, while benefiting from the reducing costs of general-purpose CPUs. In this paper, we onload onto ProOnE, several tasks relevant to communication sub-systems such as MPI that are too complex for current hardware offload engines to support, and demonstrate significant benefits in terms of overlap of computation and communication and improved application performance.

Keywords Protocol offload/onload, multi-core, many-core

This work was supported in part by the National Science Foundation Grant #0702182, and the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357; DOE grants #DE-FC02-06ER25749 and #DE-FC02-06ER25755; NSF grants #CNS-0403342, #CPA-0702675, and #CCF 0833169.

P. Lai · D. K. Panda
Computer Science and Engineering, Ohio State University, USA,
E-mail: laipi, panda@cse.ohio-state.edu

P. Balaji · R. Thakur
Mathematics and Computer Science, Argonne National Laboratory, USA,
E-mail: balaji, thakur@mcs.anl.gov

1 Introduction

High-end computing systems have benefited from the use of specialized accelerators [27] to improve their performance and scalability for many years. Network sub-systems were among the early adopters of such techniques, providing hardware-based solutions for intelligent communication offloading. GigaNet [7] was one of the earliest network offload solutions for the Virtual Interface Architecture (VIA) [1]. The early generations of Myrinet [2] used similar network processing solutions with specialized hardware. This trend was continued to modern high-speed networks, including InfiniBand [8], Quadrics [3] and TCP or iWARP offload engines [6].

While such hardware-based offload engines have continued to grow in performance and complexity, the requirement for even more advanced processing has grown as well. For example, with systems scaling to hundreds of thousands of cores available today, more and more complex tasks such as advanced network processing and data center services are required to be offloaded. However, the manufacturing complexity of such hardware units increases exponentially with the number and complexity of tasks they offload. As an instance, various communication processing tasks, such as zero-copy *Rendezvous* communication which is common for popular programming models like Message Passing Interface (MPI) [13], are too complicated for current hardware to handle, especially given the various corner cases supported by the MPI standard. Similarly, aspects of fault detection and process management also increase hardware complexity tremendously, thus making pure hardware offload solutions expensive and inflexible.

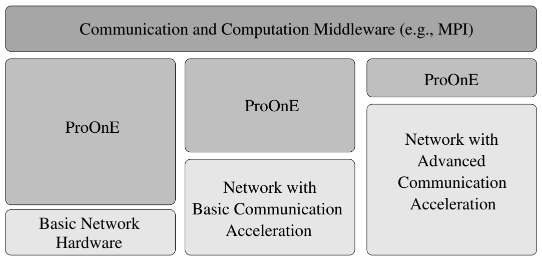

Communication and Computation Middleware (e.g., MPI)
ProOnE
ProOnE
ProOnE
Basic Network Hardware
Network with Basic Communication Acceleration
Network with Advanced Communication Acceleration

Fig. 1 **H**ybrid onload-offload architecture with ProOnE

On the other hand, multi- and many-core processors are being increasingly deployed in clusters and even widely used in desktops and laptops. Quad-core and Hex-core processors are quickly gaining ground in many applications. In fact, more than 80% of the systems in the November 2008 ranking of the Top500 supercomputers [12] belonged to the multi/many-core processor family. Future generation systems are getting further augmented with not only multiple cores per processor, but also with multiple hardware threads (SMTs) per core, as illustrated by the Intel eXtreme [4] and SUN Niagara [5] families of processors that promise to support up to 2048 threads within a single physical node in the near future.

Consequently, to take advantage of the obvious benefits of multi-core architectures, we propose, design and evaluate a general purpose Protocol Onload Engine, named as ***ProOnE***. ProOnE dedicates a small subset of the available cores on a multi-core equipped node to "onloading" complex tasks while still utilizing the features provided by hardware "offload" engines. In other words, as the example illustrated in Figure 1, ProOnE utilizes the existing offloaded features, but extends them by onloading more complex functionality that cannot be easily offloaded. The general purpose processing capabilities of multi-core architectures allow ProOnE to be designed in a flexible, extensible and scalable manner, while benefiting from the reducing costs. We present details of the design for the intra-node and inter-node communication interfaces, synchronization handshake between ProOnE and application processes, and various other aspects of the ProOnE design.

Further more, as a case study, we use ProOnE to onload complex communication tasks relevant to MPI. We specially focus on onloading the *Rendezvous* protocol for large message communication and present the design issues and our solutions. Our experimental results show that ProOnE can improve MPI communication significantly, allowing almost full overlap between communication and computation and close to 100% application availability for large message communication. We also demonstrate the resilience of ProOnE to process skew which is a major concern in large-scale systems, and illustrate significant application-level benefits as well.

