MT-MPI: Multithreaded MPI for Many-Core Environments

Min Si
University of Tokyo, Tokyo, Japan
msi@il.is.s.u-tokyo.ac.jp

Antonio J. Peña
Argonne National Laboratory, USA
apenya@mcs.anl.gov

Pavan Balaji
Argonne National Laboratory, USA
balaji@mcs.anl.gov

Masamichi Takagi
NEC Corporation, Kawasaki, Japan
m-takagi@ab.jp.nec.com

Yutaka Ishikawa
University of Tokyo, Tokyo, Japan
ishikawa@is.s.u-tokyo.ac.jp

ABSTRACT

Many-core architectures, such as the Intel Xeon Phi, provide dozens of cores and hundreds of hardware threads. To utilize such architectures, application programmers are increasingly looking at hybrid programming models, where multiple threads interact with the MPI library (frequently called "MPI+X" models). A common mode of operation for such applications uses multiple threads to parallelize the computation, while one of the threads also issues MPI operations (i.e., MPI `FUNNELED` or `SERIALIZED` thread-safety mode). In MPI+OpenMP applications, this is achieved, for example, by placing MPI calls in OpenMP critical sections or outside the OpenMP parallel regions. However, such a model often means that the OpenMP threads are active only during the parallel computation phase and idle during the MPI calls, resulting in wasted computational resources. In this paper, we present MT-MPI, an internally multithreaded MPI implementation that transparently coordinates with the threading runtime system to share idle threads with the application. It is designed in the context of OpenMP and requires modifications to both the MPI implementation and the OpenMP runtime in order to share appropriate information between them. We demonstrate the benefit of such internal parallelism for various aspects of MPI processing, including derived datatype communication, shared-memory communication, and network I/O operations.

Categories and Subject Descriptors: D.4 [Communications Management]: Message sending

Keywords: MPI; OpenMP; hybrid MPI + OpenMP; threads; many-core; Xeon Phi;

1. INTRODUCTION

Although multicore processor chips are the norm today, architectures such as the Intel Xeon Phi take such chips to a new level of parallelism, with dozens of cores and hundreds of hardware threads. With the number of processing cores increasing at a faster rate than are other resources in the system (e.g., memory), application programmers are looking at hybrid programming models, comprising a mixture of processes and threads, that allow resources on a node to be shared between the different threads of a process. In such models, one or more threads utilize a distributed-memory programming system, such as MPI, for their data communication. The most prominent of the threading models used in scientific computing today is OpenMP [5]. In OpenMP, the application developer annotates the code with information on which statements need to be parallelized by the compiler and the associated runtime system. The compiler, in turn, translates these annotations into semantic information that the runtime system can use to schedule the computational work units on multiple threads for parallel execution.

A common mode of operation for hybrid MPI+OpenMP applications involves using multiple threads to parallelize the computation, while one of the threads issues MPI operations (i.e., MPI `FUNNELED` or `SERIALIZED` thread-safety mode). This is achieved, for example, by placing MPI calls in OpenMP critical sections or outside the OpenMP parallel regions. However, such a model often means that the OpenMP threads are active only in the computation phase and idle during MPI calls, resulting in wasted computational resources. These idle threads translate to underutilized hardware resources on massively parallel architectures.

In this paper, we present MT-MPI, an internally multithreaded MPI implementation that transparently coordinates with the threading runtime system to share idle threads with the application. We designed MT-MPI in the context of OpenMP, which serves as a common threading runtime system for the application and MPI. MT-MPI employs application idle threads to boost MPI communication and data-processing performance and increases resource utilization. While the proposed techniques are generally applicable to most many-core architectures, in this paper we focus on Intel Xeon Phi as the architectural testbed (in "native mode," where applications are executed directly on the coprocessor).

To demonstrate the performance benefits of the proposed approach, we modified the Intel OpenMP runtime (http://www.openmprtl.org) and the MPICH implementation of MPI (http://www.mpich.org). Specifically, we modified the MPI implementation to parallelize its internal processing using a potentially nested OpenMP parallel instantiation (i.e., one OpenMP parallel block inside another). We studied new algorithms for various internal processing steps within MPI that are more "parallelism friendly" for OpenMP to use. In theory, such a model would allow both the application and the MPI implementation to expose their parallelism requirements to the OpenMP runtime, which in turn can schedule

ICS'14, June 10–13 2014, Munich, Germany.

ACM 978-1-4503-2642-1/14/06 ...$15.00.
http://dx.doi.org/10.1145/2597652.2597658.

them on the available computational resources. In practice, however, this has multiple challenges:

1. The modified algorithms for internal MPI processing, while efficient for OpenMP parallelism, are in some cases not as efficient for sequential processing. Consequently, they can improve performance only when sufficient OpenMP parallelism is available. However, the actual number of threads that will be available at runtime is unknown. Depending on the application's usage of threads, this can vary from none to all threads being available to MPI for processing. Thus, if not designed carefully, the algorithms can perform worse than the traditional sequential implementation of MPI.
2. Unfortunately, the current implementation of the Intel OpenMP runtime does not schedule work units from nested OpenMP parallel regions efficiently. It simply creates new pthreads for each nested parallel block and allows the operating system to schedule them on the available cores. This results in creating more threads than the available cores, and degrading performance.

To work around these limitations, we modified the Intel OpenMP runtime to expose information about the idle threads to the MPI implementation. The MPI implementation uses this information to schedule its parallelization only when enough idle resources were available. Furthermore, such information allows the MPI implementation to selectively choose different algorithms that trade off between parallelism and sequential execution in order to achieve the best performance in all cases.

We present our parallelization designs for three different parts within the MPI implementation: (1) packing and unpacking stages involved in derived datatype processing and communication, (2) shared-memory data movement in intranode communication, and (3) network I/O operations on InfiniBand. We also present a thorough experimental evaluation, validation, and analysis using a variety of micro- and macrokernels, including 3D halo exchanges, NAS MG benchmark, and the Graph500 benchmark [11].

2. BACKGROUND

In this section we provide some details about the Intel Xeon Phi architecture and the different threading modes defined by MPI for multithreaded environments.

2.1 Intel Xeon Phi Architecture

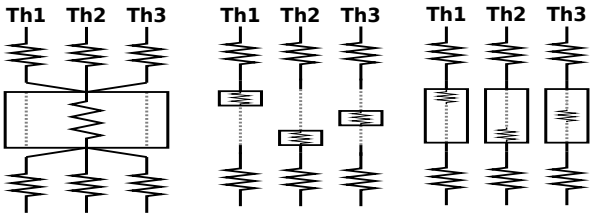
The Intel Xeon Phi architecture features a large number of CPU cores inside a single chip. The Xeon Phi cards run their own Linux-based operating system and can launch full operating system processes. In the native mode, system calls that cannot be handled directly on the Xeon Phi card are transparently forwarded to the host processor, which executes them and sends the result back to the issuing process. Although these devices also offer the possibility of running in *offload mode*, following a GPU-like approach, this mode is not considered in our research because it does not allow the coprocessors to run hybrid MPI + OpenMP applications.

