Improving concurrency and asynchrony in multithreaded MPI applications using software offloading

Karthikeyan Vaidyanathan
Parallel Computing Lab, Intel
karthikeyan.vaidyanathan@intel.com

Dhiraj D. Kalamkar
Parallel Computing Lab, Intel
dhiraj.d.kalamkar@intel.com

Kiran Pamnany
Parallel Computing Lab, Intel
kiran.pamnany@intel.com

Jeff R. Hammond
Parallel Computing Lab, Intel
jeff.r.hammond@intel.com

Pavan Balaji
Math and Computer Science Division,
Argonne National Laboratory
balaji@anl.gov

Dipankar Das
Parallel Computing Lab, Intel
dipankar.das@intel.com

Jongsoo Park
Parallel Computing Lab, Intel
jongsoo.park@intel.com

Bálint Joó
Thomas Jefferson National Accelerator Facility
bjoo@jlab.org

Abstract

We present a new approach for multithreaded communication and asynchronous progress in MPI applications, wherein we offload communication processing to a dedicated thread. The central premise is that given the rapidly increasing core counts on modern systems, the improvements in MPI performance arising from dedicating a thread to drive communication outweigh the small loss of resources for application computation, particularly when overlap of communication and computation can be exploited. Our approach allows application threads to make MPI calls concurrently, enqueuing these as communication tasks to be processed by a dedicated communication thread. This not only guarantees progress for such communication operations, but also reduces load imbalance. Our implementation additionally significantly reduces the overhead of mutual exclusion seen in existing implementations for applications using `MPI_THREAD_MULTIPLE`. Our technique requires no modification to the application, and we demonstrate significant performance improvement (up to 2X) for QCD, 1-D FFT and deep learning CNN applications.

Categories and Subject Descriptors C.5.1 [*Computer System Implementation*]: Large and Medium Computers—Supercomputers; D.2.2 [*Software Engineering*]: Design Tools and Techniques—Software libraries; D.4.4 [*Operating Systems*]: Communications Management—Network communication

General Terms Performance

SC '15, November 15 - 20, 2015, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2807591.2807602

1. Introduction

MPI+X continues to be the de facto standard programming model for HPC applications. Scaling MPI+X applications to current large supercomputers and beyond presents a number of challenges, among which are: (1) overlap of communication and computation, which includes asynchronous communication; (2) low overhead for MPI calls, (3) high message-rate, even for lightweight cores; and (4) low system noise, which precludes thread migration and compels minimization of hardware interrupts. An additional requirement in the case of applications that make concurrent calls to MPI from multiple threads (i.e. `MPI_THREAD_MULTIPLE`) is the need to avoid high overheads associated with mutual exclusion, which are common in many MPI implementations today [6]. The need for these features in next-generation supercomputers is illustrated in recent procurements, such as CORAL [13] and Trinity-NERSC8 [4]. All application benchmarks therein use MPI, and most use OpenMP®; and while many applications today do not require `MPI_THREAD_MULTIPLE`, it is a requirement in CORAL in order to allow for programming model flexibility in the future.

To meet the challenges of scaling MPI+X applications, we focus on the key underlying issue: servicing an MPI operation requires a certain amount of computation within the MPI layer. For instance, small message transfers commonly use an "eager" protocol, where the MPI implementation allocates internal buffers, performs memory copies, initiates data transfer, and performs appropriate matching (tags, communicator, source) to place incoming data in the right memory buffer. In most MPI implementations, some or all of these operations are handled in software. Large messages are often transferred using a rendezvous protocol, which on most systems requires pinning the application buffer and sending a control message to determine the buffer location for data transfer. Again, this requires software intervention. Collectives such as reductions also require compute resources, to configure a hierarchy for data exchanges or to prepare buffers for scatter/gather operations. For nonblocking MPI calls, this gets even trickier as compute resources are often required to ensure progress of operations. For instance, a delay in

processing control messages during the rendezvous protocol can result in exposing communication latencies to the application.

However, the typical MPI+X application uses all available threads for its own compute needs. Communication is typically funneled to a master thread that issues all MPI calls. In this model, all MPI compute needs are met by the master thread, which must therefore do more work than the other threads. The consequent load imbalance hurts the application's scalability.

It is extremely difficult to determine at what points during application computation must MPI be invoked to ensure asynchronous progress; if this is not done correctly, compute-communication overlap cannot be achieved and performance will suffer. Moreover, the thread used for this purpose will again cause imbalanced load.

When an application uses multiple threads for communication, it can encounter high overheads associated with mutual exclusion that are common in many MPI implementations.

We address all these challenges by dedicating a processor thread in each MPI rank to which all MPI communication operations are offloaded. The remaining threads, used by the application, may issue MPI calls in any manner—serialized, funneled, or concurrently. These are routed to the MPI offload thread via a lock-free command queue. This approach significantly reduces MPI call latency and consequently allows better balanced load among the application threads. The challenge of ensuring asynchronous progress is met since the MPI offload thread tracks the progress of all MPI operations constantly. The loss of a compute resource for the application is outweighed by the improvements in communication performance, and is in any case a small cost given the increases in core counts seen in modern systems.

In this paper, we detail our approach and demonstrate that its use in MPI+X applications results in excellent computation-communication overlap; simplifies application implementation, thereby reducing programmer effort significantly; and delivers high performance with minimal overhead even for concurrently issued MPI calls.

The remainder of the paper is organized as follows. In Section 2 we describe some current techniques used to manage these challenges. In Section 3 we detail our method of using a dedicated thread for offloading MPI communication. In Section 4 we use microbenchmarks to show the benefits of our approach in a number of areas, including operation latency and compute-communication overlap; and in Section 5 we show these benefits on three well-known HPC applications. In Section 6 we consider related work. In Section 7 we summarize our conclusions and briefly consider future work.

2. Background

In this section, we illustrate the challenge of overlapping communication with computation with a typical stencil computation code. We describe two approaches used today to address this challenge.

Consider the code in Listing 1. The first step performed is MPI initialization, using `MPI_Init_thread()`. An OpenMP parallel region follows, in which all threads prepare the boundary buffers for communication (shown as boundary pack). A thread barrier is used to ensure that all buffers are ready for communication, after which the master thread initiates the boundary exchange by posting nonblocking MPI calls (line 6). The remaining threads begin internal volume processing (lines 7 through 17); the master thread joins in this work once it has completed posting the nonblocking MPI calls. The master thread then invokes `MPI_Waitall()` to complete the boundary exchange while the remaining threads wait at a barrier. Finally, boundary processing is performed by all the threads.

The performance of this stencil code depends on several factors. Since the master thread alone issues all MPI calls, there is an inherent load imbalance between it and the remaining threads which manifests at the thread barrier in line 19. Further, when there is sufficient internal volume processing to overlap the boundary exchange communication, we would expect the master thread to spend a minimal amount of time waiting for the boundary exchange to complete at line 18. However, the MPI standard does not force progress during nonblocking calls. For example, if an application posts an `MPI_Irecv()` followed by an `MPI_Isend()` and then performs computation before posting the corresponding `MPI_Wait()` calls, the communication often will not be overlapped with the computation. This is because the implementation of the send call requires the corresponding receive to be posted on the target before data transfer is initiated. Thus, data transfer actually occurs during the `MPI_Wait()` call even when there is enough computation to overlap, thus resulting in load imbalance and performance degradation.

We refer to this as the *baseline* approach.

```
Baseline  : THREAD_LEVEL=MPI_THREAD_FUNNELED

Iprobe    : THREAD_LEVEL=MPI_THREAD_FUNNELED
            PROGRESS=_Pragma(``omp master'')
            { MPI_Iprobe(...); }

Comm—self : THREAD_LEVEL=MPI_THREAD_MULTIPLE

/* sample stencil code */
{
1:  MPI_Init_thread(THREAD_LEVEL)
    /* Progress thread starts (comm—self or
        offload) */
    ...

2:  #pragma omp parallel
3:  {
4:    { /* boundary pack */ }
5:  #pragma omp barrier
6:  #pragma omp master { MPI_Irecv(...);
                        MPI_Isend(...); }
      /* internal volume processing */
7:    {
8:      for x loop do;
9:        PROGRESS
10:       for y loop do;
11:         PROGRESS
12:         for z loop do;
13:           ...
14:         done
15:       done
16:     done
17:   }
18: #pragma omp master { MPI_Waitall(...); }
19: #pragma omp barrier
20:   { /* boundary processing */ }
21: }
}
```

Listing 1: MPI overlap challenges

2.1 Using Iprobes

The problem of lack of progress in the *baseline* approach can be addressed by periodically invoking MPI during the internal volume computation with an `MPI_Iprobe()` call to facilitate progress. This is illustrated in Listing 1 by the `PROGRESS` phase (lines 9 and 11). We refer to this as the *iprobe* approach. While this approach provides some overlap of communication with computation, the master thread lags even further behind other threads because of the time spent inside the `MPI_Iprobe()` call, which causes additional load imbalance that can outweigh the benefits of the achieved overlap. Moreover, it is extremely difficult to determine where and

how frequently to insert these calls for best overlap. For example, a PROGRESS on line 9 might be more effective in achieving overlap than on line 11, or vice versa.