The rest of this paper is organized as follows. In Section 2, we discuss the related work. Then we describe the design including the general ProOnE infrastructure and MPI *Rendezvous* protocol onload design in Section 3. We analyze the experimental results in Section 4, and summarize conclusions and possible future work in Section 5.

2 Related Work

The idea of protocol onloading is not new. There has been a lot of work that focusses on onloading different computation and communication related tasks such as TCP/IP processing in data centers [23,22], MPI collective operations [25], and distributed data management [30]. There has also been work that compares onloading and offloading based solutions for network communication [21]. However, all of this existing literature takes ad hoc approaches for onloading specific tasks relevant to their environment. In this paper, on the other hand, we propose a general purpose protocol onload engine that can onload any task using a few of the many available cores. Different tasks can be added to ProOnE using task-specific plug-ins, making it a central framework that allows all of the existing research and more to be plugged in.

There has also been a lot of research on improving communication and computation overlap and asynchronous progress for the MPI *Rendezvous* protocol. R. Brightwell et al. analyzed the impact of communication and computation overlap [15] providing theoretical insights into this problem. Sancho et al further quantified this for large-scale applications in [24]. Amerson et al. improved asynchronous progress with an interrupt based approach [14] and an event-driven MPI library was also designed to improve the communication responsiveness in [20]. In [28] and [19], the authors improved communication progress for zero-copy communication using interrupts and helper threads. Our work complements these efforts by utilizing a few cores on the system to provide the required computation and communication overlap, which in turn can be used by the existing MPI libraries.

Using helper threads to allow for communication progress has also been studied in [29] and the Myri-10G MX protocol [2]. However, these approaches primarily focus on the communication tasks and are not generic for all tasks. They need to be extended by taking into account more aspects in HPC, which is the objective ProOnE targets.

Thus, in summary, our work in this paper is orthogonal and complementary to the existing work in that we focus on designing a general purpose onload engine for onloading any protocol (task), e.g., communication protocol, middleware-specific tasks and application-specific tasks.

3 Designing a Protocol Onload Engine

In this section, we describe the design of ProOnE. We first present the general infrastructure, and then take the *Rendezvous* protocol in MPI as an example to illustrate how to onload a particular protocol.

3.1 The ProOnE Infrastructure

In parallel applications, all processes are inter-related, with each following the pattern of interleaved computation and communication. If a part of this work is to be onloaded by ProOnE, the application processes and the ProOnE processes need to be considered as a whole, instead of separately. Figure 2 shows the overall architecture for a system with four cores on each node where some of the cores are used by ProOnE in a dedicated manner (typically, only a small subset of cores is used by ProOnE). Each ProOnE process runs as a daemon that executes a part of the work for one or more application processes. To design such an infrastructure, we need to consider two aspects, namely, (i) the local intra-node communication and synchronization, and (ii) the inter-node communication.

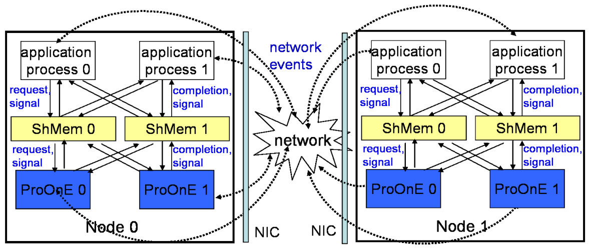


Fig. 2 Architecture of ProOnE-equipped system

3.1.1 Intra-node Communication Interface

We used a shared-memory based approach for intra-node communication and synchronization because of its simplicity. Specifically, an application or ProOnE process puts requests, signals or completions into shared memory regions from which other application or ProOnE processes can read. To reduce overhead, these requests or completions are managed as queues. Hash functions are used to locate the free shared memory blocks. As shown in Figure 2, each ProOnE process allocates one shared memory region which is used for the communication with all other processes on the same node.

3.1.2 Inter-node Communication Interface

The inter-node communication in Figure 2 goes through the traditional network messaging. Since each ProOnE process may need to communicate with any remote ProOnE or application process, there could be a large number of network connections needed. As a performance and resource usage tradeoff, these connections can be established either upon launching of the application processes or on demand.

3.1.3 ProOnE Runtime Infrastructure

Considering the above requirements, we design the runtime infrastructure for ProOnE as follows.

ProOnE processes: Each ProOnE process performs three tasks during initialization: (i) creates its own communication shared memory, (ii) attaches to the shared memory created by other ProOnE processes, and (iii) listens to and accepts connection requests from remote processes.