When MPI processes are launched on a combination of multiple nodes and adapters, these processes internally communicate with each other using a number of mechanisms. Processes on the same Xeon Phi card communicate with each other using shared memory. Processes on the same node communicate using the PCIe peer-to-peer capabilities. When communicating outside the node, for some networks such as InfiniBand, communication is performed directly without host intervention through the PCIe root complex.

The first generation of the product released to the public, code-named Knights Corner [4], features a minimum of 60 simple cores each capable of 4 hardware threads, providing a total of 240 hardware threads per coprocessor. The card is equipped with 8 GB of GDDR5 RAM. One difference between this architecture and GPU architectures is the fully private and coherent cache provided to each processing unit: 32 KB instruction + 32 KB data L1, and 512 KB L2 (unified), offering high data bandwidth. Further details on the Intel Xeon Phi architecture can be found in [9, 4].

2.2 Hybrid Programming Models

The MPI standard provides four levels of thread safety.



(a) FUNNELED. (b) SERIALIZED. (c) MULTIPLE.

Figure 1: Threading modes in MPI. A line represents a thread; the zigzag part represents an active thread in an OpenMP region; the straight part represents a thread outside an OpenMP region; the dotted part represents an idle thread in an OpenMP region; the boxes represent MPI calls.

```
#pragma omp parallel
{

  /* user computation */

}

MPI_Function();
```

(a) Outside a parallel region

```
#pragma omp parallel
{
  /* user computation */
  #pragma omp master
  {
    MPI_Function();
  }
}
```

(b) Inside omp `master` region

```
#pragma omp parallel
{
  /* user computation */
  #pragma omp critical
  {
    MPI_Function();
  }
}
```

(c) Inside omp `critical` region

```
#pragma omp parallel
{
  /* user computation */
  #pragma omp single
  {
    MPI_Function();
  }
}
```

(d) Inside omp `single` region

Figure 2: Different use cases in hybrid MPI+OpenMP.

MPI_THREAD_SINGLE. In this mode, a single thread exists in the system. This model is commonly referred to as the MPI-only model, where a bunch of MPI processes communicate with each other and no threads are involved.

MPI_THREAD_FUNNELED. In this mode, multiple threads can exist, but only the master thread (the one that initialized MPI) is allowed to make MPI calls. Different threads can parallelize computational phases, but all MPI communication has to be funneled through the main thread (see Figure 1(a)). In typical OpenMP environments, this involves making MPI calls either outside the OpenMP parallel region (Figure 2(a)) or within OpenMP master regions (Figure 2(b)).

MPI_THREAD_SERIALIZED. In this mode, multiple threads can exist, and any thread can make MPI calls but only one thread at a time. Different threads can parallelize computational phases, but the threads need to synchronize in order to serialize their MPI calls (Figure 1(b)). In typical OpenMP environments, this involves making MPI calls within OpenMP critical regions (Figure 2(c)) or single regions (Figure 2(d)).

MPI_THREAD_MULTIPLE. In this mode, multiple threads can exist, and any thread can make MPI calls at any time (Figure 1(c)). The MPI implementation is responsible for using appropriate synchronization to protect accesses to shared internal data structures.

In this paper we focus on FUNNELED/SERIALIZED modes.

3. DESIGN AND IMPLEMENTATION

In this section we describe the design of MT-MPI, including modifications to the MPICH implementation of MPI (v3.0.4) and the Intel OpenMP runtime (version 20130412).

3.1 OpenMP Runtime

As described in Section 1, for MPI to share OpenMP parallelism with the application, two challenges need to be addressed. The first is the different MPI internal algorithms that trade off between parallelism and faster sequential execution. The second is the behavior of nested parallel regions in current OpenMP implementations, including that of the Intel OpenMP runtime that is used on Xeon Phi architectures. Specifically, the OpenMP runtime creates new pthreads for each nested OpenMP region, thus creating more threads than the available cores and degrading performance.

To handle these issues, we modified the Intel OpenMP runtime to expose the number of idle threads to the MPI implementation. The idea is for the OpenMP runtime system to track how many threads are being used by the application vs. how many threads are idle (e.g., because they are in an OpenMP barrier or outside an OpenMP parallel region). Then, the OpenMP runtime can provide this information through a new runtime function. The expectation in this model is that MPI could query for the number of idle threads and use this information to (1) choose the most efficient internal parallelization algorithms and (2) use only as many threads in the nested OpenMP region as there are idle cores, by explicitly guiding the number of threads in OpenMP (using the `num_threads` clause in OpenMP).

Arguably, the second challenge described above (additional pthreads created in nested OpenMP regions) is an issue only with the current implementation of the Intel OpenMP runtime. An alternative OpenMP runtime that internally uses user-level threads (e.g., [12]) might not face this challenge. However, given that most OpenMP implementations today use pthreads internally and that Intel OpenMP is the only formally supported OpenMP implementation on the Xeon Phi architecture, we consider this to be a real problem that needs to be addressed.

3.1.1 Exposing Idle Threads

To expose the number of idle threads in OpenMP, we need to understand the status of threads in the following cases.

MPI call made outside the OpenMP parallel regions (Figure 2(a)). In this case, all threads except the main thread are idle (often equal to `OMP_NUM_THREADS`). Thus, we expect MPI to be able to benefit from a large number of idle threads.

MPI call made in an OpenMP single region (Figure 2(d)). OpenMP single regions provide an implicit barrier on exit. Thus, we can ideally expect threads to be available "soon" if the number of idle threads is queried within an OpenMP single region. In practice, however, not all threads might have reached the barrier yet, for example, because there is some skew between the threads or because they are working on a user computation. Thus, the number of idle threads available can vary anywhere between zero and the maximum number of threads. We modified the OpenMP runtime to track each thread in order to return the actual number of idle threads. In this case, the amount of parallelism available to MPI is unknown in the general case. However, for OpenMP parallel regions where the work shared between threads is mostly balanced and threads are reasonably synchronized, the number of idle threads is expected to be close to the maximum number of threads.

MPI call made in an OpenMP master region or single region with a nowait clause (Figure 2(b)). This case is similar to the previous case (single region) with the primary difference that there is no implied barrier at the end of such a region. Hence, there is no natural synchronization point for the threads. Nevertheless, depending on how the application is written, it is possible to have an external synchronization point (such as a user-specified OpenMP barrier) that would cause more idle threads to be available. Consequently, we use a similar solution here as in the previous case, that is, to track the number of idle threads. In practice, however, we do not expect too many idle threads to be available for MPI to use in this case.