2.2 The SELF communicator thread

Another approach dedicates a thread for this purpose, duplicating the MPI_COMM_SELF communicator, and posting a blocking receive on this duplicated communicator for which a corresponding send is never posted. This leaves the thread always inside MPI which uses it to drive the progress engine. Clearly, this reduces by one the number of threads available to the application. However, it also requires use of MPI_THREAD_MULTIPLE at MPI initialization, to allow the master thread to make MPI calls. Thus, performance is largely dependent on the underlying multithreaded MPI implementation (which is typically poor). Furthermore, this approach does not address the load imbalance issues caused by the master thread spending time inside MPI; on the contrary, because of contention with the SELF communicator thread, the master thread typically sees more time spent in MPI calls, resulting in greater load imbalance. We refer to this as the *comm-self* approach.

Note that the term dedicated software thread or communication thread or MPI thread refers to a physical core used explicitly for handling communication related activities.

3. MPI Offload Infrastructure

In this section, we detail the design of our MPI offload infrastructure. The core idea is to decouple application computation from MPI communication by providing an infrastructure that imposes minimal overhead on the interaction between application threads and MPI. We achieve this by dedicating a thread to drive MPI and by offloading all application MPI calls to this thread via a lightweight lock-free command queue.

Thus, the offload infrastructure consists of two parts: (1) an MPI offload thread that services a command queue and invokes MPI to process received commands and (2) a library that translates application MPI calls into commands that are placed in the command queue.

3.1 Decoupling computation from MPI communication

Figure 1 illustrates the overall design of our MPI offload infrastructure. For applications using MPI_THREAD_FUNNELED or MPI_THREAD_SERIALIZED, the master thread communicates with the offload thread using a circular command queue, as shown in the figure (we describe support for MPI_THREAD_MULTIPLE later in this section). When the application's master thread makes an MPI call, our library serializes the call parameters into a call-specific structure and inserts this information into the command queue. The offload thread polls this queue and on finding a command, extracts the call parameters from the structure and issues the MPI call. Note that the application thread never enters MPI itself; only the offload thread does so.

For a blocking MPI call, the application master thread spins on a *done* flag that will be set by the offload thread when it finishes processing the MPI call. For a nonblocking MPI call however, in order to minimize overhead, it is desirable for the application thread to return immediately after the command is inserted into the queue. However, as the offload thread has not yet invoked MPI at this point, a valid MPI_Request is not yet available to return to the application. We address this by allocating an array of MPI_Request objects within the offload infrastructure; we assign a free object from this pool to each nonblocking call and return its index to the application as the MPI_Request. We maintain this pool as an array-based singly linked list in order to minimize allocation and free time.

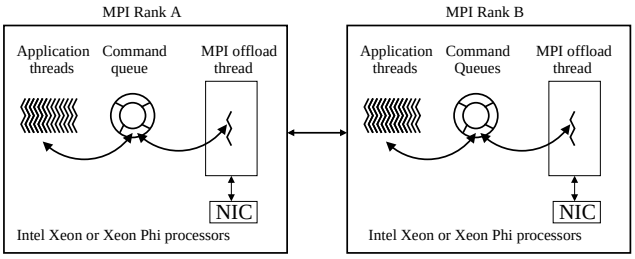

Figure 1: MPI offload using a dedicated thread

Note that no additional memory copies are required with our approach since the application threads and the MPI offload thread run in the same address space.

We refer to this as the *offload* approach.

3.2 Improved asynchronous progress

As discussed in Section 2, the MPI standard does not force progress during nonblocking MPI calls. For this reason, the offload thread keeps track of all in-flight nonblocking MPI calls and ensures progress on them by issuing MPI_Testany() calls whenever its command queue is empty. When one of these nonblocking MPI calls completes, the offload thread sets its *done* flag. This approach not only ensures asynchronous progress but also optimizes application calls to MPI_Wait(), MPI_Waitany(), and MPI_Waitall(), since they only need to check the appropriate *done* flag.

3.3 Support for MPI_THREAD_MULTIPLE

Our MPI offload infrastructure supports applications that use MPI_THREAD_MULTIPLE, i.e., use multiple threads to make MPI calls simultaneously. This is not a widely used model because of the performance loss caused by mutual exclusion and serial bottlenecks present in typical MPI implementations that support reentrancy, in accordance with the standard, but generally offer limited true parallelism. For most MPI implementations, using MPI_THREAD_MULTIPLE can significantly increase MPI call latency, particularly when there is contention, but even without. Yet, support for MPI_THREAD_MULTIPLE is increasingly important, as evidenced by CORAL [13].

Our approach allows for highly efficient MPI_THREAD_MULTIPLE support as follows. We use atomic operations to convert both the offload thread's command queue and the pool of MPI_Request objects into lock-free data structures, thereby allowing scalable concurrent MPI calls. Further, in order to prevent the blocking MPI call of one application thread from delaying the progress of the calls of other threads, we convert all blocking calls into their nonblocking equivalents and use MPI_Test() calls to check for their completion. This approach ensures that the offload thread is not impeded from processing MPI commands issued by other application threads.

Our results in Section 4 demonstrate that our approach outperforms standard MPI_THREAD_MULTIPLE implementations despite using a single thread to actually interact with the MPI layer.

We note that one of the advantages of this approach is that since all communication is funneled through a single communication thread (even in the case of MPI_THREAD_MULTIPLE), no locking is required within the MPI implementation (i.e. it would be equivalent to MPI_THREAD_FUNNELED or MPI_THREAD_SERIALIZED). This feature, however, has the shortcoming that a single thread is driving communication, which might not always be sufficient to take complete advantage of the network, especially for small messages.

Another shortcoming of this approach is that not all blocking calls have nonblocking equivalents (e.g. `MPI_WIN_FENCE`). We acknowledge this shortcoming but note that (1) it exists only for applications using `MPI_THREAD_MULTIPLE` (a very small fraction of existing applications) and (2) most MPI calls have nonblocking equivalents and our approach can support all applications that do not rely on the small set of functions that do not have a nonblocking equivalent.

3.4 Support for unmodified applications

We enable applications to use our MPI offload infrastructure without modification by using the `LD_PRELOAD` feature to dynamically interpose our library between the application and MPI. Our implementation intercepts all MPI calls—spawning the required thread at `MPI_Init()` and creating the necessary command queues—and translates MPI calls into commands that are submitted to the command queue for processing by the offload thread. Thus, our approach does not require any code changes to the application.

4. Microbenchmarks

In this section, we use a number of microbenchmarks to evaluate the performance and overheads of our MPI offload infrastructure along a number of dimensions. We compare with a *baseline* that uses MPI in the typical funneled manner as well as with two existing approaches that attempt to address the asynchronous progress problem—*iprobe*, and *comm-self* (described in Section 2).

For all our experiments, we use the following two clusters:

Endeavor: This cluster consists of dual socket 14 core Intel® Xeon® E5-2697 v3 processor nodes with 64 GB host memory. Each node has a 61 core Intel® Xeon Phi™ coprocessor with 8 GB memory.[1] The nodes are connected with InfiniBand FDR network adapters. We use Intel MPI 5.0.2.044.

NERSC Edison: This Cray® XC30 cluster consists of dual socket 12 core Intel Xeon E5-2695 v2 processor nodes with 64 GB host memory and 256 KB L2 cache. The nodes are connected with Cray Aries using a DragonFly network topology. We use Cray MPI 2.03 and ICC 14.0.0.

Our experiments with these microbenchmarks focus on: (1) evaluating the effectiveness of asynchronous progress by measuring compute-communication overlap, (2) quantifying the overhead of issuing nonblocking MPI calls such as `MPI_Isend()` and `MPI_Irecv()`, also including nonblocking collectives, and (3) latency and bandwidth measurements using the standard OSU single and multithreaded microbenchmarks [5]. For all our experiments, we use one MPI rank per socket.

4.1 Compute-communication overlap

We have developed an overlap benchmark for point-to-point MPI calls across two processes that operates as follows. Each process issues a nonblocking `MPI_Irecv()` followed by a nonblocking `MPI_Isend()` to receive and send a message to the other process. We measure the time taken for these calls, which we call the post time. Next, each process issues two `MPI_Wait()` calls to wait for the nonblocking receive and send calls made earlier to complete. The time these calls take to complete is the wait time. The total time taken to complete all four calls is the communication time.

The overlap benchmark then repeats these calls, introducing some computation between the nonblocking `MPI_Isend()` call and the first `MPI_Wait()` call. The computation time is equal to the communication time measured in the previous step.

