2018 IEEE International Parallel and Distributed Processing Symposium Workshops

Parallelizing MPI using Tasks for Hybrid Programming Models

Surabhi Jain, Gengbin Zheng, Maria Garzaran, James H. Cownie, Taru Doodi, Terry L. Wilmarth
Intel Corporation
Email: {surabhi.jain, gengbin.zheng, maria.garzaran, james.h.cownie, taru.doodi, terry.l.wilmarth}@intel.com

***Abstract*—Many-core architectures such as the Intel® Xeon Phi$^{\text{TM}}$ provide dozens of cores and hundreds of hardware threads. For these machines, a basic MPI implementation is inefficient, as it does not take advantage of the shared data across the ranks on the same node. A hybrid-programming model called MPI+X, where MPI is used between the nodes and a threading model X (Pthreads or OpenMP*, among others) can better utilize the compute and memory resources.**

In this paper, we investigate how the MPI library can run in parallel without competing with the threads of X that are running the application, and in particular without oversubscribing the available hardware resources. We assess whether MPI can use tasks that can be executed by idle threads in X. To prototype such a system, we have chosen X to be OpenMP. Our experimental results show that parallelizing MPI calls using OpenMP tasks can provide significant speedups.

I. Introduction

Many-core architectures such as the Intel® Xeon Phi$^{\text{TM}}$ processor provide dozens of cores and hundreds of hardware threads per chip. For these many-core machines, using only MPI parallelism is inefficient, as it does not take advantage of the shared data across the ranks on the same node, requiring message exchanges across ranks in the same chip. For these systems, a hybrid-programming model can better utilize the compute and memory resources. This model is usually called "MPI+X", where MPI is used between the nodes and the threading model "X" (X can be Pthreads, OpenMP* or Intel® Threading Building Blocks (TBB), among others) is used across the cores in a node. Applications running in such a programming model use several threads to execute the computation in parallel, but usually only one of the threads calls the MPI library. Thus, the MPI library runs serially, while many threads are idle. This is especially true if the MPI call is made from the serial part of the application, but it might also be true when the MPI call is made from the parallel region. Furthermore, the MPI implementation itself typically is sequential. In a many-core machine, this poor use of the resources can result in a significant number of idle threads.

In this paper, we investigate how to use threads of X to parallelize an MPI implementation without competing with the threads of X that are running the application, and in particular without oversubscribing the available hardware resources. Similar work was done by Si et al [1]. In their study, they implemented a multi-threaded MPI using OpenMP and tried to exploit idle threads to speedup the MPI calls. Their approach relies on a modified OpenMP runtime that provides information about the number of idle threads. However, it is challenging to gather that information in the general case where the user code is using tasks. This is because a thread that is in a barrier is not guaranteed to be idle since it could be executing OpenMP tasks. Furthermore, this approach may lead to oversubscription when the OpenMP application uses nested parallelism. These problems motivated us to look for parallelization methods that do not require creating new threads. One such method is to use tasks. Tasks are found in OpenMP and TBB and can be executed by the thread that creates them or by other threads in "X", at least at task-scheduling points, such as barriers.

In this paper, we investigate whether task parallelism can be used to parallelize MPI. To prototype such a system, we have chosen X to be OpenMP and have consequently used OpenMP tasks within the MPI library. We show experimental results for parallel execution of intranode *MPI_Send()*, *MPI_Pack()*, and communication using derived datatypes. Our experimental results show that when the amount of work to be performed inside the MPI library is large enough, parallelizing MPI calls with OpenMP tasks can result in significant speedups. Applications in new domains such as Machine Learning and Deep Learning communicate large messages, imposing a great challenge to the underlying MPI communication library. Therefore, it is critical to optimize the MPI library for large messages on the many-core architectures.

The paper is organized as follows. Section II presents an overview of our approach; Section III discusses our design and implementation; Section IV presents our environmental setup; Section V shows our experimental results; Sections VI discusses related work; Finally, Section VII concludes.

II. Overview

The goal of this work is to consider MPI parallelization approaches that avoid oversubscription. In this Section, we compare two possible thread execution models to achieve this. In the first approach, given a system with N threads, we reserve X threads for the MPI library. In this model, when a thread calls the MPI library, this will run in parallel using the X threads reserved for the use of the MPI library. Assuming that the application uses $N-X$ (or fewer) threads, the system will not be oversubscribed. An example of such approach is shown in Figure 1(a), where $N=6$ and $X=2$. This approach, however, does not use the resources efficiently. First, when the MPI library is called from the serial part of the application, the MPI library is run using only two threads, when all the six other threads could have been used. Even when the MPI library is called from a parallel region, more than two threads could be used to execute the MPI call if other application threads are idle, for instance, when the application has load imbalance. The example in Figure 1(a) shows a case like this, where the first two threads have finished earlier and are waiting for the other threads to complete and arrive at the barrier. These two idle threads could also help in the execution of the MPI call. A similar approach is proposed in [2], where they offload MPI processing to a dedicated thread.

Figure 1(b) shows the other thread execution model. In this case, the application uses all the threads, $N = 6$, to execute the compute part in parallel. When the MPI library is called from a serial region, all the threads participate in the execution of the MPI call. When the MPI library call is made from the parallel section, the application thread that called the MPI library and all idle threads (for instance, waiting at a barrier) help in the execution of the MPI call. As threads complete the execution of the compute part and reach the barrier, if there is work created by the MPI call (in the form of tasks) that has not yet been executed, the newly idle threads can steal it and thereby contribute to the execution of the MPI call. In the worst case, when there are no idle threads, only the thread that called the MPI library executes the tasks. As long as the overhead of creating tasks is small, this is no different from the current sequential MPI implementation, as the MPI routines are executed sequentially. Thus, the approach shown in Figure 1(b) is more desirable than that of Figure 1(a), as it results in the best utilization of the resources, where all the threads are used during the compute part of the application and help to execute the MPI library routines when they are idle. To demonstrate this task-based approach, we chose to parallelize several MPI functions using OpenMP tasks. Thus, we assume X to be OpenMP.