MPI call made in an OpenMP critical region (Figure 2(c)). OpenMP critical regions force some synchronization between threads because only one thread can enter a critical region at a time. While this is not quite an implicit barrier, its behavior with respect to the availability of threads can be similar to that of an OpenMP single region. Specifically, when the first thread enters the OpenMP critical region, the remaining threads can be ideally expected to be idle "soon." As discussed earlier, this is not necessarily true if the other threads are busy with the user computation or are skewed, but it can give a reasonable model for us to consider. When the second thread enters the OpenMP critical region, the first thread is no longer expected to be idle because it has already finished executing its critical region. Similarly, when the last thread enters the critical region, none of the remaining threads are expected to be idle because they have all finished executing their critical regions. As in the previous cases, we track the number of idle threads inside the OpenMP runtime, although we expect that the number of idle threads would be high for the first few threads entering the critical section and low for the last few threads.

In some of the cases described above (e.g., single region with nowait), utilizing the idle threads can be risky because their status can change at any time. For example, they might have been idle because they were in an unrelated critical section that has now completed. This would cause those idle threads to become active again, degrading performance. In our implementation, we distinguish how many threads are "guaranteed to be idle" and how many are "temporarily available at the current time." To understand this distinction, we

need to look into when a thread can be idle. There are two cases when a thread can be idle: (1) if it is waiting in a barrier waiting for other threads in the team to arrive or (2) if it is outside a critical section waiting to enter it.

A thread that is in a barrier is guaranteed to be idle till all other threads in that team reach the barrier. Thus, when a thread in that team queries for the number of guaranteed idle threads, all the threads that are waiting in the barrier will contribute to the returned value. Waiting to enter a critical section is a bit more tricky in that a thread is guaranteed to wait only till the thread that is already in the critical section does not exit the critical section. Thus, if the thread that is already in the critical section queries for the number of guaranteed idle threads, the threads waiting to enter the critical section will contribute to the returned value. For all other threads, the threads waiting to enter the critical section will not contribute to the guaranteed idle threads but will contribute to the temporarily available threads.

Thus, the following semantics hold true for the number of guaranteed idle threads:

1. It is thread-specific. At a given point of time, depending on which thread is querying for the information, the returned value might be different (it can increase or decrease).
2. It is OpenMP-region specific. If the querying thread enters a new OpenMP region (e.g., critical or single) or exits it, the returned value might be different (it can increase or decrease).
3. It is time-specific. At two different points of time the returned value might be different (e.g., if more threads reached a barrier). However, if the same thread queries for the value and it is in the same OpenMP region, the value can only increase, not decrease.

We modified the OpenMP runtime to keep track of which type of OpenMP region each thread is in, in order to return both the guaranteed number of idle threads and the number of temporarily idle threads. We note that our implementation treats a thread as idle only when it is not engaged in any OpenMP activity, including OpenMP parallel loops and OpenMP tasks. We also note that in our implementation the performance overhead associated with tracking whether a thread is actively being used by the OpenMP runtime is too small to be observed and hence is not demonstrated in this paper.

3.1.2 Thread Scheduling for Nested Parallelism

For well-balanced OpenMP parallel loops with little to no skew, thread synchronizations such as barriers are often short-lived because threads tend to arrive at the barrier at approximately the same time. Thus, when a thread arrives at a barrier, if it is put to sleep while waiting for the other threads to arrive, only to be woken up in a short amount of time, performance is degraded because of the cost of waking up threads from a sleep state. To work around this situation, the Intel OpenMP runtime does not put threads to sleep immediately when they reach a barrier. Instead, they spin waiting for other threads to arrive, for a configurable amount of time: `KMP_BLOCKTIME`. A large value for this variable would mean that threads do not become truly idle for a long time. While this situation is not a concern for regular OpenMP parallel loops, it can degrade performance for nested OpenMP parallel loops since the Intel OpenMP runtime creates more threads than the number of cores in such cases. Having the primary threads spin for `KMP_BLOCKTIME` would cause more threads to be active than the number of available cores for that much time.

When MPI calls are outside the application OpenMP parallel region (such as in Figure 2(d)), this is not a concern since MPI would use the same threads as the application in its parallel region. When MPI calls are inside the application parallel region, however, this would require MPI to use a nested OpenMP parallel region. And since the threads that arrived at the barrier would not yield the available cores immediately, this would either require MPI to utilize lesser parallelism by only using the idle cores or cause thread thrashing on the available cores for `KMP_BLOCKTIME` amount of time. Neither solution is ideal.

In MT-MPI, to be able to employ these resources as soon as possible, we implemented and exposed a new function in the OpenMP runtime: `set_fast_yield`. This function plays two roles. First, it forces the threads in the current team to skip the active wait during the barrier operation and immediately yield the core. Second, it continuously yields the core using `sched_yield` calls instead of simply sleeping. We used this approach primarily because of the overhead associated with sleep vs. that of yield. We found that yielding allows us to manage the cores with a much lower overhead (about 30 μs even with 240 threads) compared with sleeping (more than 100 μs even at 16 threads).

We note that (1) our thread scheduling optimization impacts only those threads that are guaranteed to be idle (e.g., threads waiting in an OpenMP barrier); (2) the fast yield setting is performed internally inside the MPI call and reset once the internal parallelism in MPI is complete, so future OpenMP barriers are not affected by it; and (3) the proposed thread scheduling optimization affects only that case when MPI uses nested OpenMP parallelism (e.g., when an MPI function is called in an OpenMP single region) and does not affect the case when the MPI function is called outside the OpenMP parallel region.

3.2 MPI Internal Parallelism

Using the information about the idle threads exposed by our extended OpenMP runtime, the MPI implementation can schedule its internal parallelism efficiently to obtain performance improvements. In this section, we demonstrate the benefit of such internal parallelism for various aspects of the MPI processing, including derived datatype communication, shared-memory communication, and network I/O operations. In our MPI implementation, we utilize only those idle threads that are guaranteed to be available. Although here we do not utilize temporarily available threads, one could envision cases (e.g., short MPI operations) where they could be. In our implementation, when all threads are idle (e.g., when the MPI call is outside the OpenMP parallel region), we do not specify the number of threads to be utilized by OpenMP; instead, we let it manage such parallelism internally. If fewer than the maximum number of threads is idle, however, we direct the amount of thread parallelism to use through the `num_threads` OpenMP clause.

3.2.1 Derived Datatype Processing

MPI allows applications to describe noncontiguous regions of memory using user-derived datatypes such as *vector*, *indexed*, and *struct*. These derived datatypes can be used to describe arbitrarily complex data layouts to be processed

```
for (i=0; i<count; i++){
  *dest++ = *src;
  src += stride;
}
```
(a) Sequential implementation.

```
#pragma omp parallel for
for (i=0; i<count; i++){
  dest[i] = src[i * stride];
}
```
(b) Parallel implementation.