We define the overlap time as the difference between the wait times for the two steps, and report the post time, the wait time of the second step, and the overlap time as a percentage of the communication time. For 100% overlap of compute with communication, we expect the overlap time to match very closely with the communication time.

For measuring overlap on collectives, we use the standard IMB-NBC suite distributed as part of Intel MPI; it uses a similar methodology.

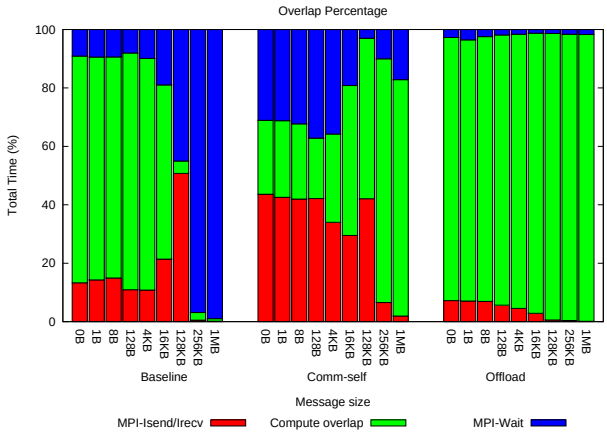

Figure 2: Compute-communication overlap for nonblocking point-to-point calls

Figure 2 shows post time, overlap time, and wait time as a percentage of communication time. For the *baseline* approach, we see reasonable overlap between 70% and 80% for small messages up to 4 KB. However, as the message size increases, the overlap drops drastically to 1% for large messages (2 MB). This is expected behavior, as the MPI implementation uses the eager protocol for messages up to 128 KB, for which an internal memory copy is made during `MPI_Isend()`. Thus we see reasonable overlap for very small messages, but the time spent in `MPI_Isend()` increases as the message size approaches 128 KB—reducing the overlap potential. For messages larger than 128 KB, the MPI implementation switches to the rendezvous protocol for which only a control message handshake is performed at `MPI_Isend()`—data transfer is deferred. However, in the microbenchmark's computation phase, MPI is unable to process these control messages and therefore the entire communication is handled only at `MPI_Wait()` which results in poor overlap. Note that the problem of non-blocking send progress in the absence of a posted receive actually arises from MPI's rendezvous protocol that must be used for messages above a certain size – with no receive posted, the sender cannot know where the message should be written.

With the *comm-self* approach, multiple threads enter the MPI region simultaneously. Thus, issuing the `MPI_Isend()` call itself takes longer which results in reduced overlap of 20% to 30% for small messages of up to 4 KB. However, for large messages we see that the *comm-self* approach allows up to 80% overlap since the control messages of the rendezvous protocol are handled in a timely manner by the *comm-self* thread.

With our *offload* approach, we see that overlap is consistently above 85% for small messages and reaches up to 99% for large messages. Since all MPI calls are offloaded, the time spent in the calls themselves is significantly less, which results in maximum overlap potential.

We see similar trends for overlap of nonblocking MPI collectives with both small and medium messages in Figures 3(a) and 3(b).

[1] Intel, Xeon, and Intel Xeon Phi are trademarks of Intel Corporation in the U.S. and/or other countries.

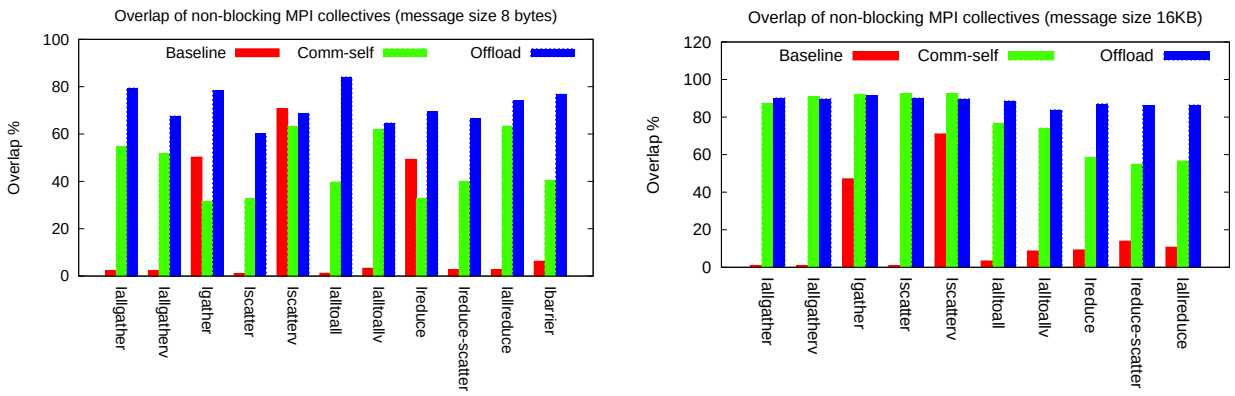

Figure 3: Compute-communication overlap for nonblocking MPI collectives: (a) 8 bytes and (b) 16 KB

4.2 Nonblocking MPI call overhead

We have modified the OSU latency microbenchmark for point-to-point and collectives by replacing blocking calls with nonblocking calls followed by `MPI_Wait()`s. We measure the time taken for issuing these nonblocking calls separately.

Figure 4 shows the time spent in issuing a nonblocking MPI call using the *baseline*, *comm-self* and *offload* approaches. We see in Figure 4 that the *baseline* approach takes significantly longer for `MPI_Isend()`s for up to 128 KB messages, and much less time for larger messages. As explained in the previous sub-section, this arises from the MPI implementation's switch from the eager protocol to the rendezvous protocol.

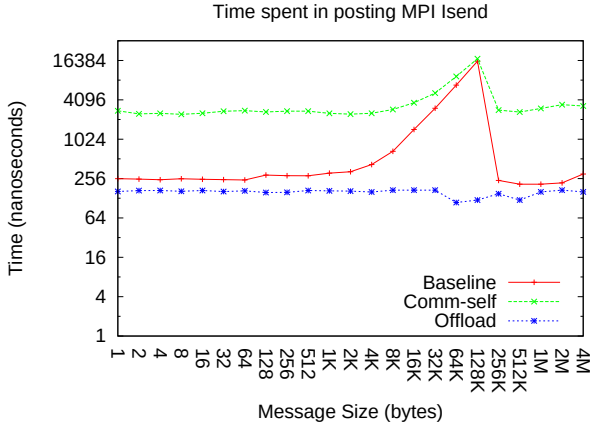

Figure 4: Nonblocking MPI_Isend call as part of OSU ping pong test on 2 Endeavor Xeon nodes

The *comm-self* approach shows similar behavior, except that the latencies are higher by 2.5 microseconds—this arises from the required use of `MPI_THREAD_MULTIPLE` which introduces overhead.

Our *offload* approach shows constant latency for `MPI_Isend()` of about 140 nanoseconds, irrespective of the message size. This is due to the fact that all MPI calls are simply offloaded to a dedicated thread via a command queue. As a consequence, an application thread making an MPI call will return quickly to join other application threads in computation which results in significantly reduced load imbalance.

Figures 5(a) and 5(b) show similar trends for nonblocking MPI collectives on 16 Endeavor Xeon nodes and further justify the need to decouple application computation and MPI communication.

4.3 Overlap and load imbalance benefits in applications

While we have shown significant overlap and reduced load imbalance as benefits of our offload infrastructure, it is important to understand how these benefits translate into application performance. Here, we report the improvements in overlap and the reduction in load imbalance seen in QCD and in 1-D FFT—overall application performance is reported in Section 5.

Table 1 compares the *baseline* approach with the *offload* approach in QCD Dslash on a $32^3 \times 256$ lattice. This data was gathered on the Endeavor Xeon cluster and we report the time as seen by thread 0 of MPI rank 0. We measure the post time as the time taken for posting all nonblocking `MPI_Irecv()` and `MPI_Isend()` calls. We also include the time taken for boundary processing such as pack and unpack operations and barrier time among the threads and report it as misc time.

First, observe that the *offload* approach shows up to 5% slowdown in internal compute as compared to the *baseline* approach. This is a consequence of the use of a dedicated thread for communication offloading—the application has one less thread available to it which causes an increase in internal compute time.

However, we see 99% overlap up to 128 nodes, and 33% overlap for 256 nodes with the *offload* approach. We also see >99% reduction in post time. Observe that at 256 nodes, the post time when using the *baseline* approach occupies a significant fraction of the total time. Reducing this to a few nanoseconds with the *offload* approach results in a significant reduction of overall runtime. The reason for the large post time (50 microseconds) seen in the *baseline* approach is that the communication message sizes across all directions in the lattice goes down to 48 KB, crossing below the rendezvous threshold. For this message size, as shown in Figure 4, significant time is spent in posting the nonblocking calls when using the *baseline* approach. At 128 nodes, only two out of the six directions in the lattice go below the rendezvous threshold and hence we do not see a significant increase in post time (13 microseconds) as compared to 256 nodes.

