MPI+ULT: Overlapping Communication and Computation with User-Level Threads

Huiwei Lu Sangmin Seo Pavan Balaji

Mathematics and Computer Science Division
Argonne National Laboratory, Argonne, IL 60439, USA
{huiweilu, sseo, balaji}@anl.gov

***Abstract*—As the core density of future processors keeps increasing, MPI+Threads is becoming a promising programming model for large scale SMP clusters. Generally speaking, hybrid MPI+Threads runtime can largely improve intra-node parallelism and data sharing on shared-memory architectures. However, it does not help much on inter-node communication due to the inefficient integration of existing communication and threading libraries. More specifically, existing MPI+Threads runtime systems use coarse-grained locks to protect their thread safety, which leads to heavy lock contention and limit the scalability of the runtime. While kernel threads are efficient for intra-node parallelism, we found that they are too heavy for computation/communication overlap in an MPI+Threads runtime system. In this paper we propose a new way for asynchronous MPI communication with user-level threads (MPI+ULT). By enabling ULT context switching inside MPI, MPI communication in one ULT can overlap with computation or communication in other ULTs. MPI+ULT can be used for communication hiding in various scenarios, including MPI point-to-point, collective and one-sided calls. We use MPI+ULT in two applications, a high-performance conjugate gradient benchmark and a genome assembly application, to show how MPI+ULT can help effectively hide communication and reduce runtime overhead. Experiments show that our method helps improve the performance of these applications significantly.**

I. Introduction

The computing power on a single CPU chip continues to grow exponentially as more cores are squeezed into one chip. For example, Knights Landing, Intel's next-generation Many Integrated Core architecture, will have more than 60 cores on a single chip. While this increased parallelism continues to provide performance improvements for single-node performance, it will also put more pressure on the communication network. As communication overhead grows with the number of cores, the scalability of future large-scale scientific applications is likely to be limited by communication. Although during the past decade the bandwidth of interconnect networks has increased, their latency has experienced only limited improvement since it is bound by the speed of light. One can hide this communication latency with computation, but asynchrony will be needed with multiple levels of concurrency [1].

MPI provides nonblocking routines to overlap communication and computation. Its benefits have been explored by several applications [2], [3], [4]. Recently, the MPI-3.0 standard has added support for nonblocking collectives [5], [6], further extending the use of MPI nonblocking operations to applications with collective communication patterns [7]. However, the support of asynchrony in MPI is still not complete. Some MPI one-sided functions, such as `MPI_Win_Flush`, still do not have corresponding nonblocking APIs.

Another possible solution is to use MPI+Threads. Each thread will do computation and communication independently, thus increasing the concurrency and achieving computation and communication overlap. However, most previous work [8] has focused primarily on using threads to increase the parallelism of intra-node computation; little investigation has been done on the use of threads for computation/communication overlap. The problem of MPI+Threads lies in thread safety. Current MPI implementations either do not support `MPI_THREAD_MULTIPLE` or support it only at a preliminary level with a coarse-grained lock. Recent work shows heavy lock contention in MPI+Threads runtime [9], limiting its use by the community.

In this paper we propose MPI+ULT, a new approach to overlap communication and computation in MPI with user-level threads (ULTs). ULT provides thread semantics in user space, which makes it lightweight at thread creation and yielding. With ULT, overlap of communication and computation can be achieved easily at a low cost. The idea is to create multiple computation and communication tasks in different ULTs; if one ULT is blocked in a communication task, MPI runtime will detect it and context switch to another ULT to make progress. In this way, we can keep the CPU busy doing useful work rather than waiting the blocked communication to finish. MPI+ULT provides several advantages over existing runtime systems. Compared with nonblocking MPI calls, MPI+ULT provides better programmability by providing modularity in computation/communication overlap. Compared with MPI+Pthreads and MPI+OpenMP, MPI+ULT provides asynchronous communication without additional overhead on hardware resources or added parallel complexity.

We propose a new thread level for MPI: `MPI_THREAD_ULT`, to clarify the case where only one kernel thread will execute but multiple ULTs may call MPI functions with no restrictions. This thread level is different from `MPI_THREAD_SERIALIZED`

and MPI_THREAD_MULTIPLE, as ULTs are able to overlap different MPI calls without executing concurrently.

We evaluate MPI+ULT with several microbenchmarks on point-to-point, collective and one-sided MPI operations. Results show that the overhead of using ULT for blocking MPI calls is close to or even lower than their nonblocking counterparts. Moreover, there are no nonblocking MPI functions for one-sided synchronization. With MPI+ULT, one can execute MPI one-sided synchronization asynchronously, thus providing comprehensive support for communication/computation overlap for all types of MPI calls.

We use MPI+ULT to improve the communication performance in HPCG [10], a high-performance conjugate gradient benchmark. HPCG includes two key communication patterns: global collective communication and neighborhood communication. With MPI+ULT, both patterns can be easily wrapped in a ULT to overlap with computation. Experiments on an Intel CPU-based cluster show that HPCG using MPI+ULT gets a performance improvement of 19.8% over MPI-only version on 2,048 cores because of communication hiding.

We use MPI+ULT to improve the performance of SWAP [11], a parallel genome assembly application for processing massive sequence data on thousands of cores. By replacing Pthreads with ULT in the original algorithm, we reduce the overhead of threads and eliminate the need for locks in MPI runtime. The resulting MPI+ULT implementation is between 2.0 and 6.3 times faster than MPI+Pthreads implementation, depending on the number of cores used.