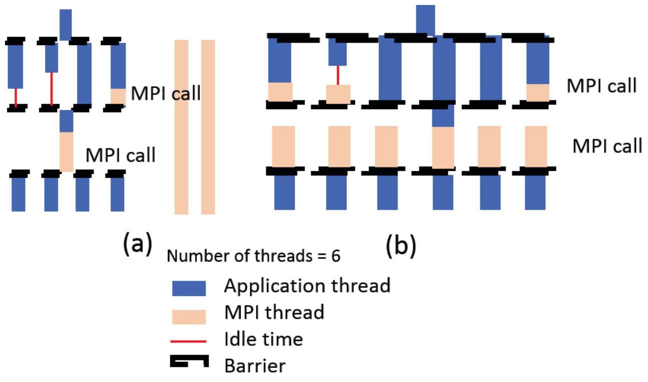

Fig. 1: Ways to use MPI and application threads

The MPI standard describes four levels of thread support. Programs running with *MPI_THREAD_SINGLE* do not take advantage of hybrid programming models, but they can benefit from parallelism within the MPI runtime when the application is not using all the threads. This is not a very interesting case for our study, as the benefit of our proposed solution is that the MPI runtime can take advantage of application threads that are idle (rather than unused threads). Programs written with threading models FUNNELED, SERIALIZED, or *MPI_THREAD_MULTIPLE* use several threads to run the application. With models FUNNELED and SERIALIZED only one thread at a time can call the MPI library, while with *MPI_THREAD_MULTIPLE*, several threads can call the MPI library concurrently. With these three models, since the application is already running in parallel, parallelizing the MPI runtime using threads can easily result in oversubscription of the hardware resources. With our proposal, this issue is avoided by using OpenMP tasks, as we do not create additional threads. Currently, most of the applications use FUNNELED or SERIALIZED threading models, due to the poor implementation of *MPI_THREAD_MULTIPLE*.

Notice that the approach described in this paper is orthogonal to the parallelism that comes from several threads running in parallel and concurrently calling the MPI runtime using *MPI_THREAD_MULTIPLE*. Our proposal is about parallelizing each MPI call, by parallelizing the MPI runtime itself. So, while we create nested parallelism, we never oversubscribe.

III. Design and Implementation

In this section, we explain the design and implementation details of our task-based parallelization approach. We consider our solution opportunistic because it makes use of idle threads. If all the threads are busy, the thread creating the tasks ends up executing all the tasks. This type of parallelism may help in applications that use a synchronizing model such as OpenMP which potentially generates idle threads at synchronization points that can be exploited by MPI. Figure 2 shows pseudo-code of how an MPI call using OpenMP tasks may be implemented. We first check if the MPI call is made from a sequential or from a parallel region in the application. In the case of sequential, we create a parallel region and one of the threads creates tasks. In the case of parallel region, the calling thread creates tasks and waits at *#pragma omp taskwait* until all the tasks have been executed either by itself or other idle threads. Our proposed solution is to use OpenMP tasks which can then naturally be executed by the OpenMP runtime on threads which are waiting at task-scheduling points (these task scheduling points include implicit or explicit barriers, the point of encountering a taskwait construct, and the completion point of a task). As previously mentioned, this approach has a number of advantages: i) it uses only standard OpenMP features so is portable to any OpenMP 3.0 (or later) implementation; ii) it does not cause over-subscription and therefore kernel scheduling; iii) it does not require a modified OpenMP library. iv) it naturally exploits idle OpenMP threads without requiring intrusive changes.

Next, we show how we can use this task-based approach for parallelizing MPI routines. In particular, we demonstrate this approach in parallelizing memory copying for intranode communication, explicit and implicit packing and unpacking of non-contiguous datatypes (e.g. *MPI_Pack()* and

```
if (omp_in_parallel())
{
 //Create tasks for whatever MPI wants to do in parallel
 //which will run on idle pre-existing OpenMP threads

 #pragma omp taskwait
 //All tasks we created have completed when we get here
}
else
{//No pre-existing parallelism so create some
 #pragma omp parallel
 {
  #pragma omp single nowait
  {
   //Create tasks for what MPI wants to do in parallel
  }
 }
 //All tasks we created have completed when we get here
}
```

Fig. 2: MPI code using OpenMP tasks

MPI_Unpack()). Notice that these MPI calls have been chosen to demonstrate the benefit of this approach, but other MPI calls can also potentially benefit from it. The main goal of this work is not to parallelize a particular MPI call, but to show that it is possible to parallelize MPI itself using tasks to exploit idle threads in multithreaded hybrid programming model and achieve significant speedups without creating oversubscription.

A. *Shared Memory Communication*

Shared memory communication or intranode communication takes place when different MPI ranks running on the same node need to communicate, for instance using *MPI_Send()*. Since each rank has a different address space, *MPI_Send()* between two MPI ranks is implemented as a shared memory copy. MPICH implements this copy using a pipelined double-copy approach [3] that uses a shared memory circular buffer divided into cells so that the sender can copy the data to an empty cell while the receiver is copying from the cell when it is full or a complete message is put in.

When the amount of data to copy is large, we expect parallel copying can provide significant performance improvement. Parallelism in this case can be implemented at two different levels. One is intra-level parallelism, where we copy each cell in parallel. However, in the original MPICH implementation, each cell is only of 32KB, which makes it difficult to gain any performance from parallel copying. The other scheme is to use multiple threads to copy different cells concurrently. The problem with this scheme is that it eliminates the pipeline parallelism due to concurrent copying, and copying may finish out-of-order. Therefore, we implemented a mixed strategy that uses OpenMP tasks to copy each cell and that retains the existing pipeline parallelism in MPICH. Figure 3 shows this approach. In the left side of the figure, we show the sequential case currently implemented in MPICH, where 4 cells are copied in a pipelined fashion[1]. In the right side of the figure 3, we show our approach, where each cell is copied in parallel using two tasks. In our parallel implementation, we use a variable cell size, where the cell size depends on the number of threads the application is using and the size of each task. Thus, $cellsize = tasksize \times nthreads$. The optimal task size depends on the target platform. Profile information needs to be collected to determine the task size, so that task creation overhead is kept to a low percentage.

[1]Notice that the MPICH implementation uses a total of 8 cells, where each cell is 32KB

As discussed in [1], increasing the cell size can hurt the performance of data copying. This is because the receiver cannot start the copying out until the complete cell has been copied, reducing the overlap between the sender and receiver. Also, when the number of threads is small, a large cell can result in slowdown, as the amount of parallelism is not enough to compensate the reduced overlap between the sender and receiver. Thus, in general, the cell size needs to be large enough to provide enough parallelism for the threads, while avoiding significant task creation overhead.