We see similar trends for 1-D FFT in Table 2—up to 96% reduction in post time, 87% overlap, and a speedup of up to 31% using the *offload* approach as compared to the *baseline* approach. Due to the presence of multiple segments in FFT, the number of nonblocking calls in each iteration increases. Thus, the post time occupies a significant portion of overall time.

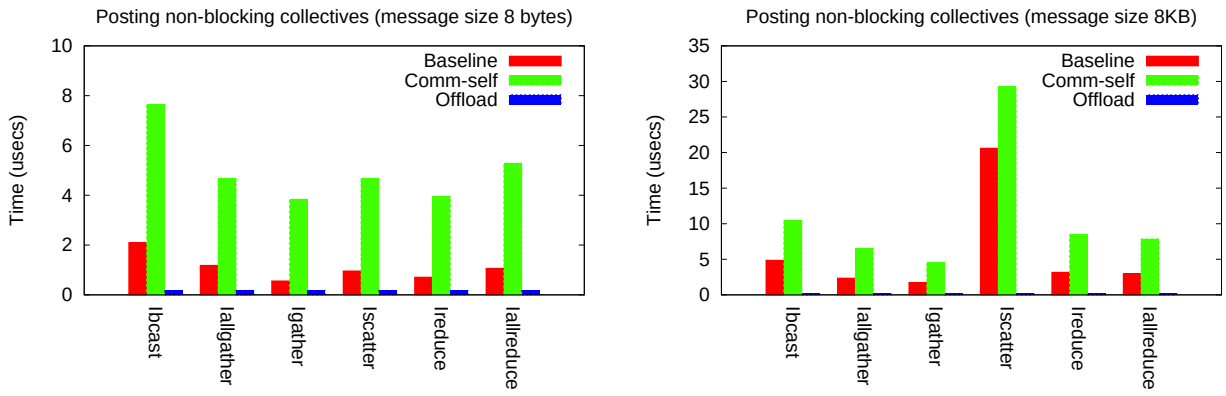

Figure 5: Nonblocking collectives MPI call latency: (a) MPI_Icollectives 8 byte on 16 Endeavor Xeon nodes and (b) MPI_Icollectives 8 KB on 16 Endeavor Xeon nodes

Nodes	Baseline (Time in microseconds)					Offload (Time in microseconds)					Internal Compute Slowdown	Post Time Reduction	Wait Time Reduction
	Internal Compute	Post Time	Wait Time	Misc Time	Total Time	Internal Compute	Post Time	Wait Time	Misc Time	Total Time			
8	3,448	7	623	174	4,252	3,625	<1	7	333	3965	5%	>99%	99%
16	1,652	6	608	152	2,418	1,709	<1	4	257	1970	3%	>99%	99%
32	400	5	344	78	827	417	<1	4	104	525	4%	>99%	97%
64	164	7	278	50	499	166	<1	25	76	268	1%	>99%	97%
128	86	13	212	48	359	87	<1	78	59	224	1%	>99%	63%
256	62	50	67	47	226	64	<1	45	47	156	3%	>99%	33%

Table 1: QCD Dslash Time spent per iteration for $32^3 \times 256$ lattice on Endeavor Xeon Cluster

Nodes	Baseline (Time in milliseconds)					Offload (Time in milliseconds)					Internal Compute Slowdown	Post Time Reduction	Wait Time Reduction
	Internal Compute	Post Time	Wait Time	Misc Time	Total Time	Internal Compute	Post Time	Wait Time	Misc Time	Total Time			
2	314	0.085	358	313.9	986	320	0.008	44	322	686	2%	90%	87%
4	316	0.230	458	433.7	1,208	317	0.014	174	353	844	2%	94%	62%
8	313	0.759	501	432.2	1,247	329	0.027	340	311	980	5%	96%	32%
16	303	0.988	665	369.0	1,338	305	0.050	497	316	1,118	3%	95%	25%
32	303	2.334	705	431.6	1,442	308	0.084	546	322	1,176	2%	96%	22%

Table 2: FFT Time spent using Endeavor Xeon Phi coprocessor cluster

4.4 Improvements for `MPI_THREAD_MULTIPLE`

We report the message latency of the multithreaded OSU latency benchmark that uses `MPI_THREAD_MULTIPLE` on the Endeavor Xeon cluster in Figure 6. The benchmark creates several threads in each rank; each thread creates a pair with a remote rank's thread. All the thread pairs perform the OSU latency benchmark in parallel and the corresponding one-way latency is reported.

We see significant overhead when multiple threads enter MPI which increases with the number of threads; this poor scalability is common to many MPI implementations that support `MPI_THREAD_MULTIPLE`. As a result, both the *baseline* and the *comm-self* approaches suffer severely, showing up to 30 microseconds one-way latency when using eight parallel threads (Figure 6(c)).

Our offload infrastructure offers significantly better scalability arising from the lock-free algorithms used (described in Section 3) in MPI call offloading. The dedicated MPI offload thread issues MPI calls on behalf of the microbenchmark's threads using `MPI_THREAD_FUNNELED`, thereby escaping the poor scalability of MPI's multithreading support and reducing the message latency by up to 6X as compared to the *comm-self* approach. Note that the time reported here is the time for the message to be completely sent and not the time taken only to post the request to the command queue.

4.5 Overheads of offloading MPI calls

Finally, we characterize the overheads introduced by the *comm-self* and *offload* approaches using the OSU latency and bandwidth microbenchmarks.

Figure 7(a) shows the one-way latency achieved by the *baseline*, *comm-self*, and *offload* approaches using the OSU microbenchmarks run on Intel Xeon processors on the Endeavor Xeon cluster. We see that the *offload* approach shows an additional latency of 0.3 microseconds as compared to the *baseline* approach for small messages. This additional latency cost is a consequence of the offload process—converting the MPI call to a command and inserting the

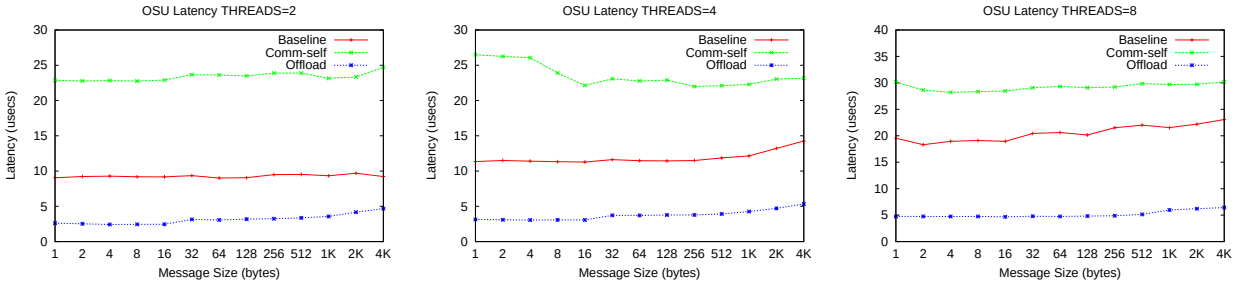

Figure 6: OSU multithreaded latency benchmark: (a) 2 threads, (b) 4 threads and (c) 8 threads

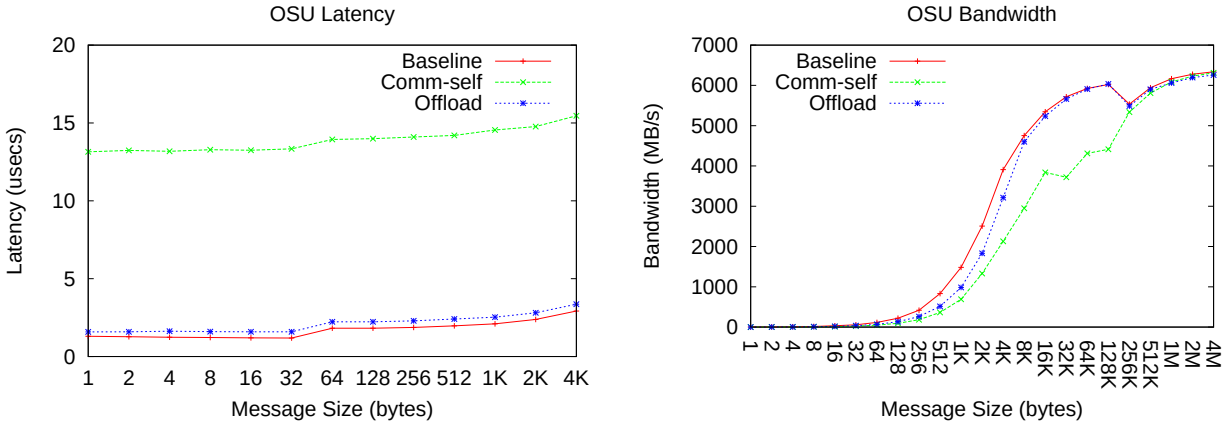

Figure 7: OSU microbenchmarks on Intel Xeon processors: (a) Latency and (b) Bandwidth