Figure 3: Sequential and parallel data packing.

by MPI for packing/unpacking data to/from a contiguous buffer (using `MPI_PACK` and `MPI_UNPACK`) or to send/receive data. When communicating using derived datatypes, MPI implementations typically internally pack data into contiguous buffers, communicate these contiguous buffers, and internally unpack them into the recipient buffer. Halo exchanges [19] are a well-known example of communications that are well suited to employ derived datatypes.

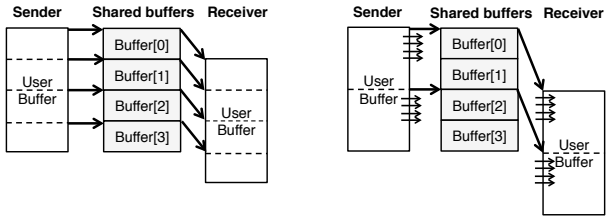
The pack and unpack processing stages consist of a set of local memory copies. A typical implementation traverses the derived datatype tree and copies each noncontiguous chunk of data separately. Some implementations of MPI optimize such processing by representing the entire datatype as a stack structure so that it can be iteratively traversed rather than using a recursive traversal [14]. Given that each noncontiguous data chunk is copied to a different location and there are no dependencies among the different data elements, such copies are a good candidate for OpenMP parallelization. Moreover, thanks to the relatively large private caches per core on the Xeon Phi architecture, concurrent accesses to separate memory regions by the different threads are expected to be highly efficient. Therefore, we modified the MPI implementation to parallelize the datatype data copy using OpenMP. We note that only the lowest level of a nested datatype (e.g., a vector of vectors) is parallelized in MT-MPI.

One issue that we found using MT-MPI was an unintended consequence of the compiler vectorization. The original datatype copy code that is used in MPICH is shown in Figure 3(a). While this code works correctly for sequential data copy, it cannot be easily parallelized by using OpenMP because the compiler cannot understand the constant stride of accesses used through all iterations. We therefore modified the code as shown in Figure 3(b). While this new implementation makes it easier for the compiler to understand the computation and thus parallelize it, the implementation also makes it easier for the compiler to vectorize the code. This situation in itself is not a concern. However, the Intel compiler is inefficient in vectorizing strided loops with large stride values when the amount of data copied in each loop is small. Specifically, the compiler does incorrect prefetching in this case, causing additional cache misses and thus losing performance. Consequently, our modification to the code is not always beneficial and can perform worse than the sequential implementation when very few threads are available. To work around this issue, we could either disable vectorization in the parallel implementation or explicitly choose only the parallel approach when a sufficiently large number of threads are available. We chose the latter approach because vectorization is still beneficial in some cases (e.g., when the stride is small or the copy size is large).

We note that the incorrect cache prefetching and additional cache misses that it causes have been experimentally verified, but the results are not shown in this paper because of space restrictions. The issue has also been reported to Intel and has been confirmed by their compiler team. They are expected to fix it in a future release of the compiler.

3.2.2 *Shared-Memory Communication*

When multiple MPI processes reside on the same node, since each process has a different virtual address space, most MPI implementations, including MPICH, use a pipelined double-copy strategy through shared memory for intranode communication [3]. As shown in Figure 4(a), a shared-memory ring buffer is allocated between the sender and receiver processes and divided into multiple cells; the sender process then copies part of data into an empty cell while the receiver process copies a full cell out.



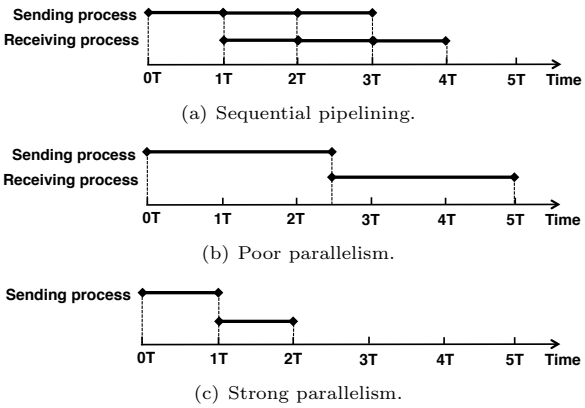
(a) Sequential pipelining. (b) Parallel pipelining.

Figure 4: Data movement of parallelization and pipelining.

In MT-MPI, we parallelize this copy on both the sender and the receiver side using the available idle threads. We implemented this optimization by extending the pipelined double-copy strategy used within MPICH. As shown in Figure 4(b), in our approach we reserve multiple contiguous available cells and concurrently copy data from the user buffer to these cells on the sender side and from the cells to the user buffer on the receiver side. For messages larger than what can be held in the reserved cells, additional pipelining is used, similar to the sequential case. Compared with the sequential pipelining algorithm, however, the parallel algorithm can degrade performance in the following cases.

Small messages. When the message size is small, there is not enough work in MPI to be parallelized. In such cases, the thread management and synchronization within OpenMP are more expensive than the sequential copy mechanism already used in MPICH. Thus, in this case we do not expect any performance benefit from parallelization.

Large messages but few idle threads. In our parallel implementation, we reserve as many shared-memory cells as possible and parallelize the copy using all of the available idle threads. Thus the receiver process now has to wait until data is filled into all of the reserved cells before it can start its data copy out of shared memory. This approach, in essence, increases the pipeline unit to a much larger size. Thus, while parallelism can improve the performance of each memory copy operation, it can also hurt the data copy pipeline. We note that we cannot simply reduce the size of each shared-memory cell or the maximum number of cells reserved to work around this issue because that would reduce the amount of work done by each thread, thus causing the thread management overhead to dominate. This trend is illustrated in Figure 5. Specifically, compared with sequential pipelining (Figure 5(a)), when the number of threads available to MPI is small, the parallel copy does not improve performance much but delays the receiver process from getting started with its copy (Figure 5(b)). On the other hand, when a large number of threads are available to MPI, the parallel copy is significantly faster, thus balancing the loss of performance due to the reduced pipelining (Figure 5(c)).



(a) Sequential pipelining.

(b) Poor parallelism.

(c) Strong parallelism.

Figure 5: Sequential pipelining vs. parallel data copy.

Few shared-memory cells. The amount of data to be copied is decided not just on the number of available threads but also on the number of available shared-memory cells. Specifically, during the communication process, some cells might be in use for transferring previous messages (or previous parts of the same message). In such cases, there is not enough work to be parallelized, and hence the thread management would add too much overhead to justify the performance improvement through parallelization.