When a ProOnE process is initialized (e.g., ProOnE process 0 on node 0 in Figure 2), it first creates or attaches to (if one has been created) a global shared memory according to a predefined `shmem` key. This global shared memory is used to exchange process-specific information; we do not include this in Figure 2 since it is not used after initialization. Then the ProOnE process creates its own communication shared memory (e.g., *ShMem 0*) and writes the corresponding address ID into the global shared memory where other processes can read from. After that, it reads the address IDs of other ProOnE processes' communication shared memory (e.g., *ShM- em 1*) to which it can attach. Finally, ProOnE processes listen on predefined ports, waiting for connect requests from remote processes. For network connections that are setup during initialization, we use a client-server model to build all-to-all connections among ProOnE processes.

It is to be noted that each ProOnE process creates only one shared memory region. All the local ProOnE or application processes will attach to this shared memory, but will use different offsets to read or write different segments. An alternative is to create one shared memory for each of the other processes. Both approaches are quite light-weight, so we used the first approach in our design due to its simplicity.

Application processes: Upon launch, each application process performs a ProOnE-specific initialization (by linking to a ProOnE client-side library). During initialization, it attaches to the global shared memory, reads all the address IDs of the communication shared memory regions and attaches to them. For example, in Figure 2, application process 0 on node 0 reads address IDs of *ShMem 0* and *ShMem 1* and attaches to them, respectively. Each application process then sends *connect* requests to remote ProOnE processes. The initialization completes after all the *connect* requests are accepted. Here we choose to establish all the network connections before the main application routine starts, but this can be easily extended to be done on demand as well.

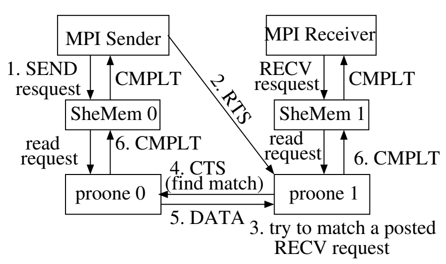
3.2 Onloading MPI Rendezvous

ProOnE is a general purpose onload engine that can be utilized to onload various tasks. In this section, we target MPI as a case study and onload its *Rendezvous* protocol. First we describe the asynchronous progress problem of the *Rendezvous* protocol in most MPI implementations [10]. We then present the details of our design, discuss the critical design issues and describe our solutions.

3.2.1 Communication Progress in MPI Rendezvous

Many MPI designs [17, 10] utilize a *Rendezvous* protocol for communicating large messages. *Rendezvous* is typically a three-step protocol. First, the sender sends a RTS (request to send) message to the receiver. On receiving this, the receiver ensures it has enough buffer to accommodate the incoming data and sends a CTS (clear to send) message to the sender. On receiving the CTS, the sender can send the actual data.

An important but elusive issue in the *Rendezvous* protocol design is its inability to achieve efficient computation and



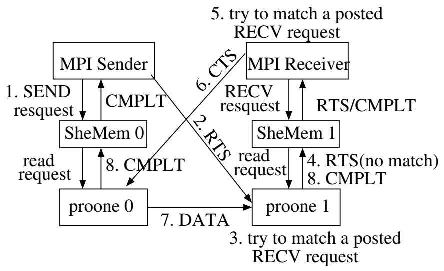


Fig. 4 Communication/handshake protocol: (a) receiver arrives earlier (b) receiver arrives late

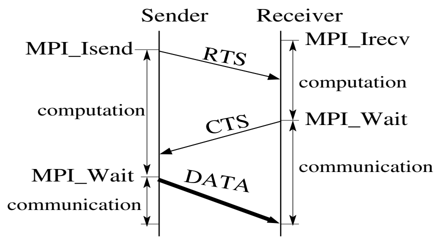


Fig. 3 Effect of process skew on *Rendezvous* protocol

communication overlap [15]. Specifically, the performance of *Rendezvous* heavily depends on the skew between the sender and the receiver. For example, as shown in Figure 3, when the receiver calls an `MPI_Irecv`, if it cannot find any matching RTS, the receive buffer is posted and the application can continue with its computation. Only when the computation completes and the receiver makes another MPI call, it finds the RTS and then sends back CTS. Similarly, when the CTS arrives at the sender, if the sender is still busy with its computation, it has to wait for the computation to complete before sending the actual data. Thus, the delay in detecting control messages eventually results in large out-of-sync communication between the sender and the receiver, which leads to overall loss of performance.

Recent work [28, 19] has addressed this issue mainly by using hardware supported one-sided communication such as RDMA operations. However, the benefits of RDMA-based overlap are limited to specific cases. For example, at least one of the sender or the receiver has to be ready for communication. If both are not (e.g., when the sender calls a non-blocking send, the receiver calls a non-blocking receive and both perform computation), then the communication can be delayed. Similarly, while there are existing network stacks with message matching capabilities (MX Myri-10G [2] and Quadrics QsNet [3]), they perform such matching only for simple cases. For more complicated cases, these stacks cannot perform any asynchronous progress.