Our experimental results in Section V confirm this issue. When the number of threads is small, the parallel approach can result in slowdowns. In the general case when the MPI runtime is called from a parallel region in the application, there is no reliable way for the OpenMP runtime to determine the number of idle threads because idle threads are free to execute tasks if there are any. Thus, in our current implementation, parallel copying is only enabled when the MPI call is done from the sequential region of the application and if the thresholds in number of the available threads and message size are met. Figure 4 shows the pseudo-code to perform the memory copy using tasks when *MPI_Send()* is called from the serial region. This particular code creates one task per thread. If the number of threads is less than a certain threshold (th1) or the size of each task is small (th2), the memory copy is performed sequentially; otherwise, one of the threads creates tasks that can potentially be executed by all the threads in the OpenMP runtime.

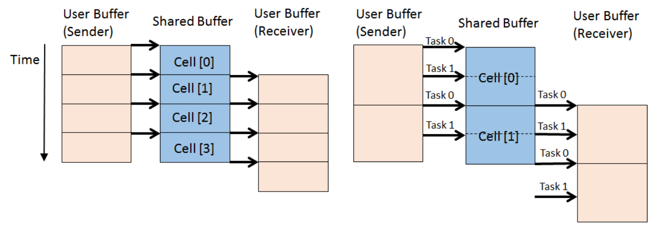

Fig. 3: Sequential and Parallel data copying

B. *Derived Datatype Packing and Unpacking*

MPI Derived datatypes, like vector, indexed, and struct, allow the application to specify non-contiguous data in a convenient manner [4] [5]. Derived datatype packing/unpacking is done in MPI in two scenarios: (1) when *MPI_Pack/MPI_Unpack* is called explicitly, and (2) when communicating a non-contiguous datatype [4] [5]. In both cases, the MPI library calls the internal function *Segment_pack/Segment_unpack* to pack non-contiguous data into

```
int num=omp_get_max_threads();
size_t chunksize=len/num;
if (num < th1 || chunksize < th2)
  memcpy(dst,src,len);
else
{
#pragma omp parallel
 {
  #pragma omp single nowait
   {
    for(i=0; i<num; i++){
     #pragma omp task firstprivate(i)
      memcpy(dst+i*chunksize,src+i*chunksize,chunksize);
    }
    if(len%num != 0)
     memcpy(dst+i*chunksize, src+i*chunksize, len%num);
   }
 }
}
```

Fig. 4: Memory copy using OpenMP tasks when MPI_Send() is called from the serial region

a contiguous buffer or to unpack a contiguous buffer to a non-contiguous memory region.

To process the derived datatype, a typical implementation recursively traverses the derived datatype tree and issues a local memory copy that copies the non-contiguous trunk of data from/to the buffer. MPICH [6] optimizes this process to avoid the recursive traversal by representing the entire derived datatype as a stack structure and iteratively walks through the stack in a *for* loop [7].

1) Vector Datatype: In this section, we discuss how to use OpenMP tasks to parallelize the packing/unpacking of a vector datatype, when it is at the lowest level of a derived datatype tree. The next section discusses a different parallelization scheme for a nested datatype at a higher level of the tree.

A vector is specified by a number of blocks, a number of elements in each block, and the block stride. Figure 5 shows the code that we have used to parallelize the *MPI_Pack()* call. This code first checks if the call is done from a sequential or from a parallel region. If the call is from the sequential region of the application, it first creates a parallel region. One of the threads creates tasks (*#pragma omp single* region), while the others wait at the implicit barrier at the end of the *#pragma omp parallel*. The tasks will be executed by all the threads, the one that created them and all the idle threads waiting at the barrier. If the call is made from the parallel region of the application, then the calling thread is the one that creates tasks. These tasks can be executed by the thread that created them and by other threads waiting at task synchronization points, such as barriers. In the worst case, no thread in the system is idle, in which case, only the thread that called *MPI_Pack()* will execute the tasks. Our experimental results show that even in this case, the overhead is small if the task size is appropriately selected. Notice that *#pragma omp taskwait* in Figure 5 is a synchronization construct and forces the thread that created the tasks to wait until all its child tasks have been completed.

We have observed that when the MPI call is made from a sequential region, having several threads creating tasks delivers

```
if (!omp_in_parallel()){
 #pragma omp parallel
  {
   #pragma omp single nowait
    create_MPI_Pack_tasks(..);
  }// Other threads wait here and
} // Execute tasks if idle
else{
 create_MPI_Pack_tasks(..);
 #pragma omp taskwait
 //At this point all tasks have finished
}
void create_MPI_Pack_tasks(...){
 int num_tasks= total_count/task_size;
 for (tsk_id=0; tsk_id<num_tasks; tsk_id++){
  int begin = tsk_id*(total_count/num_tasks);
  int end = (tsk_id+1)*(total_count/num_tasks);
  #pragma omp task firstprivate(begin,end)
  {
   int i, c, j_iter;
   for(i=begin; i<end; i++){
    c=i/nelms*l_stride;
    for (j_iter=0; j_iter<nelms; j_iter++)
     l_dest[i+j_iter] = l_src[j_iter+c);
   }
  }
 }
}
```

Fig. 5: MPI_Pack() using OpenMP tasks

some additional performance, as all the other threads are idle and waiting for the tasks to be created before being able to start execution.

The advantage of parallelizing *MPI_Pack()* using OpenMP tasks versus OpenMP threads appears when the MPI call is made from the parallel region, as in that case the calling thread can create tasks that other threads can execute if they are idle. The tasking approach is also helpful to balance the load dynamically, and as long as the tasks are large enough, they should not add significant overhead. For the sequential region, there is really not much difference between using tasks or threads, as all the threads, except the one calling MPI, are idle. Notice that the approach that uses tasks can never create oversubscription, as it always uses the threads that the application created and never creates additional threads. When using *#pragma omp parallel* in the MPI library, it is possible to create oversubscription when the MPI call is made from inside a parallel region in the application. Whether oversubscription happens or not will depend on the value of the *OMP_NESTED* environment variable, which allows for nested parallelism when set to *TRUE*. In the GCC, LLVM, Intel OpenMP runtime system this variable is set to *FALSE* by default, but the programmer can override that value, in which case oversubscription may occur.

2) Nested Datatype (Vector of Vectors): The same parallelization scheme may not work efficiently for nested datatypes such as a vector of vectors. This is because task partitioning always occurs at the leaf level of the nested datatype, and when the leaf vector is too small to be divided into tasks, the parallelization does not deliver any performance benefit. One possible solution is that instead of partitioning the leaf vector as tasks, we partition along the higher level of vector for tasks, i.e. each task is responsible for packing/unpacking one or more entire leaf vectors.