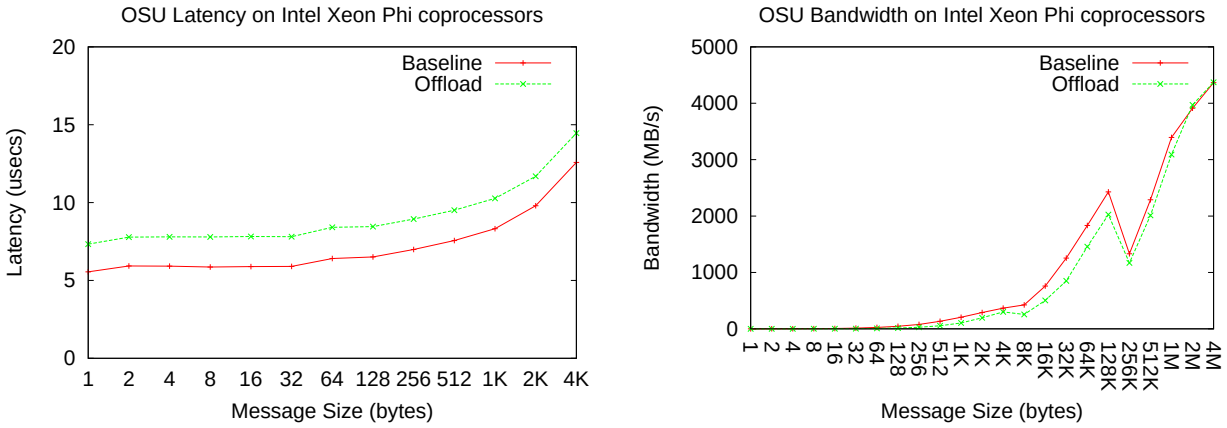

Figure 8: OSU microbenchmarks on Intel Xeon Phi coprocessors: (a) Latency and (b) Bandwidth

command into the command queue, followed by the offload thread picking up the command, issuing the MPI call and updating the response, further followed by the calling thread's retrieval of this response.

For the *comm-self* approach, we observe a much larger overhead of 11 microseconds which is a consequence of the use of `MPI_THREAD_MULTIPLE` in the MPI layer.

Figure 7(b) shows the uni-directional MPI bandwidth using the *baseline*, *comm-self*, and *offload* approaches. We see little to no degradation in bandwidth for medium to large messages from the *offload* approach as compared to the *baseline* approach.

However, we see that bandwidth drops by 50% for messages between 4 KB and 256 KB with the *comm-self* approach which can again be attributed to the overheads of `MPI_THREAD_MULTIPLE` support in the MPI layer.

We see similar trends for manycore platforms in Figures 8(a) and 8(b), although offload overhead increases to 1.7 microseconds due to lower single thread performance.

5. Applications

In this section, we provide brief descriptions of three significant HPC applications: QCD, FFT, and CNN, and present their performance using the *baseline*, *iprobe* and *comm-self* approaches as well as with our *offload* approach.

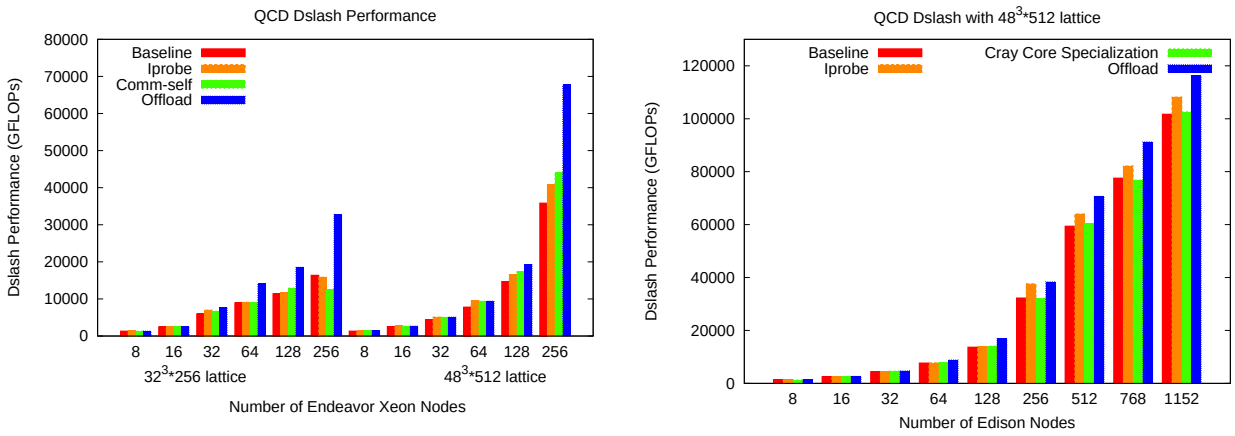

Figure 9: QCD Dslash performance: (a) Endeavor and (b) NERSC Edison

5.1 QCD

Lattice Quantum Chromodynamics (LQCD) [7, 9] is a four dimensional hypercubic lattice, with fermion (quark) fields ascribed to the lattice sites and gauge (gluon) fields ascribed to the links between sites. The Wilson-Dslash operator is used as the gauge covariant derivative in a variety of lattice Dirac Fermion operators. A large proportion of time in LQCD applications is spent solving linear systems with these operators typically using sparse iterative solvers, such as Conjugate Gradients (CG) [19] or BiCGStab [34]. This operator is very much like a 4-dimensional nearest-neighbor stencil (9 point stencil in 4 dimensions) with the added elaboration that the data at the lattice sites (so called spinors), and on the lattice links (gauge fields) are now respectively represented by complex valued matrices. Further when accumulating the stencil for the central point, the neighboring spinors must be multiplied by the gauge matrix ascribed to the link connecting the neighbor and the central site.

The communication pattern is primarily a nearest-neighbor point-to-point as described in Joo et al. [20]. Typical multi-node implementations overlap computation of the body with the communication of the faces. For the Wilson-Dslash operator, the faces (boundaries or ghost regions) are projected into separate communication buffers. Next, the boundary buffers are exchanged with the nearest neighbors using non-blocking MPI point-to-point calls. Once the internal volume (body) computation and boundary exchange completes, the received faces are multiplied appropriately with gauge links, and their contribution to Dslash is accumulated. Since the faces are non-contiguous in memory and require a projection, our previous implementation [20] packs the boundary exchange buffers using the parallelism available within the system, then initiates the send operation for each dimension in both forward and backward directions. In the CG and BiCGStab solvers, the matrix-vector product used is composed primarily of applications of the Dslash operator. Also, some level 1 BLAS operations are required as well as some global reductions for inner-product and vector norm operations, which are further exposed to latencies from `MPI_Allreduce()` operations. Hence, typically the solver has a lower performance than the Dslash operator in isolation.

Here, we work with a four dimensional hypercubic space-time lattice, and we consider the set of MPI processes as running on a four dimensional virtual processor grid with dimensions (P_x, P_y, P_z, P_t). We place one MPI task per socket and consider the MPI ranks to run lexicographically through our virtual processor grid, partitioning on the largest dimension followed by the other three dimensions (first T, then Z, followed by Y and finally X).

Wilson-Dslash performance: Figure 9(a) demonstrates the strong scaling performance of the Wilson-Dslash operator on the Endeavor cluster for $32^3 \times 256$ and $48^3 \times 512$ lattices with increasing number of nodes.

We see that our *offload* approach performs similarly to the other approaches until 16 nodes. Beyond this, our approach begins to outperform the others – the highest impact is observed at 256 nodes where we see 2X improvement, delivering 33 TFLOPs using 256 nodes for the $32^3 \times 256$ lattice.

We observe that the *comm-self* approach shows reasonable benefit for smaller node counts, but degrades significantly at 256 nodes. The message sizes at this scale are about 48 KB; we know from our experiments in Section 4.2 that this approach has large overhead for small and medium messages and this overhead outweighs the benefits of asynchronous progress. However, when we increase the lattice size to $48^3 \times 512$, we see that the *comm-self* approach begins to show improvement. The large lattice not only increases the message sizes, but also increases the surface to volume ratio which automatically gives more room for overlapping communication with computation.

Nonetheless, even for the larger lattice, our *offload* approach offers the best and our highest ever reported performance of 67 TFLOPs on 256 nodes. Note that we see super-linear speedup for 256 nodes as compared to 128 nodes mainly because the lattice starts to fit in cache. For $32^3 \times 256$ lattice, we see super-linear speedup at 32 nodes.