On the other hand, the issues are much simpler with ProOnE, since such processing is performed on a general purpose CPU core. ProOnE processes are fully used for handling *Rendezvous* processing in a dedicated manner, so they can detect control messages in a timely manner and consequently better asynchronous progress can be obtained.

3.2.2 Design Overview

As mentioned earlier, the basic idea of onloading *Rendezvous* protocol is to hand over the *Rendezvous* negotiation to ProOnE. All the control messages are sent to ProOnE processes, so we not only need to design the logic in ProOnE, but also have to adjust the communication flow inside MPI. Suppose MPI process 0 intends to send a large message to MPI process 1, and *proone* 0 and *proone* 1 are their associated onload engines, respectively. Figures 4(a) and 4(b) illustrate the interactions in two situations, involving the following data structures:

SEND/RECV requests: These contain information about MPI send/recv calls, such as tag, source/destination rank etc. They are generated by the MPI processes and passed to ProOnE through shared memory.

RTS/CTS messages: These are the MPI control messages that ProOnE processes receive.

CMPLT notifications: These are created by ProOnE processes upon finishing sending or receiving data, in order to inform MPI processes about the completion.

Shared memory segments: The above three structures may be stored in a shared memory region between a pair of ProOnE process and MPI process, so this region is divided into three segments for each purpose.

ProOnE Request queue: ProOnE processes maintain SEND and RECV requests as queues.

Using these structures, ProOnE processes execute three types of work: send or receive messages, write messages or read requests through shared memory, and perform auxiliary functions such as message matching.

3.2.3 Communication/Handshake Protocol

We present the flow of handshake among the MPI sender, MPI receiver and their associated ProOnE processes in Figure 4.

When the sender has a large message to send, it posts a SEND request into shared memory (step 1) where its associated onload engine *proone* 0 will read from. It then sends an RTS to the receiver's associated ProOnE *proone* 1 (step 2). *proone* 1 detects this message and tries to match it with an existing RECV request by searching the request queue and the requests newly posted in shared memory which are not enqueued yet (step 3). There are two resulting cases according to the arrival order of the sender and receiver, i.e., the receiver arrives earlier or late. In the first case (Figure 4(a)), *proone*

1 finds a matched RECV request, so it sends CTS back to *proone* 0 (step 4). In the second case (Figure 4(b)), *proone* 1 does not find a match, so it posts the received RTS into shared memory (step 4). Later, when the receiver performs an MPI call, it will get matched with this RTS (step 5) and send a CTS to *proone* 0 (step 6). Here, either the MPI receiver or its on-load engine *proone* 1 sends a CTS to *proone* 0 so that it will be detected in time. When *proone* 0 receives a CTS, it matches it with the SEND request. With these steps, the *Rendezvous* negotiation has completed. The following data transmission can be handled by either MPI or ProOnE. We choose the latter one due to its potential for slightly larger overlap (step 5 in Figure 4(a) and step 7 in Figure 4(b)). At the end, ProOnE processes post CMPLT in shared memory to notify MPI processes about the send/recv completion.

3.2.4 Design Issues and Solutions

In this section, we discuss several design issues and our solutions in incorporating them into MPICH2.

Message Matching: In MPICH2, a RTS is matched with a RECV request based on a tuple *(src_rank, context_id, tag)*. However, in the context of ProOnE, there will be false matching if we still use this criteria.

According to the MPI specification [13], a receiver is allowed to post a buffer larger than the actual data size. Thus, the receive buffer size alone is not sufficient to decide whether MPI would use a short message eager protocol or the large message *Rendezvous* protocol. To handle this, in our design, any *MPI_Irecv* with a large buffer posts a RECV request to ProOnE if no existing match is found (Figure 4(b)). At the same time, the RECV request is also enqueued into the MPICH2's posted request queue [1] in case it turns out that the message is small and the sender uses the eager protocol. Now both MPICH2 and ProOnE have the same RECV request, which introduces false matching. For example, as shown in Figure 5(a), MPI processes 0 and 1 issue two send and receive requests respectively, all of which have the same matching tuple of *(0,0,0)*. The correct semantic should be that first message is matched with the first RECV at ProOnE side and the second message with second RECV at MPI side. Unfortunately, when MPI process 1 receives the second message with a matching tuple of *(0,0,0)*, it gets matched with the first RECV. Similarly, in Figure 5 (b), the second message (RTS) will be falsely matched with the first RECV request in ProOnE process 1.

One possible solution to this problem is to force the MPI and ProOnE processes to synchronize before matching, but this introduces undesirable high overhead. Our approach is to add one more field, i.e., a sequence number, on each *channel*, to the matching tuple. A *channel* is decided by the original matching tuple. Take Figure 5(a) as an example. The second

[1] Posted request queue contains the arrived receive requests that have not got matched yet.