We have designed and implemented a prototype using this approach in MPICH for vector of vectors datatypes. Currently, we have only implemented this approach for vector of vectors datatypes, but the same idea applies to other nested datatypes.

In MPICH, packing/unpacking a nested datatype is handled in the function *Segment_manipulate()*, which traverses the derived datatype tree using a stack structure in a *for* loop and does a memory copy whenever a leaf level datatype (e.g. vector) is encountered. To parallelize this *for* loop with OpenMP tasks, we transform the *for* loop into an OpenMP parallel region. As the stack is traversed, and a vector leaf datatype is encountered, an OpenMP task is created for packing and unpacking the entire vector. This approach can potentially increase the OpenMP task size. While our experiments showed some encouraging results, we will need to rewrite the MPICH packing function to make this approach more general, which is part of our future work.

IV. Experimental Setup

In this Section, we discuss our environmental setup. Experiments were run on a single or multiple nodes of a Knights Landing (KNL) machine. This is an Intel® Xeon Phi$^{\text{TM}}$ Processor 7210 (1.3 GHz, 64 cores, 4 threads/core, with 32KB L1 data and instruction cache, 1MB L2 cache, 96GB of DDR, and 16GB of MCDRAM), running Linux 3.10.0. Codes are compiled using Intel®Parallel Studio XE Composer Edition for C++ (version 2016.0.109) for both the MPICH libraries and the applications. We use MPICH v3.2b4-98-g4551de1 as a baseline. The KNL is configured to have memory model -Flat, Cluster mode -Quadrant, no SNC [8]. The code is publicly available at https://github.com/jain-surabhi-23/mpich/commit/22f8ed6.

KNL has DDR4 RAM and in package memory called MCDRAM. MCDRAM has a higher bandwidth than DDR4 (using the STREAM benchmarks, we have measured that the memory bandwidth of MCDRAM is 400 GB/s, while the memory bandwidth of DDR4 is 90GB/s). MCDRAM can be configured in different modes (a discussion of these modes is outside the scope of this paper). For the experiments reported in this paper, we have run by placing all the data in DDR4 (using numactl -m 0) and by placing all the data in MCDRAM (using numactl -m 1). Our experimental results show similar trends in both cases, although MCDRAM sometimes delivers higher speedups,although MCDRAM sometimes delivers higher speedups for benchmarks limited by bandwidth. Since the focus of this paper is not on whether to use MCDRAM or DDR, we only focus on one configuration, and we have chosen the one where all the data are placed in MCDRAM.

To adapt to dynamic workloads, we use a set of parameters to determine when to parallelize. Parallelization only occurs if the input to the MPI call is large enough to use tasks. We perform offline tuning on the target hardware to determine these thresholds. The optimal value of *task_size* and *buffer_size* used in case of shared memory communication is also determined specifically for the target hardware.

V. Experimental Results

In this section, we first assess the benefit of our approach using benchmarks, and show the benefit of parallelizing shared memory communication (Section V-A) and *MPI_Pack()* or communication using derived datatypes (Section V-B). We also show experimental results using real applications (Section V-C).

A. Shared Memory Communication

To assess the benefits of shared memory communication we ran experiments with the point to point communication latency benchmark in the OSU MPI benchmark suite with 2 MPI ranks running on a single node. Figure 6 shows speedups for this benchmark with respect to the original MPICH, for different message sizes as the number of threads per rank increases. Original MPICH uses an internal shared memory buffer of 256 KB and cells of 32 KB, but the experimental results in this figure (both original and our parallel implementation) use an intermediate buffer of 4MB. In the parallel implementation $cellsize = tasksize \times nthreads$. We have chosen $tasksize$ to be 32KB. The sizes 4MB and 32KB provide the maximum benefits, as explained in detail later in this section.

As the figure shows, we only parallelize for messages of 64KB or larger, as each task has a size of 32KB. As explained in Section III-B for low number of threads (2 and 4) the parallel approach results in slowdowns with respect to the sequential version. The reason is that the parallel approach increases the cellsize and as a result delays the time when the receiver can start copying out, reducing the overlap between sender and receiver. As the number of threads increases (8 or more), the added parallelism can compensate for the reduced overlap, achieving speedups of almost 2.5X for the largest messages. For messages of size 64KB or 1MB, this approach does not produce speedups. For 64K messages, only two threads can perform the data copying, as the task size is 32KB. For 1M messages, with 16 threads, $cellsize = 32KB * 16 = 512KB$ and only two cells are used. In this case, the receiver can only start the copyout when half of the message has been copied to the intermediate buffer and there are not enough stages in the pipeline (only two cells are used) to compensate for the delay in start of the copyout. It is important to notice that the limitations that we observe in the parallel execution of the shared memory communication are due to the pipeline approach used in the original MPICH to perform the memory copy across ranks. This problem can be avoided by using a different implementation so that each individual sub-cell can be copied out right after it has been copied in to the intermediate buffer without having to wait for the whole cell to be copied. This implementation requires some extra bookkeeping and is left as future work.

We have also evaluated the performance of the memory copying as the task size varies and have found that 32KB is usually a good tradeoff. With a task size of 16KB, the shape of the plot is similar to that of Figure 6, but with lower speedups, with the maximum speedup being 1.7 instead of 2.5. Finally, we have noticed that for messages of size 256KB or larger, having a larger intermediate buffer of 4MB runs faster than with the small buffer in the original MPICH. However, for small messages, a small buffer is better. Our comparison uses 4MB in both cases, so that we only evaluate the impact due to parallelism and not to different buffer sizes.

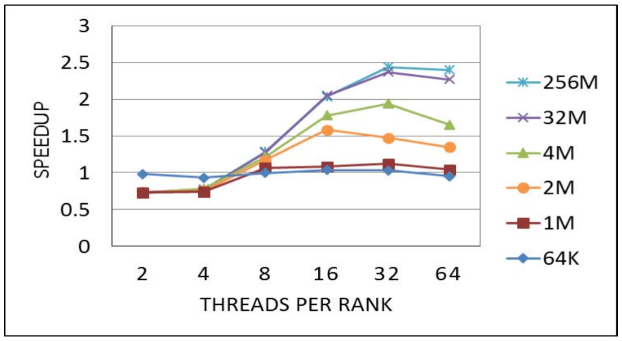

Fig. 6: Speedup of parallel memory copying with 32KB task size

B. *MPI_Pack()*

We have evaluated the performance benefit of parallelizing *MPI_Pack()* when using vector datatype and vector of vectors datatypes. We have also considered whether the call is made from the sequential part or from the parallel part of the application with different threading modes.