We see similar trends in performance using the NERSC Edison cluster as shown in Figure 9(b). Here, we additionally compare with the built-in Cray core specialization feature [28] for better asynchronous progress. We see that our *offload* approach significantly outperforms all the other approaches and again delivers our highest ever reported performance of 116 TFLOPs on 1152 nodes for the $48^3 \times 512$ lattice.

To better understand the sources of the performance improvements enabled by our approach, we show where time is spent in the Wilson-Dslash operator for the $32^3 \times 256$ lattice in Figure 10 when using the *baseline* and *offload* approaches on both the Intel Xeon and Intel Xeon Phi platforms. We see that due to better overlap in the *offload* approach, the time spent in waiting for communication to finish (shown in blue) is significantly lower when compared to the *baseline* approach. This is especially evident at 64 Intel Xeon nodes, where wait time is less than 5% for the *offload* approach

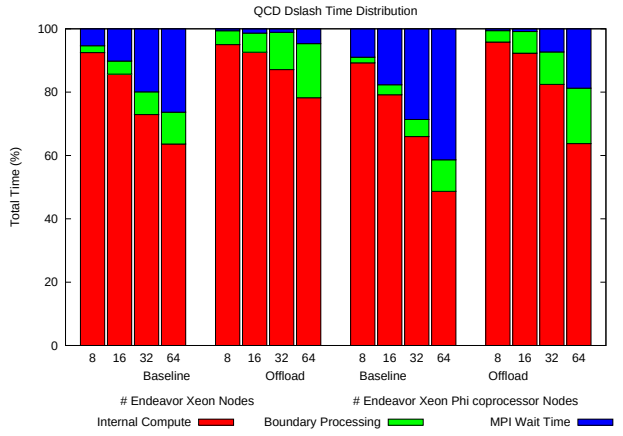

Figure 10: Wilson-Dslash timing splitup

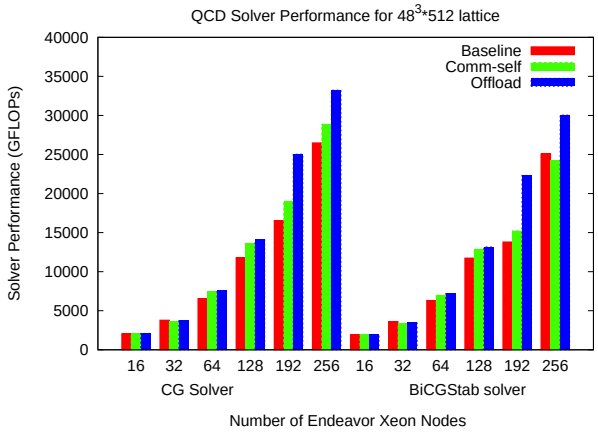

Figure 11: QCD solver performance

whereas the *baseline* approach shows about 25%. A similar trend holds for Intel Xeon Phi coprocessors.

QCD Solver Performance: In Figure 11, we show the full QCD solver performance, including CG and BiCGStab operations. The presence of global calls such as `MPI_Allreduce()` and the use of BLAS-like kernels which do not scale as well as the Wilson-Dslash operator naturally reduce the achieved performance, thus the maximum performance seen is 34 TFLOPs, with our *offload* approach.

Wilson-Dslash performance with `MPI_THREAD_MULTIPLE`: We explore the impact of allowing multiple application threads to concurrently issue MPI calls by modifying the Wilson-Dslash operator code to do so. In previous work [33], we have demonstrated a *thread-groups* library that enables simplified application use of `MPI_THREAD_MULTIPLE` by allowing groupings of threads to increase compute and communication parallelism. We use this library and study the impact of its use coupled with `MPI_THREAD_MULTIPLE` over each approach. We report performance relative to the corresponding performance without the library and with `MPI_THREAD_FUNNELED`, to highlight the benefits of improved multithreading in MPI.

Figure 12 shows these results, for the *baseline*, *iprobe*, *comm-self*, and *offload* approaches. We see that our *offload* approach outperforms all the others and shows up to 15% improvement in performance relative to using `MPI_THREAD_FUNNELED`. This result emphasizes the potential performance benefit to applications from effective multithreaded MPI.

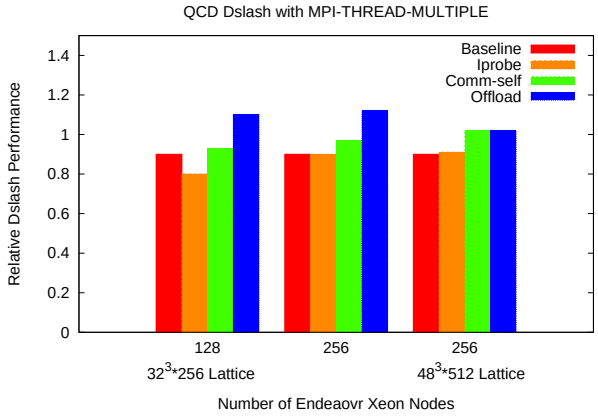

Figure 12: QCD performance with `MPI_THREAD_MULTIPLE`

5.2 FFT

Distributed 1-D FFT is a widely used HPC kernel that stresses the underlying communication infrastructure considerably. Three all-to-all data exchanges are required in the Cooley-Tukey 1-D FFT factorization [12], which is used by virtually all high-performance FFT implementations [17]. A low-communication FFT algorithm called SOI FFT has recently been devised that reduces the number of all-to-all exchanges from three to one at the expense of more computation [32]. The SOI FFT algorithm also facilitates overlapping of computation and communication by partitioning the input on each node into multiple segments and then by pipelining the computation and communication of the segments.

Figure 13(a) shows the FFT performance on the Endeavor Xeon cluster when using increasing numbers of Intel Xeon nodes. We use a problem size of 2^{29} double precision complex numbers per node and demonstrate the weak scaling performance. For smaller node counts, we see up to 20% performance improvement using the *offload* approach as compared to the *baseline* approach. The *comm-self* approach also performs well.

As node count increases to 128, due to the fact that all-to-all bandwidth does not scale with increasing node counts, we notice that the performance benefits from the *offload* approach reduce to 10% relative to the *baseline* approach, and 5% relative to the *comm-self* approach.

At 256 nodes, the problem becomes communication-bound and there is little computation available to overlap, thus we see only marginal improvement from our approach.

Figure 13(b) shows FFT performance on Intel Xeon Phi co-processors. Here, we use a problem size of 2^{25} double precision complex numbers per node and demonstrate the weak scaling performance. As `MPI_THREAD_MULTIPLE` is unsupported on this platform, we cannot evaluate the *comm-self* approach. In this figure, we see 43% performance improvement up to 4 nodes and 26% performance improvement at 64 nodes from the *offload* approach relative to the *baseline* approach. These improvements arise primarily from overheads in the MPI implementation on manycore platforms which are hidden when using *offload*.

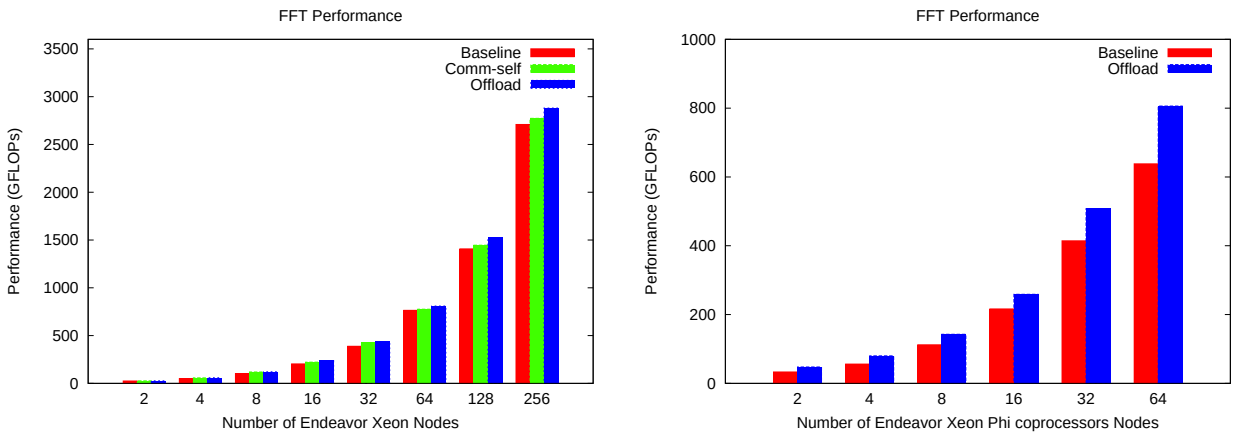

Figure 13: FFT performance: (a) Intel Xeon processors and (b) Intel Xeon Phi coprocessors

5.3 CNN

Convolutional Neural Networks (CNNs) [26] have emerged as the algorithm of choice for image, face, and speech recognition due to the superior quality of classification. This is reflected in CNNs winning visual understanding contests like the ImageNet [29] contest in recent years. CNN consists of multiple layers, each having a set of feature maps (matrices), and performing operations on the set of feature maps of the previous layer to produce output stored in feature maps of the current layer. These operations vary from convolutions, dot-products, pooling, normalization, and others. There are two computational problems pertaining to the use of CNNs: training and classification. The training problem takes as input a set of labeled images, and produces the set of weights for each layer by solving an optimization problem. The classification problem takes an image as input and performs the set of operations to produce a classification of the image. Of these, the training problem is difficult to parallelize across multiple nodes due to limited data parallelism (restricted to a small set of data points called a minibatch), while the classification problem is trivially data-parallel. Hence we present experimental results for training.