MPI rank 0	MPI rank 1
send 1(src=0, ctx_id=0, tag=0, len=1M)	recv 1(src=0, ctx_id=0, tag=0, len=1M)
send 2(src=0, ctx_id=0, tag=0, len=1)	recv 2(src=0, ctx_id=0, tag=0, len=1)

(a) send 2 is falsely matched with recv 1 at MPI process with matching tuple of (0,0,0)

MPI rank 0	MPI rank 1
send 1(src=0, ctx_id=0, tag=0, len=1)	recv 1(src=0, ctx_id=0, tag=0, len=1M)
send 2(src=0, ctx_id=0, tag=0, len=1M)	recv 2(src=0, ctx_id=0, tag=0, len=1M)

(b) send 2 is falsely matched with recv 1 at ProOnE process with matching tuple of (0,0,0)

Fig. 5 **F**alse message matching

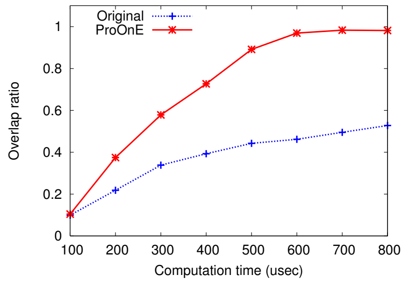
send has the matching tuple *(0,0,0,1)* while the first RECV has the matching tuple *(0,0,0,0)*, so they will not be matched.

Another problem with message matching is that one RTS may be matched with multiple RECV requests. Within a node, one ProOnE process could be associated with multiple MPI processes that may post RECV requests targeting to receive from the same source. These requests can have the same matching tuple. The ProOnE process cannot differentiate among them when performing match. To address this problem, we include the *destination rank* to the matching tuple as well. Therefore, the final matching tuple contains *(src_rank, dst_rank, context_id, tag, sequence_num)*. The sequence number is increased per group of the first four elements. It is to be noted that more complex mechanisms are required for wild-card matching and will be investigated in future work.

Shared Memory Contention: Shared memory is a critical section resource, and we use *semaphores* to avoid contention on it for a pair of MPI process and ProOnE process.

As mentioned in Section 3.2.2, a shared memory region is divided into three segments. A natural question is whether to use separate locks. In our approach, we do not use separate locks. The segment containing SEND or RECV requests and the segment containing RTS messages must be bound together using one lock. As an example, suppose an *MPI_Irecv* arrives and the MPI process searches the RTS message segment for any matched RTS, while at the same time, the associated ProOnE process receives the corresponding RTS and checks the SEND and RECV request segment for a matching RECV request. It is possible that both of them cannot find a match if the segments are locked independently. Thus, they would post a RECV request and an RTS, respectively, assuming that the later on the other side will handle the matching. This pair of RECV request and RTS, then, would be left unmatched forever. A simple solution is to lock the two segments together so that a pair of MPI and ProOnE processes cannot try matching simultaneously.

Memory Mapping: As mentioned in Section 3.2.3, in the current implementation data messages are sent and received by ProOnE. At the sender side, ProOnE reads data from the MPI process' sending buffer and sends it out; and at receiver side, ProOnE writes the received data into MPI process' receiving buffer. Generally there are two approaches allowing a ProOnE process to access the buffer in MPI processes. One is



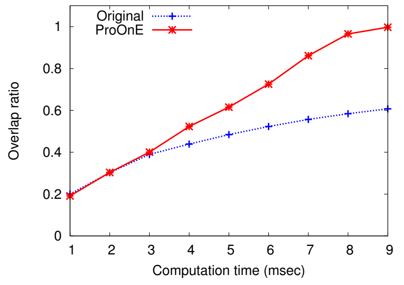
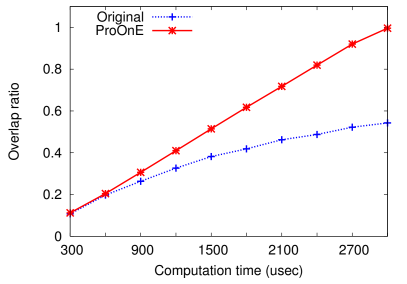


Fig. 6 Sender overlap performance: (a) 256 KB message (b) 1 MB message



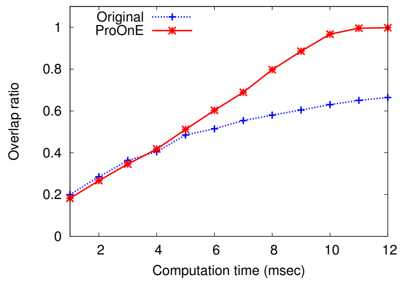


Fig. 7 Receiver overlap performance: (a) 256 KB message (b) 1 MB message

the user-level shared memory-based approach [16]. The other is the kernel direct-copy method [18]. We use the second approach due to its lower overhead. MPI processes perform the kernel-level mapping and wrap the mapping information in the SEND/RECV requests. While the kernel context-switch introduces some overhead, it is negligible for large messages.