1) Vector Datatype: We have evaluated the performance benefit of executing *MPI_Pack()* in parallel with the same benchmark used by Si et al. [1]. This benchmark packs the top 2D (XZ) plane of the 3D matrix. The volume of data in the 3D matrix is 1GB (1024 $\times$ 65536 $\times$ 2 doubles). A pictorial view in Figure 7 shows in green color the top part being packed. This experiment uses a vector datatype, where the number of vectors is determined by the value of the Z dimension, the length of the block is determined by the value of the X dimension, and the stride is determined by the values of the X and Y dimensions, as shown in the Table I in the second row, Axis 2. In the table, the number written within the parenthesis is the value of that field for the cube in Figure 7. We assume the X dimension to be contiguous. Along axis 2 (the communication of ZX plane) the X dimension is contiguous but Z is not. So, the data is represented as a vector datatype. For our experiments, the size of the Y dimension is fixed to 2. We pack the top (XZ plane), of volume 0.5 GB and parallelize over the Z dimension. Thus, one or more rows of Z consisting of X contiguous doubles are packed (or copied) by 1 thread or task to the output buffer.

***MPI_Pack()* is called from sequential region**: The plot in Figure 8 shows the speedups when packing the top surface as we vary the number of blocks (the Z dimension) and

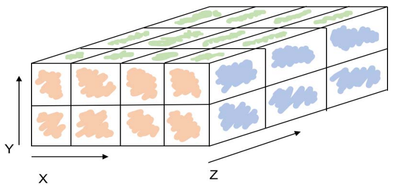

Fig. 7: Pictorial view of the 3D cube

the length of each block (the X dimension), while maintaining the amount of data packed constant (X*Z=0.5GB). For this experiment we used 1 MPI rank. In Figure 8 we vary num_blks along the X-axis, and the Y-axis shows the speedup we obtain over the original sequential MPICH. Different trend lines show the number of threads we used in data packing. We see speedups of up-to 62x when using 64 threads. We also notice that speedups are higher when using 64 or 128 threads, irrespective of the number of blocks. Higher speedups are usually achieved with larger number of blocks, 64K. The speedup with 256K blocks is slightly lower than that with 64K, because with 256K the block size is 256 doubles. This adds overheads due to too many *memcpy()* calls on small sizes.

We also have results using DDR, instead of MCDRAM. In that case, maximum speedup is 8x, which is usually achieved with 32 or 64 threads. Finally, notice that since *MPI_Pack()* is being called from the sequential region, we have the choice to implement the *MPI_Pack()* using the OpenMP *parallel for* or OpenMP tasks, as we know all the threads are idle and oversubscription is not possible. Our experimental results, show that in this case, both approaches deliver the same performance, so we do not show it.

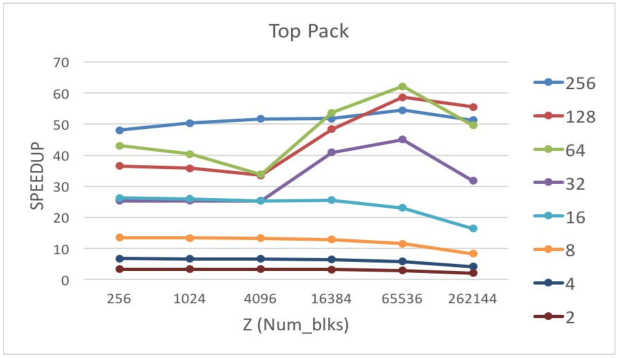

Fig. 8: Results from packing the top surface (MPI_Pack from serial region)

***MPI_Pack()* is called from the parallel region**: We have also evaluated the performance of *MPI_Pack()* with a vector datatype when called from an OpenMP parallel region under a multithreaded environment. For that, we have used the synthetic benchmark shown in Figure 9. In this benchmark, when the program enters the OpenMP parallel region, the threads are divided into three groups depending on the thread ID number: (1) packing threads, which call *MPI_Pack()*; (2) compute

	Dimension	Face	Datatype	Blocklength	Num_blks	Stride	
Axis 1	Z(3)	YX (Orange)	memcopy				
Axis 2	Y(2)	ZX (Green)	Vector	X(4)	Z(3)	X*Y(8)	
Axis 3	X(4)	ZY (Blue)	Vector of Vectors	1	Y(2)	X(4)	Leaf Vector
				Y(2)	Z(3)	X*Y(8)	Top Vector

TABLE I: Datatypes used along each of the three axes on the cube

threads, which perform a certain amount of computation, and (3) idle threads. For this experiment, the MPI threading mode used is *MPI_THREAD_MULTIPLE*. We compare two different parallel implementations of *MPI_Pack()*, one that uses OpenMP tasks as proposed in this paper, and another that uses the OpenMP *parallel for* construct.

This experiment uses nested parallelism as the application is running in parallel while the MPI runtime is trying to run *MPI_Pack()* in parallel. However, by default OpenMP does not enable nested parallelism. Therefore, if *MPI_Pack()* is implemented with a *parallel for* construct, it will run sequentially, as nested parallelism is disabled. To enable nested parallelism, we set the OpenMP environment variable *OMP_NESTED* to *TRUE*. When nested parallelism is enabled, the packing thread that uses the parallel-for inside *MPI_Pack()* creates a new team of threads. This may result in thread oversubscription. The task-based scheme does not introduce an OpenMP *parallel for*, instead, it creates OpenMP tasks to parallelize the *MPI_Pack()*, avoiding thread oversubscription.

```
#pragma omp parallel
{
  thread_id=omp_get_thread_num();
  if (thread_id < 4) {
      call MPI_Pack();
  }
  else if (thread_id < 4 + num_idle_threads) {
      do nothing
  }
  else {
      do_computation
  }
}
```