In summary, parallelism would improve performance only when (1) the message size is not too small ($\geq$ 64 Kbytes), (2) the number of threads is not too few ($\geq$ 8), and (3) the total size of free cells is not too small ($\geq$ 64 Kbytes). In our implementation, we utilize the parallel shared-memory communication algorithm only when all three conditions are met; otherwise we fall back to the original sequential algorithm. We note that the above-mentioned thresholds are empirically evaluated on our test platform and must be tuned for different platforms.

3.2.3 *Optimizations for the InfiniBand Network*

Several MPI implementations are optimized for a variety of networks through a layered software architecture where one of the layers provides network-specific functionality. In MPICH, this layer is called the `netmod layer`. Multiple netmod implementations exist for MPICH over InfiniBand (IB), with more-or-less similar functionality and performance. In this paper we utilize the implementation described in [17].

An MPI implementation utilizing IB needs to create and manage a number of objects, including contexts, protection domains (PDs), queue pairs (QPs), and completion queues (CQs). A process can create one or more IB contexts, each of which maintains a collection of state information associated with the communication. Each context can contain one or more PDs, each of which defines the protection semantics of memory and other objects used by the program, for example to allow different connections access to different sets of memory regions. Within a PD, the program can create one or more QPs, each of which consists of a send queue and a receive queue. A QP is used to communicate between a pair of processes. A PD can also have one or more CQs, each of which is used to check for the completion of communication operations on one or more QPs. IB also provides shared queues for better memory management, but for simplicity we do not describe them here.

The IB software stack [13] is thread-safe. When multiple threads access the same QP or CQ, it internally uses mutexes to maintain state consistency. Such state consistency is expensive, however, and can degrade performance. Therefore, in our approach we try to avoid such usage and instead have different threads manage different QPs in order to maximize performance. Even with this approach some shared data structures still need to be protected. To understand how much performance improvement MPI can gain by parallelizing the posting of network operations, we studied how much potential parallelism there is in the IB stack that can theoretically be exploited. We modified the `ib_write_bw` benchmark from the OpenFabrics Enterprise Distribution (OFED) package [13] to measure the multithreaded point-to-point IB RDMA write bandwidth between two Intel Xeon Phi coprocessors on different nodes. We define three parallelism levels:

IB contexts. Each process has 64 IB contexts, and each context has one QP and one CQ. Each thread handles operations on a different context, CQ and QP.

QPs and CQs. Each process has a single IB context with 64 QPs and 64 CQs. Each CQ is dedicated to a different QP. Each thread handles operations on different QPs and CQs, but they all share the same context.

QPs only. Each process has a single IB context with 64 QPs and one shared CQ. Each thread handles operations on different QPs, but they all share the same context and CQ.

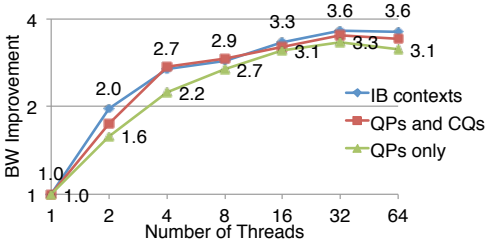


Figure 6: Small (64-byte) IB RDMA write bandwidth.

Figure 6 compares the communication bandwidth of small messages (64 bytes) for the cited parallelism levels. We make two primary observations from the figure. The first is that the performance improvement with increasing threads is higher when the number of shared resources is less. For example, when each thread has a separate context (`IB contexts`), with increasing threads, the parallel performance is 3.6-fold higher than the sequential performance. But when the context and the CQ are shared by all threads (`QPs only`), the parallel performance is only 3.1-fold higher than the sequential performance. This result is expected because more sharing typically means more critical sections and hence more serialization. The second observation is that the maximum parallelism that the IB-stack can provide is 3.6-fold when all resources are dedicated per thread and 3.1-fold when the context and CQ are shared between all the threads. Most MPI implementations are increasingly moving toward more shared resources (i.e., closer to `QPs only`) in order to manage the per-process resource usage. Thus, in current MPI implementations, 3.1-fold improvement is the maximum benefit that we can expect even in the ideal case.

In MT-MPI, each QP is managed by a single thread; multiple QPs might be managed by a single thread, but a single QP is never managed by multiple threads. This strategy minimizes the mutexes that the IB stack needs to do. We also ensure that the number of threads used for parallelism

is never more than the number of QPs, in order to minimize thread synchronization overheads.

We note that in the MPICH IB netmod, small-message communication employs temporary buffers that are preregistered with the network. Since the network can communicate only to/from preregistered buffers, user data needs to be copied into these buffers on the sender side and out of these buffers on the receiver side. Each connection uses a separate QP and preregistered buffers, so the data copies on the send and receive side are also part of the parallelism and are executed concurrently by different threads.

Despite the potential for parallelism on many-core architectures, several factors limit the practical parallelism achievable in the MPI implementation. For example, in order to achieve the best parallelism, the MPI implementation can benefit from a large number of operations to be issued to the network, which can be evenly shared between the available threads. However, such ideal conditions are hampered by several practical restrictions in current IB network stacks and applications. For example, the number of operations that can be issued to a QP or to the shared CQ is limited. While the QP or CQ can be configured to allow for a large number of operations, such configuration causes (sometimes large) performance degradation due to the internal bookkeeping associated with these data structures within the IB stack. Consequently, the MPICH IB netmod configures this limit to 1,024 for QPs and 32,768 for CQs, thus forcing the maximum number of network operations each thread can post to 1,024 and the maximum number of network operations posted across all threads to 32,768, before thread synchronization is needed. A similar parallelism-limiting factor is the number of preregistered buffers available at the sender and receiver side.

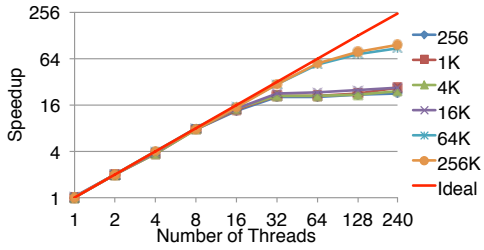
Still another parallelism constraint comes from the application characteristics. Specifically, since in MT-MPI we exploit parallelism at the granularity of a QP, for ideal parallelism we need the same amount of work per QP—a process should have close to uniform communication with its peer processes. In practice, however, this assumption does not hold; indeed, the amount of communication can vary dramatically between different processes, thus limiting the available parallelism.

4. EVALUATION AND ANALYSIS

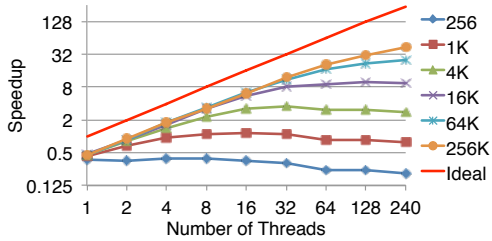
In this section, we evaluate the various techniques designed within MT-MPI. All our experiments are executed on the Stampede supercomputer at the Texas Advanced Computing Center (https://www.tacc.utexas.edu/stampede/). Stampede consists of 6400 Dell Zeus C8220z compute nodes, each with two Xeon E5-2680 processors and 32 GB RAM, and an Intel Xeon Phi SE10P coprocessor with 8 GB of onboard RAM connected by an x16 PCIe 2.0 interconnect. The nodes are interconnected by a Mellanox FDR InfiniBand network. All our experiments are executed on the Xeon Phi coprocessor, with every MPI process running on a separate coprocessor.