4 Experimental Results

In this section, we present a comprehensive analysis of the experimental results. First we evaluate the ProOnE based *Rendezvous* protocol in terms of computation/communication overlap, program progress and the process skew impact. Then, we evaluate ProOnE in the context of two popular scientific application kernels: matrix multiplication and 2D Jacobi sweep. For all the experiments, we use the basic MPICH2 without ProOnE as the baseline for comparison. We use the legend of "Original" referring to the basic MPICH2, and the legend of "ProOnE" for our design with ProOnE.

Experimental Testbed: Our test bed consists of 64 nodes with dual quad-core (8 cores in total) Xeon processors and 4 GB memory, running RHEL4 U4 with the kernel 2.6.9.34. The nodes are connected by InfiniBand DDR cards, but we use IPoIB (TCP/IP over IB) as the network transport as our approach is independent of the transport and does not assume any specific features from the network stack. For all the experiments with ProOnE, we use one core (by setting CPU affinity) on each node to run ProOnE daemon.

4.1 Computation and Communication Overlap

In this section we evaluate the capability of the ProOnE-based *Rendezvous* protocol design to overlap computation with communication. The benchmark is similar to that suggested in [28]; overlap is defined as the ratio of the computation time over the total time (computation and communication).

Sender-side overlap: Figure 6(a) presents the sender-side overlap performance using a message size of 256 KB. With ProOnE, as computation time increases, the overlap ratio becomes larger as longer computation time provides better opportunity to overlap communication. In fact, when the computation time is larger than 600 μs, ProOnE can provide almost full overlap, while vanilla MPICH2 can only provide an overlap of less than 0.6. This is because when the MPI process is busy with computation, its associated ProOnE daemon can proceed with communication simultaneously; however, in MPICH2, the sender process cannot perform any communication during the whole computation period. In Figure 6(b), we show a similar measurement by using 1MB messages; we observe a similar trend, except that the full overlap is achieved at a higher computation time. This is because the higher communication time for transferring 1MB messages requires longer computation for efficient overlap.

Receiver-side overlap: The receiver overlap performance is shown in Figures 7(a) and 7(b). We see a similar trend as the sender side overlap, except that a higher computation time is needed to achieve good overlap. We attribute it to the overhead of kernel-level memory mapping and copy. As mentioned in Section 3.2.4, we use ProOnE to send and receive data in the current implementation. It involves the overhead of a kernel memory copy at both sender and receiver. While the sender's performance is affected only by the overhead at sender side, receiver's performance suffers from the overhead at both sides, resulting in longer communication time and worse overlap.

4.2 Impact of Process Skew

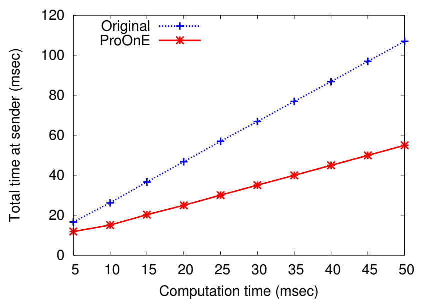


Fig. 8 Progress performance with process skew

In this section we evaluate the impact of process skew on *Rendezvous* asynchronous progress. Our benchmark consists of a pair of MPI processes performing bidirectional communication. Each process initiates a *MPI_Irecv* and a blocking *MPI_Send*. After that, they compute for some time *W*, and finally call *MPI_Wait*. In this way, the skew is introduced, as for a pair of send and receive, the receive call arrives earlier than the send call. This whole process is repeated for 1000 iterations. We average the total time on either side with varying *W* and show the results using message size of 1M in Figure 8.

In MPICH2, receiving calls at both sides arrive earlier and thus cannot make progress until the computation completes and Wait is called. Thus the computation on the sender and the receiver is serialized and the total time includes both. On the other hand, ProOnE can start the data transfer as soon as one pair of send and receive arrives, so the total time is not affected by skew and includes only its own computation time.

4.3 Application Availability

To measure application availability and overhead, we used the Sandia Benchmark [26] (SMB). Figures 9(a) and 9(b) show the sender's availability percentage and overhead with increasing message sizes. For small messages, both ProOnE and MPICH2 exhibit high availability (around 91%) and low overhead. However, performance deteriorates rapidly in MPICH2 when the message size is larger than 1KB. On the other hand, because of the independent asynchronous progress capability, ProOnE provides almost full availability and very low overhead for medium and large messages ($\geq$ 8KB). Note that here we use 4KB as the *Rendezvous* protocol threshold, so there is degradation for messages of 2KB and 4KB where ProOnE is not utilized. At the receiver side, we noticed a similar performance trend, but the results are not included here because of the space constraints.