Fig. 9: Example of MPI_Pack() called from a parallel region

Figure 10 shows the speedup results compared with the original unmodified MPICH in four different scenarios. (1) *MPI_Pack()* using task-based parallelization when nested parallelism is disabled; (2) same as (1) but with nested parallelism enabled; (3) *MPI_Pack()* using parallel for when nested parallelism is disabled; and (4) same as (3) but with nested parallelism enabled. All experiments were run on a KNL single node with a total of 256 OpenMP threads. The number of packing threads is fixed to be 4, and we vary the number of idle threads that can help in the execution of *MPI_Pack()*. As we can see in the figure, enabling the nested parallelism does not affect the performance of the task-based executions (first and second bars in the figure) because the implementation does not rely on the *parallel for* construct. Scenario (3), shown in the third bar of the figure, has the same performance as the original sequential MPICH, because nested parallelism is disabled, and as a result *MPI_Pack()* runs sequentially. As the figure shows, when the number of idle threads is 0, the first three scenarios behave similarly, while (4) is slower because nested parallelism is enabled and creates thread oversubscription. As we increase the number of idle threads, the task-based scheme achieves higher speedup due to the efficient use of the idle threads to perform the packing. The implementation using *parallel for* where nested parallelism is enabled, has consistently worse performance because new threads are being spawned that are oversubscribing the cores. This experiment demonstrates that when *MPI_Pack()* is called from inside an OpenMP parallel region, the task-based *MPI_Pack()* can efficiently use the idle threads. This results in better performance compared to the *parallel for* based *MPI_Pack()* which runs either sequential (nested disabled) or slower due to oversubscription (nested enabled). As the figure shows, if no idle threads are present, our approach does not incur any overhead (see the first two bars when number of idle threads = 0)

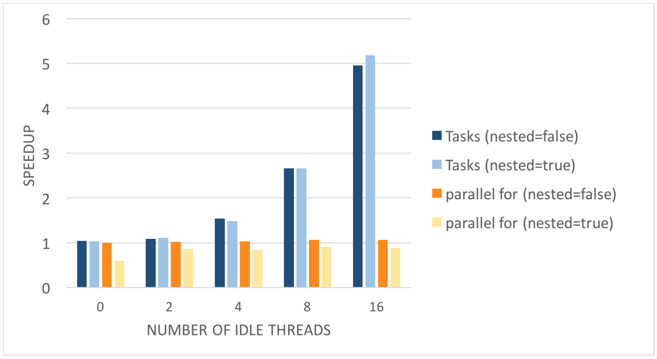

Fig. 10: Speedup of MPI_Pack() when called from parallel region under 4 different scenarios

2) *Vector of Vectors datatype:* In this section, we evaluate the speedup for packing the left surface of a 3D matrix of doubles, as shown by the blue color surface in Figure 7 and in the third row of Table I, Axis 3. In this case, we pack the Y-Z plane. The difference between the X-Z plane (described in the previous section) and the Y-Z plane (that we use in this section) is that in the Y-Z plane, none of the dimensions is contiguous. So, nested datatype is used. If we parallelize at the leaf level (vector datatype), we can only parallelize one column of Y non-contiguous elements at a time. This results in limited amount of parallelism, especially if the Y dimension is small. If we go one level higher (vector of vectors datatype), then several Y columns can be packed concurrently, increasing

the amount of parallelism per thread/task. Hence, parallelizing along Z dimension is more beneficial than along Y.

Figure 11 shows the speedup, when parallelizing the packing of the left surface of the 3D cube in Figure 7 with respect to the running time of the original sequential MPICH MPI_Pack(). For this experiment, the volume of the cube was set to 1GB and X was fixed to 2. Decreasing the value of Y, increases the value of Z. MPI_Pack() was called from the sequential part of application. The figure shows results as the value of the Y dimension (number_of_blocks) varies. Note that when the number of blocks is below 1K, we do not meet the conditions for packing in parallel, so the packing happens sequentially and all the lines overlap. Speedups are higher as the number of blocks increases. Using 128 threads gives the highest speedup (14.9X). Notice that the performance is not as good as for packing the top surface because packing the left surface has the limitation of the small size of the leaf vector.

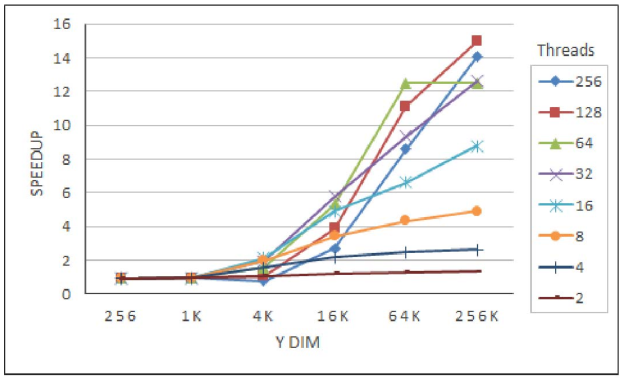

Fig. 11: Results from packing the left surface (Vector of Vectors datatype)

C. Applications

In this section, we report results for real applications. Section V-C1 reports results using the NAS MG benchmark, while Section V-C2 shows results using the transpose kernel from the Parallel Research Kernels.

1) NAS MG Benchmark: The NAS kernel Multigrid (MG) [9] is a simple 3D Multigrid benchmark. It is an implementation of several iterations of the V-cycle Multigrid algorithm to obtain an appropriate solution to the discrete Poisson problem. We used NAS version 3.3.1 in our experiments. The input to this benchmark is traditionally a cubic matrix whose size in each dimension is a power of 2. MG requires that the number of MPI ranks is also a cube of the power of 2. For example, to run a problem of total size 256x256x256 for 4 iterations using 8 processors, a 2x2x2 processor grid is formed and each partition on a processor is of size 128x128x128. Therefore, a maximum of 8 Multigrid levels may be used. These are of size 128, 64, 32, 16, 8, 4, 2, 1, with the coarsest level being a single point on a given processor. The *comm3* subroutine in NPB3.3.1/NPB3.3-MPI/MG/mg.f organizes the communication on all the borders. The *comm3* routine calls *ready*, *give*, and *take* sub-routines. The *give* sub-routine sends border data out in the requested direction. In the *give* routine, data was packed manually; we modified the code to use derived data types and call *MPI_Pack()* on it. Figure 7 shows the pictorial view of the three faces that are communicated in the NAS MG. Table I shows the datatypes used along each dimension. We assume the X dimension to be contiguous. Along axis 1, face YX gets communicated and the data is contiguous, so it can simply be copied using *memcpy*, without requiring packing. Along axis 2, the data is represented as vector datatype. Along axis 3, the data is represented as a vector of vectors datatype.