4.1 Derived Datatype Processing

In this section, we describe three types of experiments that stress derived datatype processing to various degrees: (1) derived datatype packing performance, (2) halo data exchange with derived datatypes, and (3) the NAS multigrid benchmark.



(a) Packing the top surface with varying Z dimension.



(b) Packing the left surface with varying Y dimension.

Figure 7: Performance of parallel 3D packing.

4.1.1 *Derived Datatype Packing*

In our experiments with derived datatype packing (using `MPI_PACK`), we utilized a 3D matrix of doubles, with the X dimension as the leading dimension. The matrix volume was fixed at 1 GB, so increasing one dimension would reduce another. Our experiments involved packing different 2D planes of the 3D matrix.

Figure 7(a) shows the performance improvement while packing the top surface (X-Z plane). A vector datatype is utilized in this case, with a block length equal to the length of the X dimension and stride equal to the area of the X-Y plane; the Z dimension indicates the vector count. In our experiment, the Y dimension was fixed to 2 doubles, and the Z dimension varied as indicated on the graph legend (X dimension was varied to maintain the matrix volume). As can be seen in the figure, MT-MPI gets a reasonably good speedup with increasing number of threads, achieving a 96-fold improvement compared with the original sequential version when all 240 threads are used. A larger Z dimension provides better speedup because that leads to a larger iteration count for the contiguous copies and hence more parallelism that can be exploited by MT-MPI.

Figure 7(b) shows the performance improvement while packing the left surface (Y-Z plane). A two-level datatype comprising a vector of vectors is utilized in this experiment. The X dimension was fixed to 2 doubles, and the Y dimension varied as indicated on the graph legend (the Z dimension was varied to maintain the matrix volume). As shown in the figure, MT-MPI still achieves a relatively good speedup compared with the sequential version (42-fold), although less than what it achieved while packing the top surface. This reduction in performance is because the lowest-level vector datatype always has a block length of one double and a count equal to the Y dimension. This restricts the amount of work that is done within each iteration of the contiguous data copy operation and consequently limits the work done by each thread, especially when the number of iterations (i.e., the Y dimension) is small. Furthermore, when the Y dimension is small, the parallel version is worse than the original sequential version (speedup is less than 1) because of the compiler's

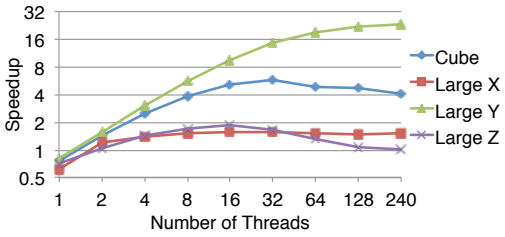


Figure 8: 3D internode halo exchange using 64 MPI processes.

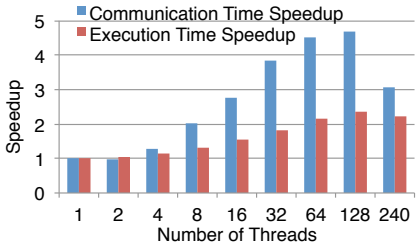


Figure 9: Hybrid MPI+OpenMP NAS MG Class E using 64 MPI processes.

inefficiency in cache prefetching for vectorized code, as described in Section 3.2.1.

4.1.2 Halo Exchange of Data

In our second set of experiments, we measured the performance of 3D halo exchanges of data as used in stencil computations. Both the data and the processes are partitioned into a 3D space. Each process communicates with its neighboring processes with which it shares a plane. For our experiments we define the following four dimension shapes for the local data on each process: (1) `Cube`, with dimensions 512 × 512 × 512 (doubles); (2) `Large X`, with dimensions 16K × 128 × 64; (3) `Large Y`, with dimensions 64 × 16K × 128; and (4) `Large Z`, with dimensions 64 × 128 × 16K. The MPI processes are evenly distributed in all dimensions.

Figure 8 shows the performance improvement achieved by MT-MPI compared with the sequential version when using 64 MPI processes. `Large Y` performs much better than the others, delivering a 23-fold speedup with 240 threads. To understand this behavior, we profiled the communication time for the different dimensions. The halo benchmark sends data in all dimensions simultaneously, so it is hard to profile how much time each dimension takes. Therefore, for profiling purposes, we modified it to serialize communication in one dimension at a time, and we observed that communication along the Y-Z dimension takes 85% of the time. While this is obviously not entirely indicative of the true halo benchmark that sends data in all dimensions simultaneously, it does give us some idea of the communication cost.

As demonstrated in Figure 7(b), a large Y dimension helps improve the performance of packing in the Y-Z dimension by providing better parallelism. This results in a large Y impacting the performance of the halo benchmark to the largest extent. With `Cube`, the Y-dimension is reduced to 512 doubles, thus reducing the speedup to around 5.8-fold as well. With `Large X` and `Large Z`, the Y-dimension further reduces to 128 doubles, which in turn reduces the overall speedup to around 1.6-fold and 1.8-fold, respectively.

4.1.3 NAS Multigrid Benchmark

We also evaluated a hybrid MPI+OpenMP version of the NAS Multigrid (MG) kernel [1] . The original MG kernel distributed as a part of the NAS parallel benchmarks does not contain a hybrid MPI+OpenMP version, so we modified the MPI version to (1) parallelize the local computation using OpenMP and (2) employ derived datatype communication instead of manual packing. The MG kernel implements a V-cycle multigrid algorithm to solve a 3D discrete Poisson equation. In every iteration of the V-cycle routine, halo exchanges are performed with various dimension sizes (count of double), from 2 to 514 in class E with 64 MPI processes, and so forth. The communication in all dimensions except the X-Y plane is noncontiguous.

Figure 9 presents the speedup achieved by MT-MPI compared with the original MPICH in class E (X, Y, Z dimension sizes are each 2K) when employing 64 processes. As shown in the figure, MT-MPI helps improve the communication of MG by 4.7-fold, and the overall execution time by 2.2-fold. The speedup in the communication time is still slightly lower than that of the 3D halo exchanges with the `Cube` shape shown in Figure 8. The reason is that the MG also contains some halo exchanges with very small dimension size whose packing process cannot be parallelized efficiently.