There are several ways to parallelize CNN training: data parallelism involves partitioning a minibatch of images into nodes, and model parallelism partitions the CNN into parts such that each node operates on multiple images only for a part of the CNN model. A hybrid parallelism scheme uses both data and model parallelism, data parallelism for the convolutional layers and model parallelism for the fully connected layers. These schemes are shown to have best-in-class scaling of the training problem [35]. We study the hybrid parallelism approach in this work.

The data parallel parts in the convolutional layers of the CNN have all-to-all exchange operations to exchange the set of weights and weight gradients on multiple nodes [22]. The backpropagation operation on convolution layers in one iteration passes data to the corresponding layers for forward propagation in the next iteration, which gives the potential for overlapping communication with computation. However, the fully connected layers in CNNs, which implement a slight variant of model parallelism, require synchronized all-to-all exchanges which pass the activations/gradients of activations from one stage to the next in the same iteration.

Figure 14 shows the performance of a CNN training implementation on the Endeavor Xeon cluster using different approaches. We observe that performance is roughly similar up to 8 nodes. This is because the compute time is significantly greater than the communication time, thus any improvements in overlap does not translate

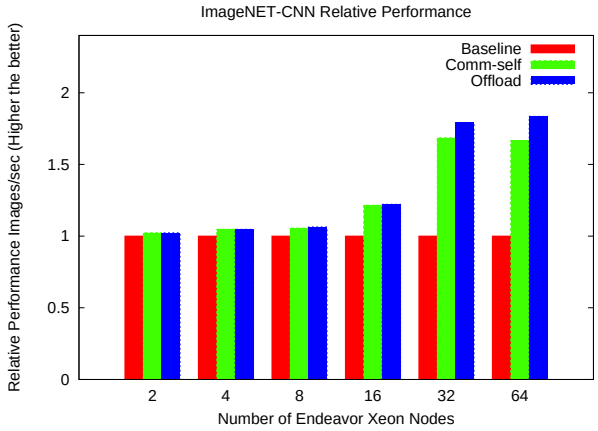

Figure 14: Deep learning CNN performance

to a performance improvement. However, as we increase the scale to 64 nodes, we notice that the *comm-self* and *offload* approaches both outperform the *baseline* approach by 2X, with the *offload* approach doing 15% better than the *comm-self* approach.

6. Related Work

The implementation of asynchronous progress on nonblocking MPI calls has been considered. Broadly speaking, four models for asynchronous progress have been studied.

Hardware-Supported Asynchronous Progress Some architectures have provided hardware support for different operations, where a hardware agent ensures that communication progresses. Mellanox® InfiniBand [2], for example, allows triggered operations, where the application (or MPI library) can post a chain of operations and the hardware asynchronously triggers operations when their dependent operations have completed. In the past, the Quadrics network architecture [27] provided capabilities similar to those of the Mellanox InfiniBand architecture. IBM® Blue Gene machines provide additional hardware to provide some collective operations such as the barrier [31]. Once initiated by a process, the hardware fully asynchronously handles the completion of the operation without additional software intervention. Hardware-based asynchronous progress is not restricted to collective operations alone. Portals-based networks such as Cray Seastar [10] provided

capabilities for hardware progress on point-to-point operations that require matching semantics (such as MPI send/recv).

In each case, asynchronous progress is made possible via some agent that drives the network in the absence of application calls into MPI to drive the progress engine. Offloading to specialized hardware such as a network card that has some or all of the capability to drive MPI operations is, however, limited to what hardware is available. No machine that we are aware of provides enough hardware to cover the full spectrum of operations in MPI that can make use of asynchronous progress (e.g., the full set of reductions or tag-matching). Thus, some software intervention is needed.

Offload engines for other protocols also exist. A common example that is often discussed is the TCP offload engine [8, 15] for TCP/IP. However, we believe that TCP is not an appropriate comparison as it is a very different environment compared to MPI. Our focus is on MPI+OpenMP applications in which even function call issue overheads can cause load imbalance and bad performance. This extreme sensitivity to the latency does not apply to typical TCP environments (e.g., web servers) where there are no tight parallel loops. In addition, MPI has a shared single rank and message ordering constraints that need to be managed. For instance, the Berkeley sockets interface to TCP has no concept of "messages", but only that of bytes—if byte-level ordering is desired the user has to do it herself. Finally, in TCP, converting blocking calls to nonblocking calls is not the same. Nonblocking calls in TCP are not message hand-offs (since there are no messages). In our proposed approach, nonblocking calls are used for actual communication hand-offs in terms of messages.

Interrupt-Based Asynchronous Progress A common model that allows hardware to trigger a software operation when needed is interrupt-driven asynchronous progress. Both Cray and IBM systems offer interrupt-driven asynchronous progress, as described in [28] and [21, 23–25]. While interrupt-driven asynchronous progress is a generic progress model, its performance tradeoffs are well understood in the community. Performance tradeoffs in the context of asynchronous one-sided communication with persistent (i.e., polling) and transient (i.e., interrupt-driven) agency on Blue Gene/P are discussed in [18]. In both cases, a low-level progress mechanism is involved, which uses a platform-specific API that is aware of the hardware features.

Threads-Based Asynchronous Progress A more customized asynchronous progress model can be implemented as part of an MPI implementation. For example, in MPICH and its various derivatives (including MVAPICH, Intel MPI, and Cray MPI), asynchronous progress can be provided by dedicating a thread that remains in the progress engine for the lifetime of the application. Often, this thread spins, consuming an entire hardware thread. A more critical issue with this approach of dedicating a software thread to poll on the progress engine is that it must hold a mutex while processing the message queue, thus preventing other threads from doing so. The performance consequences and a number of strategies for mitigation are discussed in [6].

Perhaps the closest work to the approach proposed in this paper can be found in [24], which describes the communication threads approach used on the IBM Blue Gene/Q architecture for asynchronous progress. The general idea of using communication threads to hand off work units by using atomic enqueue/dequeue operations is common between this work and our proposed approach. Two important distinguishing factors with this work is that the Blue Gene/Q approach is based on a low-level network layer, rather than the MPI implementation itself, and that it does not elide mutual exclusion in the MPI implementation. Rather, it relies upon per-object locking [14], which is finer grain than that of other implementations, but means that multiple mutexes must be acquired and released for every MPI operation. This is because asynchronous progress is provided within the PAMI library that sits underneath MPI. While PAMI provides more general asynchronous progress that supports a variety of programming models, it does not obviate mutual exclusion. While there are conceptual similarities in our approach, the PAMI layer is highly dependent on special hardware and kernel support whereas our approach is completely portable.

Shared-Memory-Based Asynchronous Progress Another approach is to avoid the need for mutual exclusion inside of MPI by using OS processes as asynchronous agents, rather than threads. This approach is described in [30] for the case of one-sided MPI communication (RMA). A major shortcoming of this approach is that the mechanism is valid only for MPI RMA and not for other operations. It can be generalized to two-sided and collective communication in cases where interprocess memory access primitives are available [11] (some implementation details are described in [16]). However, such approaches require special support from the operating system that is not available from commodity operating systems (e.g. the common Linux distributions).

7. Conclusions and Future Work

In this paper, we demonstrated a novel method for concurrent and asynchronous MPI communication in multithreaded applications. The software offload approach elides mutual exclusion by funneling communication work to a dedicated thread using a lock-free queue. This allows application threads to make concurrent MPI calls without the usual overheads associated with `MPI_THREAD_MULTIPLE` and ensures strong progress, and therefore overlap of communication and computation. All of these properties are desirable for high-performance MPI+X applications, and the cost of dedicated CPU resources for software offload is nominal on modern multicore systems with dozens of hardware threads. We have demonstrated the benefits of our approach with a number of microbenchmarks, and with three HPC applications for which we show performance increases of up to 2X.

The next steps for this project include the replacement of MPI as the communication conduit with low-level (and therefore platform-specific) APIs such as the OpenFabrics Interface [3], InfiniBand Verbs or Cray uGNI [1]. This will allow us to use multiple threads for software offload, for the APIs that support independent communication endpoints within a single process. We also intend to explore efficient implementations of other MPI operations, including RMA (i.e. one-sided).