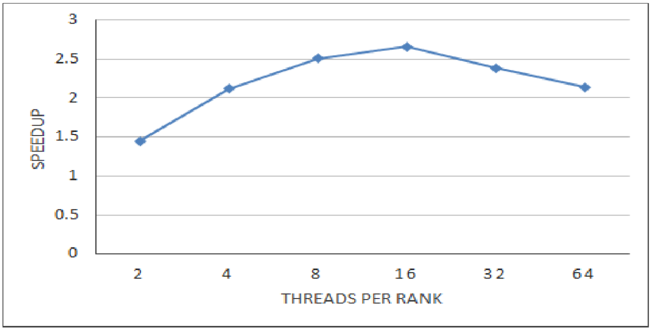

Fig. 12: Speedup in data packing along one axis in NAS MG

In this experiment, we run 8 MPI ranks on 4 KNL nodes with appropriate core bindings. The input is class D, which is a 3D matrix of order 1K. In Figure 12, we vary the number of threads per rank and the Y axis shows the speedup over the original MPICH that we get in data packing along axis 2. For the other two axes, the amount of data being communicated is small, so we do not see speedups. Note that the maximum speedup (2.65X) is obtained when using 16 threads per MPI rank. The speedup for this benchmark is not as good as expected because in NAS MG there are many data exchanges of small sizes, which do not benefit from parallelism. Note that the speedup in data packing does not contribute to a decrease of communication or overall execution time of the benchmark, as the amount of time spent in data packing is <0.5% of the total execution time.

2) Parallel Research Kernels: The Parallel Research Kernels [10] cover common communication, computation, and synchronization patterns encountered in parallel HPC applications. For this work, we modified the transpose kernel to use vector datatypes and called *MPI_Pack()* explicitly before communicating the data. In this kernel, a square matrix A of order n is divided into strips (columns) among the ranks. To calculate the transpose, the matrices are distributed identically, necessitating a global arrangement of the data (all-to-all communication), as well as a local rearrangement (per rank transpose of matrix tiles). Figure 13 shows the global view of the transpose operation on the entire matrix. Each rank first does a local transpose on data, which is not communicated across ranks. The rest of the data on the rank is transposed and packed into a contiguous buffer for each rank. Here we saw the opportunity to represent the data using derived datatypes. After the all-to-all communication, each rank unpacks the data it receives and stores it in the appropriate cell in the transposed matrix B. As the data is being transposed and packed in the same step, it is not contiguous in any dimension on any rank. So we had to use vector of vectors datatype to express the data. For the leaf level vector datatype, *block_length* is 1, *number_of_blocks* is block_order and *stride* is num_ranks*block_order. The top level vector has block_order number of vectors.

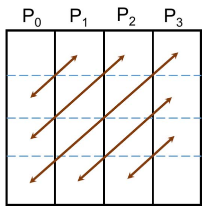

Fig. 13: Matrix transpose)

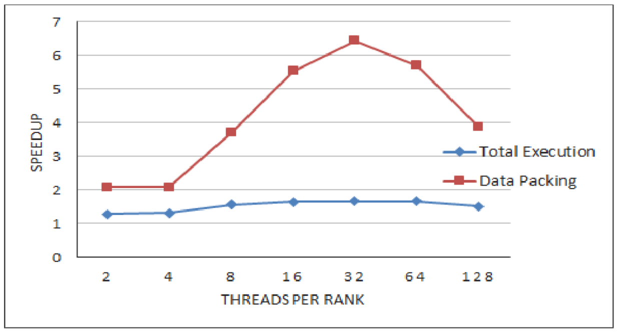

Fig. 14: Speedup in transpose kernel when 2 ranks are placed on 1 KNL node (Intranode)

Figure 14 shows the results we obtain for this experiment. We transpose a square matrix of order 8K doubles using 2 MPI ranks while running on a single node of KNL and binding ranks to cores. The X axis shows the number of threads per rank and the Y axis shows the speedup we get over the original MPICH implementation. We measure the time taken to pack data using explicit timers on each rank and call *MPI_Reduce()* on it using *MPI_MAX* as op. Speedup obtained on pack_time is shown by the red line and speedup on the total transpose time is shown by the blue line. We get the best speedup when 32 threads are used per rank. Scaling beyond this uses hyper-threading and the speedup decreases. Though we get speedup of up to 6.4X on packing, its impact on the speedup of the total transpose time is up to 1.6X. We expect to see higher speedups for transpose of matrices of higher order.

Figure 15 shows the same experiment but running on 2 KNL nodes where we bind ranks to cores. A matrix of 16K doubles is transposed using 4 MPI ranks, 2 ranks per KNL node. On the X-axis, we vary threads per rank and the Y-axis shows the speedup over original MPICH. As in the previous plot, the speedup is best when we use 32 threads per rank.

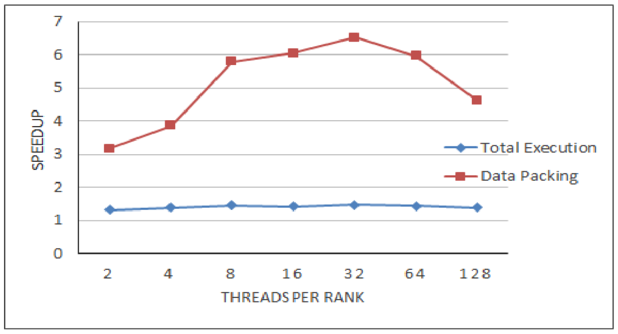

Fig. 15: Speedup in transpose kernel when 4 ranks are placed on 2 KNL nodes (Internode)

VI. Related Work

Our work in this paper was inspired by the MT-MPI work by Si et al in [1], where the idea of using OpenMP to parallelize MPI was first proposed. MT-MPI is an internally multithreaded MPI implementation that tries to coordinate with the application to utilize idle OpenMP threads. MT-MPI relies on the information about the idle threads, which is provided by a modified version of the OpenMP runtime. Although this approach works well for an OpenMP application without using tasks, the method it uses to define an idle thread does not work when OpenMP tasks are present. This is because when OpenMP threads reach a barrier, they can still be busy executing tasks before they really become idle. Furthermore, since MT-MPI directly uses the OpenMP "parallel for" construct to parallelize MPI functions, with nested parallelism, this approach can easily result in oversubscription. In comparison, we use OpenMP tasks to parallelize MPI, and since MPI does not create new threads, the oversubscription issue is avoided. In addition, this approach requires a modified OpenMP runtime library, binding the MPI library to a specific OpenMP implementation. In contrast, our approach uses only standard OpenMP interfaces supported by all OpenMP compilers, thus allowing the user to choose which OpenMP compiler to use independently of the MPI library.