4.2 Shared-Memory Communication

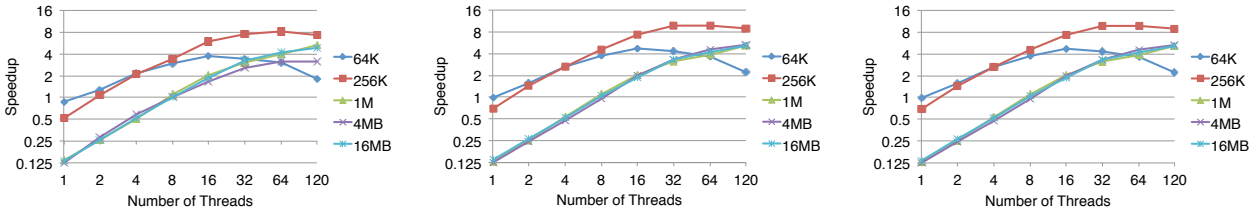
To measure the impact of MT-MPI on intranode shared-memory communication, we evaluated the point-to-point communication benchmarks in the OSU MPI microbenchmark suite version 4.1 (http://mvapich.cse.ohio-state.edu/benchmarks/). In particular, we used the latency, bandwidth, and message rate benchmarks. Both the original MPICH and MT-MPI use an internal shared-memory region of 2 MB, with each cell containing 32 KB.

Figure 10 illustrates the performance of all three benchmarks; the legends in the graph represent different message sizes. We notice that the performance trends of all three benchmarks are similar, with MT-MPI delivering up to a 5-fold performance benefit for message sizes ≥ 1 MB, given enough parallelism. When the number of idle threads is ≤ 4, however, MT-MPI's performance is worse than that of the original MPICH. As discussed in Section 3.2.2, the reason is that MT-MPI loses some of the pipelining capabilities in the original MPICH code in return for thread parallelism. But with a small number of threads, this tradeoff is not beneficial.

Another observation we make in Figure 10 is that the speedup of MT-MPI for message sizes 64 KB and 256 KB is much better than that of other message sizes. This, however, is not because of MT-MPI's superior architecture. Rather, it is because the communication protocol thresholds (i.e., eager vs. rendezvous communication thresholds) in MPICH are tuned for regular Xeon systems, by default, and are too large for the Xeon Phi architecture. We did not change the default configuration of MPICH in order to avoid introducing yet another dimension of variance in the paper. Thus, for 64 KB and 256 KB message sizes, the original MPICH ends up using a suboptimal communication protocol, resulting in MT-MPI's performance falsely appearing to be significantly better as compared to other message sizes.

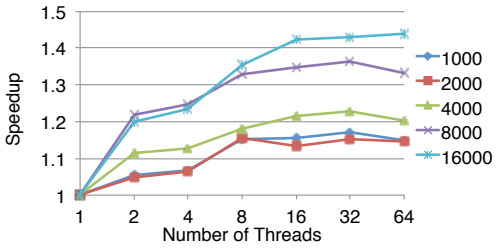
4.3 InfiniBand Communication Operations

In this section we evaluate the performance benefits achieved by MT-MPI with our modifications to the MPICH IB netmod. We performed two types of experiments: (1) a one-sided communication microbenchmark designed to demon-

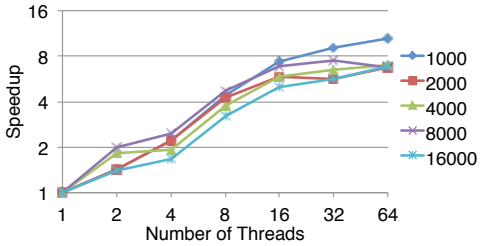


(a) Latency. (b) Bandwidth. (c) Message rate.

Figure 10: Shared-memory communication performance with varying message size between 2 MPI processes



(a) Overall Speedup



(b) Send Processing Speedup

Figure 11: One-sided communication benchmark with IB using 65 MPI processes.

strate the ideal parallelism that can be obtained within MT-MPI and (2) the one-sided version of Graph500 benchmark [11].

4.3.1 One-Sided Microbenchmark

We designed a microbenchmark in which one MPI process issues many `MPI_PUT` operations to all other processes. Each `MPI_PUT` operation is for 64 bytes. We measured the execution time of the benchmark using 65 MPI processes; thus each process communicates with 64 other processes and internally maintains 64 IB QPs. Figure 11(a) shows the speedup in execution time with MT-MPI compared with the original MPICH. As we increase the number of operations issued from 1,000 to 16,000, MT-MPI delivers an increasing performance benefit, reaching a 1.44-fold speedup when using 64 threads.

This performance benefit, however, is less than the ideal speedup of 3.1-fold that we can get by parallelizing IB communication, as discussed in Section 3.2.3. To understand the reason for this less-than-ideal speedup, we measured the execution time of the netmod send-side communication processing at the root process (SP), which consists only of the copy from the user buffer to a preregistered chunk and the posting of the operations to the IB network. Figure 11(b) shows that the execution time of SP delivers around 8-fold speedup when using 64 threads, which is as expected—we expect around a 3.1-fold speedup due to the parallelization in the posting of network operations, and some additional improvement due to the parallelized memory copy. Table 1 shows the relationship between the time spent in SP and the total execution time when issuing 16,000 operations. Although SP shows the expected performance improvement with MT-MPI, the percentage of time spent in SP is less than 10% when using more than 16 threads. This results in a reduction in the overall performance boost that we achieve.

Table 1: Profile of the one-sided communication benchmark.

Nthreads	Execution Time			Speedup	
	Total (s)	SP (s)	SP / Total (%)	Total	SP
1	5.8	2.2	38	1	1
4	4.7	1.3	27	1.2	1.7
16	4.0	0.4	10	1.4	5.0
64	4.0	0.3	8	1.4	6.9

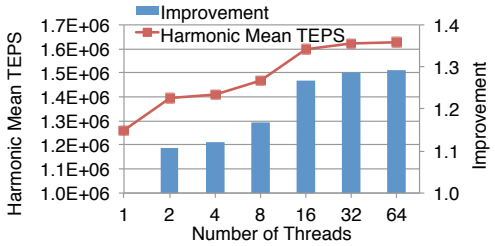


Figure 12: Performance of the Graph500 benchmark using 64 MPI processes.

4.3.2 Graph500 Benchmark

The second benchmark we studied was the Graph500 benchmark [11], which performs a breadth-first vertex-visit operation on large graphs. In particular, we used a scale of 2^{22} and an edge factor of 16 on 64 MPI processes running on different Intel Xeon Phi coprocessors at different nodes. In the one-sided version of the Graph500 benchmark, every process issues many `MPI_Accumulate` operations to the other processes in every breadth-first search iteration.