4.4 Application Performance

In this section we evaluate our design with two popular application kernels widely used in many scientific and engineering applications.

Matrix Multiplication: The kernel uses Cannon's algorithm and employs MPI for inter-node communication and OpenMP [11] for intra-node communication. We use four nodes with one MPI process on each and measure the completion time with varying problem sizes (matrix sizes), as shown in Figure 11. As shown in the figure, ProOnE performs much better especially for larger problem sizes. We further measured this benefit with different cost ratios. Figure 12 illustrates the normalized execution time for a square matrix of 1024 elements on different system configurations. NxM means N nodes with M cores on each node used, so larger M indicates relatively lower percentage of resources in the system used for ProOnE. We see that ProOnE provides consistent benefits with decreasing percentage of additional cost. This presents the promising future for applying ProOnE in applications with inexpensive multi-/many-core processors.

The second application we evaluated is a 2D Jacobi sweep [9]. In this application, on each iteration every process initiates the exchange of boundary data with all the neighbors, and then performs computation on the internal data. Upon receiving all the boundary data, it also performs computation on the received data. We use nonblocking send/recv calls to initiate the boundary data exchange and use *MPI_Waitall* after the internal data computation to complete these operations. Figure 12 illustrates the time for *MPI_Waitall* on a 4x4 process grid (4 nodes with 4 MPI processes on each). We observe that the design with ProOnE has much smaller waiting time than the design without ProOnE, offering increasing benefits with larger boundary data. This shows that ProOnE can effectively drive the communication progress and thus provide significant improvement on the overall application performance.

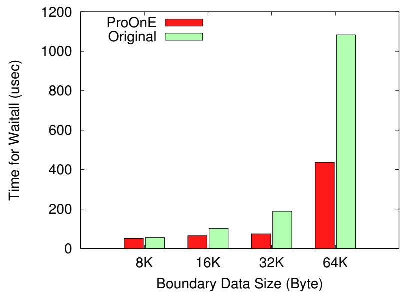
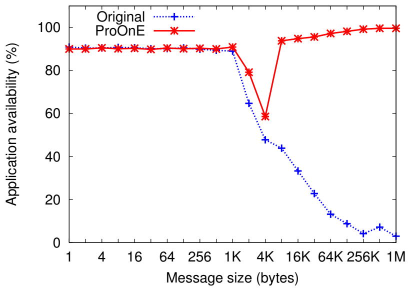


Fig. 12 Jacobi sweep with varying boundary data size

5 Conclusions and Future Work

As high-end computing systems rapidly scale to very large clusters, traditional protocol offload engines have to offload more and more complex tasks to meet the performance and scalability requirements. However, the hardware offloading is quite expensive and inflexible due to its manufacturing complexity. On the other hand, the increasing deployment of multi- and many-core processors offers new opportunities to "onload" these tasks. In this paper we presented the design of a



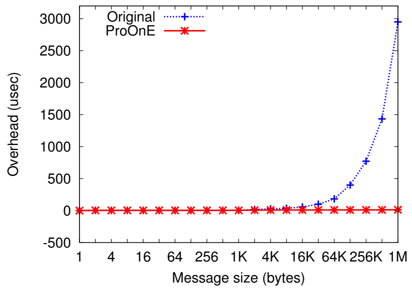


Fig. 9 Availability performance at sender: (a) availability percentage (b) overhead

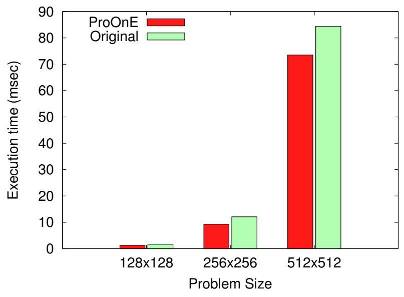


Fig. 11 Matrix multiplication with varying problem size

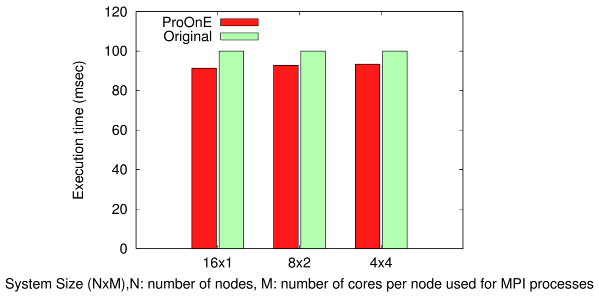


Fig. 12 Matrix multiplication with varying system configuration

general purpose Protocol Onload Engine (ProOnE) that utilizes a small subset of the available cores on a multi-core equipped node to "onload" complex tasks. Additionally, we utilized ProOnE to onload the *Rendezvous* protocol in MPI and incorporated this into MPICH2. From our evaluations, we find that the ProOnE-based MPI design provides almost full communication/computation overlap, close to 100% application availability, good resilience to process skew, and significant application level benefits in matrix multiplication and Jacobi method applications.