8. Acknowledgements

We would like to thank James Dinan and Srinivas Sridharan for their useful insights on the *comm-self* approach used in this paper. We acknowledge the Endeavor Scientific Computing Center and National Energy Research Scientific Computing Center (NERSC Edison) for providing HPC resources that have contributed to the results reported in this paper. B. Joo gratefully acknowledges support by the U.S. Department of Energy, Office of Science, Office of Nuclear Physics under contract DE-AC05-06OR23177 and by the U.S. Department of Energy, Office of Science, Offices of Nuclear Physics, High Energy Physics and Advanced Scientific Computing Research under the SciDAC-3 program.

References

[1] Using the GNI and DMAPP APIs. Technical Report S-2446-5002, Cray, Mar. 2013. URL `http://docs.cray.com/books/S-2446-5002/S-2446-5002.pdf`.

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

[3] Trinity / NERSC-8 RFP. http://ofiwg.github.io/libfabric/. URL http://ofiwg.github.io/libfabric/.

[4] Trinity / NERSC-8 RFP. http://www.nersc.gov/users/computational-systems/cori/nersc-8-procurement/trinity-nersc-8-rfp/. URL http://www.nersc.gov/users/computational-systems/cori/nersc-8-procurement/trinity-nersc-8-rfp/.

[5] OSU Micro-benchmarks 4.4.1. http://mvapich.cse.ohio-state.edu/benchmarks/.

[6] A. Amer, H. Lu, Y. Wei, P. Balaji, and S. Matsuoka. MPI+threads: Runtime contention and remedies. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 239–248, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3205-7. . URL http://doi.acm.org/10.1145/2688500.2688522.

[7] R. Babich, M. A. Clark, and B. Joó. Parallelizing the QUDA Library for Multi-GPU Calculations in Lattice Quantum Chromodynamics. In *Proceedings of SC10: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2010.

[8] P. Balaji, W. Feng, Q. Gao, R. Noronha, W. Yu, and D. K. Panda. Head-to-TOE Evaluation of High Performance Sockets over Protocol Offload Engines. In *Proceedings of the IEEE International Conference on Cluster Computing (Cluster)*, Boston, Massachusetts, Sep. 27–30 2005.

[9] P. Boyle. The BlueGene/Q supercomputer. *Proceedings of Science*, Lattice Field Theory, 2012.

[10] R. Brightwell, K. Pedretti, and K. Underwood. Initial performance evaluation of the cray seastar interconnect. In *High Performance Interconnects, 2005. Proceedings. 13th Symposium on*, pages 51–57, Aug 2005. .

[11] R. Brightwell, K. Pedretti, and T. Hudson. Smartmap: Operating system support for efficient data sharing among processes on a multi-core processor. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 25:1–25:12, Piscataway, NJ, USA, 2008. IEEE Press. ISBN 978-1-4244-2835-9. URL http://dl.acm.org/citation.cfm?id=1413370.1413396.

[12] J. W. Cooley and J. W. Tukey. An Algorithm for the Machine Computation of Complex Fourier Series. *MATHCOMP*, 19(2):297–301, 1965.

[13] A. CORAL: Collaboration of Oak Ridge and Livermore National Laboratories. DRAFT CORAL BUILD STATEMENT OF WORK. Technical Report LLNL-PROP-636244, Lawrence Livermore National Laboratory, Dec. 2013. URL https://asc.llnl.gov/CORAL/RFP_components/02_draft_CORAL_Build_SOW_12-31-13.pdf.

[14] G. Dózsa, S. Kumar, P. Balaji, D. Buntinas, D. Goodell, W. Gropp, J. Ratterman, and R. Thakur. Enabling concurrent multithreaded MPI communication on multicore petascale systems. In R. Keller, E. Gabriel, M. Resch, and J. Dongarra, editors, *Recent Advances in the Message Passing Interface*, volume 6305 of *Lecture Notes in Computer Science*, pages 11–20. Springer Berlin Heidelberg, 2010.

[15] W. Feng, P. Balaji, C. Baron, L. N. Bhuyan, and D. K. Panda. Performance Characterization of a 10-Gigabit Ethernet TOE. In *Proceedings of the IEEE International Symposium on High-Performance Interconnects (HotI)*, Palo Alto, CA, Aug. 17–19 2005.

[16] A. Friedley, T. Hoefler, G. Bronevetsky, A. Lumsdaine, and C.-C. Ma. Ownership passing: Efficient distributed memory programming on multi-core systems. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 177–186, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1922-5. . URL http://doi.acm.org/10.1145/2442516.2442534.

[17] M. Frigo and S. G. Johnson. The Design and Implementation of FFTW. *IEEEP*, 93:216–231, 2005.

[18] J. R. Hammond, S. Krishnamoorthy, S. Shende, N. A. Romero, and A. D. Malony. Performance characterization of global address space applications: a case study with NWChem. *Concurrency and Computation: Practice and Experience*, 24(2):135–154, 2012. ISSN 1532-0634. . URL http://dx.doi.org/10.1002/cpe.1881.

[19] M. R. Hestenes and E. Stiefel. Methods of Conjugate Gradients for Solving Linear Systems. *Journal of Research of the National Bureau of Standards*, 1952.

[20] B. Joo, D. D. Kalamkar, K. Vaidyanathan, M. Smelyanskiy, K. Pamnany, V. W. Lee, P. Dubey, and W. W. III1. Lattice QCD on Intel Xeon Phi. In *Proceedings of ISC13: International Conference for Super Computing*, 2013.

[21] M. Krishnan, J. Nieplocha, M. Blocksome, and B. Smith. Evaluation of remote memory access communication on the IBM Blue Gene/P supercomputer. In *International Conference on Parallel Processing - Workshops, 2008. ICPP-W '08.*, pages 109–115, Sept 2008. .

[22] A. Krizhevsky. One weird trick for parallelizing convolutional neural networks. In *Computing Research Repository (CoRR)*, 2014.

[23] S. Kumar, G. Dozsa, G. Almasi, P. Heidelberger, D. Chen, M. E. Giampapa, M. Blocksome, A. Faraj, J. Parker, J. Ratterman, B. Smith, and C. J. Archer. The deep computing messaging framework: generalized scalable message passing on the Blue Gene/P supercomputer. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, pages 94–103, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-158-3. .

[24] S. Kumar, A. Mamidala, D. Faraj, B. Smith, M. Blocksome, B. Cernohous, D. Miller, J. Parker, J. Ratterman, P. Heidelberger, D. Chen, and B. Steinmacher-Burrow. PAMI: A parallel active message interface for the Blue Gene/Q supercomputer. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 763–773, 2012. .

[25] S. Kumar, Y. Sun, and L. Kale. Acceleration of an asynchronous message driven programming paradigm on IBM Blue Gene/Q. In *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 689–699, May 2013. .

[26] Y. LeCun, F. Huang, and L. Bottou. Learning methods for generic object recognition with invariance to pose and lighting. In *In Computer Vision and Pattern Recognition, CVPR*, 2004.

[27] F. Petrini, W. Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The Quadrics Network (QsNet): High-Performance Clustering Technology. In *HotI '01*, 2001.

[28] H. Pritchard, D. Roweth, D. Henseler, and P. Cassella. Leveraging the cray linux environment core specialization feature to realize mpi asynchronous progress on cray xe systems. In *Proceedings of the Cray User Group Conference*, 2012.

[29] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. In *International Journal of Computer Vision (IJCV)*, 2015.

[30] M. Si, A. J. Pena, J. Hammond, P. Balaji, M. Takagi, and Y. Ishikawa. Casper: An asynchronous progress model for MPI RMA on many-core architectures. *29th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, May 2015. URL http://www.mcs.anl.gov/papers/P5221-1014.pdf.

[31] I. S. B. G. Solution. Blue Gene/P application development redbook, 2008. http://www.redbooks.ibm.com/abstracts/sg247287.html.

[32] P. T. P. Tang, J. Park, D. Kim, and V. Petrov. A Framework for Low-Communication 1-D FFT. In *Proceedings of SC12: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2012.

[33] K. Vaidyanathan, K. Pamnany, D. K. Kalamkar, A. Heinecke, M. Smelyanskiy, J. Park, D. Kim, A. Shet, B. Kaul, B. Joo, and P. Dubey. Improving Communication Performance and Scalability of Native Applications on Intel Xeon Phi Coprocessor Clusters. In *Proceedings of IPDPS: International Parallel and Distributed Processing Symposium*, 2014.

[34] H. A. van der Vorst. BI-CGSTAB: a fast and smoothly converging variant of BI-CG for the solution of nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 1992.

[35] R. Wu, S. Yan, Y. Shan, Q. Dang, and G. Sun. Deep Image: Scaling up Image Recognition. In *Computing Research Repository*, 2015.