The Intel MPI library for Intel Xeon Phi Co-processors supports multithreaded *memcpy* [11]. This approach is similar to MT-MPI and can also suffer from oversubscription.

Blue Gene/Q CNK tried to solve the thread oversubscription by deploying an underlying common resource manager [12], [13] that manages the hardware threads for MPI and OpenMP. By default, it does not encourage thread oversubscription. MPI attempted to pick threads that were unlikely to be used at the start for its progress engine. When a CPU is over committed, the scheduler would preempt the lower priority thread running on the CPU. Although, using a common resource manager sounds like an ideal solution to avoid thread oversubscription, in practice, having such common resource manager for all runtimes is difficult and requires substantial changes to the MPI and OpenMP runtimes.

MPI supports the *MPI_THREAD_MULTIPLE* threading model [14], [15]. To support this model, the MPI runtime needs to protect the shared resources using critical sections and so each MPI call can end up being serial if not carefully implemented. The implementation of *MPI_THREAD_MULTIPLE* is orthogonal to the issue discussed in this paper. In fact, our implementation also works with this threading model, as demonstrated in our experimental results.

Casper [16] offloads MPI RMA communication to a small, user-specified number of cores on a multicore or many-core environment. These "ghost processes" are dedicated to help asynchronous progress for user processes through appropriate memory mapping from those user processes. Since these dedicated cores are reserved only for MPI RMA communication, there is no oversubscription issue. However such an implementation potentially results in lower application performance because fewer cores are used for computation, especially when RMA communication does not dominate.

VII. Conclusions

In this paper, we have proposed and assessed the use of tasks and threads to parallelize the MPI library itself. The thread that calls the MPI runtime can create tasks that can be executed by the calling thread and any other application threads that are idle at task scheduling points, such as barriers. We have evaluated implementations using OpenMP tasks and OpenMP threads to parallelize intranode communication, packing of data when using explicit calls to *MPI_Pack()* or communication through derived datatypes. We performed experiments when the MPI call is made from a parallel region and compared the approach using a *parallel for* construct and tasks. Our results show that the approach using tasks performs significantly better than the one using OpenMP threads mostly due to avoiding oversubscription. In the case when no idle threads are present, our approach shows similar performance to the baseline implementation where the MPI library runs sequentially. The approach using OpenMP tasks does not need to take care of the values of configuration variables, such as whether nesting is enabled. We have identified the thresholds where parallelization produces speedup. This requires tuning parameters such as *task_size* to the target architecture so that the performance is never hurt. Overall, our results show that parallelizing MPI using tasks produces speedups of up-to 2.5x for shared memory communication of messages of size 32MB or larger. For *MPI_Pack()*, speedups depend on the number of available threads, but we observe speedups of up-to $62X$ when using 64 threads. We have also run the transpose kernel from PRK and obtained a speedup of 6.5X on data packing. For the NAS MG benchmark, we obtained speedup of 2.65X on data packing.

Our conclusion is that the use of OpenMP tasks provide a simple and efficient way to parallelize the MPI library. This approach is appealing because it guarantees that there is no oversubscription and provides a mechanism for application threads that are otherwise idle to help in the execution of tasks that other threads might have created, allowing the most efficient use of the computing resources.

VIII. Acknowledgement

This material is based upon work supported by Subcontract No. B609815 with Argonne National Laboratory and Intel Federal LLC.

References

[1] M. Si, A. J. Peña, P. Balaji, M. Takagi, and Y. Ishikawa, "Mt-mpi: Multithreaded mpi for many-core environments," in *Proc of the 28th ACM International Conference on Supercomputing*, 2014, pp. 125–134.

[2] K. Vaidyanathan, D. D. Kalamkar, K. Pamnany, J. R. Hammond, P. Balaji, D. Das, J. Park, and B. Joó, "Improving concurrency and asynchrony in multithreaded mpi applications using software offloading," in *Proc of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015, pp. 30:1–30:12.

[3] D. Buntinas and G. Mecier, "Implementation and shared-memory evaluation of mpich2 over the nemesis communication subsystem," in *Proc of Euro PVM/MPI*, 2006.

[4] T. Hoefler and S. Gottlieb, "Parallel zero-copy algorithms for fast fourier transform and conjugate gradient using mpi datatypes," in *Recent Advances in the Message Passing Interface*. Springer, 2010, pp. 132–141.

[5] G. Santhanaraman, J. Wu, W. Huang, and D. K. Panda, "Designing zero-copy message passing interface derived datatype communication over infiniband: Alternative approaches and performance evaluation," *International Journal of High Performance Computing Applications*, vol. 19, no. 2, pp. 129–142, 2005.

[6] "Mpich website," http://www.mpich.org.

[7] R. Ross, N. Miller, and W. D. Gropp, "Implementing fast and reusable datatype processing," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 2003, pp. 404–413.

[8] "Colfax research," http://colfaxresearch.com/knl-mcdram/.

[9] "Nas parallel benchmarks," http://www.nas.nasa.gov/publications/npb.html.

[10] R. Wijngaart, A. Kayi, J. Hammond, G. Jost, T. John, S. Sridharan, T. Mattson, J. Abercrombie, and J. Nelson, "Comparing runtime systems with exascale ambitions using the parallel research kernels," in *ISC*, 2016, pp. 321–339.

[11] Sergey Kazakov, "Multi-threaded Memcpy Support," http://software.intel.com/en-us/node/528848.

[12] J. E. Moreira *et al.*, "The blue gene/l supercomputer: A hardware and software story," *Int. J. Parallel Program.*, vol. 35, no. 3, pp. 181–206, Jun. 2007.

[13] Megan Gilge, "IBM System Blue Gene Solution - Blue Gene/Q - Application Development," http://www.redbooks.ibm.com/redbooks/pdfs/sg247948.pdf.

[14] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, and R. Thakur, "Toward efficient support for multithreaded mpi communication," in *Proc. of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 2008, pp. 120–129.

[15] D. Goodell, P. Balaji, D. Buntinas, W. Gropp, S. Kumar, B. de Supinski, and R. Thakur, "Minimizing mpi resource contention in multithreaded multicore environments," in *IEEE Cluster*, 2010.

[16] M. Si, A. J. Pea, J. R. Hammond, P. Balaji, M. Takagi, and Y. Ishikawa, "Casper: An asynchronous progress model for mpi rma on many-core architectures." in *IPDPS*, 2015, pp. 665–676.