For future work, we plan to study the performance and scalability of applying ProOnE in large scale systems and investigate its benefits in other scientific and engineering applications. We also intend to use ProOnE for other protocols such as enterprise data-center tasks and file system tasks.

References

1. http://en.wikipedia.org/wiki/Virtual_Interface_Architecture.
2. http://www.myri.com/myrinet/overview/.
3. www.quadrics.com/.
4. http://www.intel.com/products/processor/core2XE/.
5. http://www.sun.com/processors/niagara/.
6. Chelsio TOE. http://www.chelsio.com/.
7. Giganet clan. http://www.emulex.com/.
8. InfiniBand Trade Association. http://www.infinibandta.com.
9. Jacobi Method. http://en.wikipedia.org/wiki/Jacobi_method.
10. MPICH2. http://www.mcs.anl.gov/research/projects/mpich2/.
11. OpenMP. http://openmp.org/wp/.
12. Top 500 SuperComputer Sites. http://www.top500.org/.
13. MPI: A Message Passing Interface. In *MPI Forum*, 1993.
14. G. Amerson and A. Apon. Implementation and design analysis of a network messaging module using virtual interface architecture. In *In International Conference on Cluster Computing*, 2004.
15. R. Brightwell and K. D. Underwood. An Analysis of the Impact of MPI Overlap and Independent Progress. In *Proceedings of the 18th annual international conference on Supercomputing*, March 2004.
16. L. Chai, A. Hartono, and D. K. Panda. Designing High Performance and Scalable MPI Intra-node Communication Support for Clusters. In *The IEEE International Conference on Cluster Computing*, 2006.
17. W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-Performance, Portable Implementation of the MPI. Technical report, Argonne National Laboratory and Mississippi State University.
18. 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 Conference on Cluster Computing (poster presentation)*, 2007.
19. R. Kumar, A. R. Mamidala, M. J. Koop, G. Santhanaraman, and D. K. Panda. Lock-free Asynchronous Rendezvous Design for MPI Point-to-point communication. In *EuroPVM '08*, 2008.
20. S. Majumder, S. Rixner, and V. S. Pai. An event-driven architecture for mpi libraries. In *Computer Science Institute Symposium*, 2004.
21. A. Ortiz, J. Ortega, A. F. Daz, and A. Prieto. Comparison of onloading and offloading strategies to improve network interfaces. In *PDP*. IEEE Computer Society, 2008.
22. G. Regnier, D. Minturn, G. McAlpine, V. Saletore, and A. Foong. ETA: Experience with an Intel Xeon Processor as a Packet Processing Engin . In *Proceedings of the 11th Symposium on High Performance Interconnects (HOTI'03)*, 2003.
23. G. J. Regnier, S. Makineni, R. Illikkal, R. Illikkal, D. Minturn, R. Huggahalli, D. Newell, L. Cline, and A. Foong. Tcp onloading for data center servers. *IEEE Computer*.
24. J. C. Sancho, K. J. Barker, D. J. Kerbyson, and K. Davis. Quantifying the Potential Benefit of Overlapping Communication and Computation in Large-Scale Scientific Applications. 2006.
25. J. C. Sancho, D. J. Kerbyson, and K. J. Barker. Efficient offloading of collective communications in large-scale systems. In *IEEE International Conference on Cluster Computing*, 2007.
26. Sandia National Laboratories. Sandia MPI Micro-Benchmark Suite. http://www.cs.sandia.gov/smb/.

27. P. Shivam and J. S. Chase. On the elusive benefits of protocol offload. In *SIGCOMM'03 Workshop on NICELI*, 2003.
28. S. Sur, H.-W. Jin, L. Chai, and D. K. Panda. RDMA Read Based Rendezvous Protocol for MPI over InfiniBand: Design Alternatives and Benefits. In *Symposium on PPOPP* , March 2006.
29. F. Trahay, E. Brunet, A. Denis, and R. Namyst. A multithreaded communication engine for multicore architectures. In *International Parallel and Distributed Processing (IPDPS)*, 2008.
30. K. Vaidyanathan, P.Lai, S. Narravula, and D. K. Panda. Optimized distributed data sharing substrate in multi-core commodity clusters: A comprehensive study with applications. In *Int'l Symposium on Cluster Computing and the Grid (CCGrid)*, May 2008.