Figure 12 shows the performance improvement of MT-MPI compared with the original MPICH. MT-MPI delivers a 1.3-fold improvement in the harmonic mean of the traversed edges per second (TEPS) when using 64 threads. As expected, this improvement is on par with the performance improvement we see in the one-sided communication benchmark that we discussed in Section 4.3.1. The slightly smaller speedup compared with the one-sided communication benchmark (which achieves a 1.44-fold speedup) is because the Graph500 benchmark does not uniformly communicate with all peer processes, thus causing some unevenness in MT-MPI's parallelization.

5. RELATED WORK

The hybrid MPI+OpenMP programming model has been extensively used and studied in the past. For instance, Lusk

and Chan [10] explored the performance of such a model on a typical Linux cluster, a large-scale system from SiCortex, and an IBM Blue Gene/P system. The authors concluded that some applications performed better with several MPI-only processes on the same node, while others could benefit from the hybridization. While this situation is still true today, an increasing number of applications are moving to hybrid MPI+OpenMP models, not just for performance, but for per-core resource limitations (in particular, memory). Other studies [16] have, on the other hand, reported satisfactory results in porting the finite-difference time-domain algorithm to the hybrid paradigm to adapt it to SMP compute nodes.

Smith and Kent [15] also found that increasing the number of threads decreased the efficiency of the code when implementing the quantum Monte Carlo algorithm on mixed OpenMP/MPI code on an SGI Origin 2000 system. Although this phenomenon was not attributed to the idle-threads issue we address in this paper, it certainly contributes to the reduced efficiency per thread. [18] performed a comprehensive evaluation of multithreaded MPI communications, pointing to the mutually exclusive regions involved in communication as one of the reasons for the suboptimal performance obtained.

Several researchers have also looked at optimizing the MPI implementation in multithreaded environments. For example, the authors of [2, 7, 8] proposed various techniques to minimize locking within the MPI implementation in order to improve the performance of MPI in `MPI_THREAD_MULTIPLE` environments. They presented various techniques to improve performance on traditional Linux clusters as well as the IBM Blue Gene/P systems. The authors in [6] proposed extensions to the MPI standard that would allow the MPI implementation to minimize contention and improve performance in some cases. However, all these optimizations are for `MPI_THREAD_MULTIPLE` applications. A large fraction of today's hybrid MPI applications, however, still use `MPI_THREAD_FUNNELED` and `MPI_THREAD_SERIALIZED` modes, for which these optimizations are not helpful.

6. CONCLUSIONS

In this paper we analyzed the potential benefits of employing idle hardware threads to accelerate MPI processing in hybrid MPI+OpenMP applications on massively parallel many-core architectures. To this end, we modified two widely deployed implementations of these models: the Intel OpenMP runtime and MPICH MPI implementation. We described various optimizations in different parts of MPI implementation, including derived datatype processing, shared-memory communication, and IB network operations. Our experimental evaluation, based on several micro- and macro-kernels, demonstrates considerable performance benefits.

Acknowledgments

This work was financially supported by (1) the CREST project of the Japan Science and Technology Agency (JST) and the National Project of MEXT called Feasibility Study on Advanced and Efficient Latency Core Architecture and (2) the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, under Contract DE-AC02-06CH11357. The experimental resources for this paper were provided by the Texas Advanced Supercomputing Center (TACC) on the Stampede supercomputer.

References

[1] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 1991.

[2] P. Balaji, D. T. Buntinas, D. J. Goodell, W. D. Gropp, and R. S. Thakur. Toward Efficient Support for Multithreaded MPI Communication. In *Euro PVM/MPI*, 2008.

[3] D. Buntinas and G. Mercier. Implementation and Shared-Memory Evaluation of MPICH2 over the Nemesis Communication Subsystem. In *Euro PVM/MPI*, 2006.

[4] G. Chrysos. Intel Xeon Phi Coprocessor - The Architecture. White paper, Intel Corporation, Sept. 2012.

[5] L. Dagum and R. Menon. OpenMP: An Industry Standard API for Shared-Memory Programming. *IEEE Computational Science & Engineering*, 5(1):46–55, 1998.

[6] J. S. Dinan, P. Balaji, D. J. Goodell, D. Miller, M. Snir, and R. S. Thakur. Enabling MPI Interoperability Through Flexible Communication Endpoints. In *Euro MPI*, 2013.

[7] G. Dozsa, S. Kumar, P. Balaji, D. T. Buntinas, D. J. Goodell, W. D. Gropp, J. Ratterman, and R. S. Thakur. Enabling Concurrent Multithreaded MPI Communication on Multicore Petascale Systems. In *Euro MPI*, 2010.

[8] D. J. Goodell, P. Balaji, D. T. Buntinas, G. Dozsa, W. D. Gropp, S. Kumar, B. R. de Supinski, and R. S. Thakur. Minimizing MPI Resource Contention in Multithreaded Multicore Environments. In *IEEE Cluster*, 2010.

[9] Intel Corporation. Many Integrated Core (MIC) Architecture — Advanced. `http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html`, 2013.

[10] E. Lusk and A. Chan. Early Experiments with the OpenMP/MPI Hybrid Programming Model. In *OpenMP in a New Era of Parallelism*, pages 36–47. Springer, 2008.

[11] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang. Introducing the Graph 500. In *Proceedings of the Cray User's Group Meeting (CUG)*, May 2010.

[12] S. Olivier, A. Porterfield, K. Wheeler, M. Spiegel, and J. Prins. OpenMP Task Scheduling Strategies for Multicore NUMA Systems. *The International Journal of High Performance Computing Applications*, (26(2)):110–124, May 2012.

[13] OpenFabrics Alliance. OpenFabrics Alliance. `http://www.openfabrics.org`, 2013.

[14] R. Ross and N. Miller. Implementing Fast and Reusable Datatype Processing. In *In EuroPVM/MPI*, pages 404–413. Springer Verlag, 2003.

[15] L. Smith and P. Kent. Development and Performance of a Mixed OpenMP/MPI Quantum Monte Carlo Code. *Concurrency Practice and Experience*, 12(12):1121–1129, 2000.

[16] M. F. Su, I. El-Kady, D. A. Bader, and S.-Y. Lin. A Novel FDTD Application Featuring OpenMP-MPI Hybrid Parallelization. In *International Conference on Parallel Processing (ICPP)*, pages 373–379. IEEE, 2004.

[17] M. Takagi, Y. Nakamura, A. Hori, B. Gerofi, and Y. Ishikawa. Revisiting Rendezvous Protocols in the Context of RDMA-capable Host Channel Adapters and Many-core Processors. In *Euro MPI*, 2013.

[18] R. Thakur and W. Gropp. Test Suite for Evaluating Performance of Multithreaded MPI Communication. *Parallel Computing*, 35(12):608–617, 2009.

[19] A. J. Wallcraft and D. R. Moore. The NRL Layered Ocean Model. *Parallel Computing*, 23(14):2227 – 2242, 1997. Parallel computing in regional weather modeling.