The rest of the paper is organized as follows. Section II introduces some background. Section III describes the design and implementation of MPI+ULT. Section IV presents microbenchmarks results for MPI+ULT, and Section V shows how MPI+ULT can be used in applications. Section VI discusses related work, and Section VII summarizes our conclusions and briefly discusses future work.

II. Background

A. Overlapping Communication and Computation

Asynchronous communication can improve application scalability and hide communication latency. The MPI standard defines nonblocking communication routines to improve application performance by overlapping communication and computation. Nonblocking point-to-point communication was defined in MPI-1. Nonblocking collective communication was added to MPI-3 recently [5], [6]. These interfaces provide a basic building block at the API level, achieving overlap for a single operation, but lack a systematic way to overlap communication and computation together.

An alternative mechanism is to use threads for overlap. But this approach would require an MPI implementation that offers MPI_THREAD_MULTIPLE support. Unfortunately, current MPI implementations either do not support this thread level or support it only preliminarily with a global lock that leads to heavy contention, limiting the adoption of threads in MPI in practice.

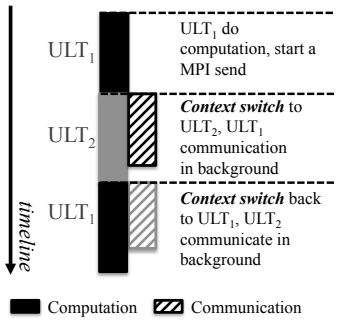


Fig. 1: Overlap computation and communication with ULT.

B. User-Level Threads

User-level threads provide thread semantics in user space. Compared with kernel threads, ULT is more lightweight. Creation and yielding can be done at a low cost because they do not require system calls. In this paper, we use ULT to denote a user-level thread, while using kernel thread, POSIX thread or thread to denote a kernel thread. ULT can be implemented with coroutines. Coroutines [12] are a generalization of routines. A coroutine enables explicit suspend and resume of its progress by preserving execution state. Some languages such as Python and Go [13] provide coroutines for asynchronous I/O. Fig. 1 shows how to use ULT for computation and communication overlapping.

The execution model of ULT is cooperative timesharing. Multiple ULTs may be mapped to the same kernel thread, where ULTs get executed by interleaving with each other. The model uses *context* to save the contents of CPU registers and stack space. Each ULT gets executed by context switching to each other. ULT can provide a thread abstraction in order to group related communication and computation together, thus providing better modularity for programs. Its stack can be used to store temperary results and simplify some programming tasks. In this paper we focus on a "fork-join" ULT model with three functions: ult_fork, ult_join, and ult_yield. In this model ult_fork creates a ULT; ult_join waits a ULT to finish; and ult_yield causes an ULT to yield execution to other ULTs.

C. MPI+X Programming Models

Threads allow the runtime to better adapt to the increasing core density in cluster nodes. Thus, the "MPI+X" model is considered a promising programming model for future extreme-scale machines, where MPI works complementarily with threads to provide efficient inter-node communication and intra-node computation.

The current MPI standard supports four thread levels for thread safety: MPI_THREAD_SINGLE, where only one thread will execute; MPI_THREAD_FUNNELED, where the process may be multithreaded but only the main thread will make MPI calls; MPI_THREAD_SERIALIZED, where the process may be multithreaded and multiple threads may make MPI calls, but only one at a time: MPI calls are not made concurrently

from two distinct threads (all MPI calls are "serialized"); and MPI_THREAD_MULTIPLE, where multiple threads may call MPI, with no restrictions. However, the standard generally assumes that the thread package used with MPI is similar to POSIX threads, but does not consider the case of user-level threads carefully.

D. MPI Progress

Current MPI libraries do not offer true asynchronous progress. MPI runtime either needs an additional kernel thread for making asynchronous progress or needs to periodically call an MPI function (e.g., MPI_Test) to advance all outstanding operations for round-transition of collective operations. In practice, the user needs to insert calls in computation to periodically poll the MPI runtime in order to make progress on communication. For brevity, this polling is not shown in the microbenchmarks and applications in the paper.

III. MPI+ULT

A. New MPI Thread Level for ULT

The current MPI standard [5] defines the interaction between MPI calls and threads but assumes the thread package is similar to POSIX threads. It does not cover the case when ULT is involved. There are two scenarios: ULTs can be either created on different kernel threads or all on the same kernel thread. For the first case, MPI should use MPI_THREAD_MULTIPLE because different ULTs may execute concurrently. However, for the second case, the current MPI does not have a proper thread level for it. We cannot use MPI_THREAD_SERIALIZED, which will not allow switching between ULTs while calling an MPI function like in Fig. 1. We can use MPI_THREAD_MULTIPLE, but this level adds a substantial overhead to MPI runtime. Given that ULT on the same kernel thread will not execute concurrently, it should be distinguished from kernel thread to avoid the lock cost when using MPI_THREAD_MULTIPLE.

To distinguish the above case, we propose a new level of thread support in MPI: MPI_THREAD_ULT, where only one kernel thread will execute but multiple ULTs may call MPI functions with no restrictions. Compared with MPI_THREAD_SERIALIZED, MPI_THREAD_ULT provides the opportunity for overlapping of different MPI calls. Compared with MPI_THREAD_MULTIPLE, MPI_THREAD_ULT uses ULT instead of POSIX threads. In MPI_THREAD_MULTIPLE, in order to protect the critical section in MPI library, each MPI call will acquire a lock before entering the library. With MPI_THREAD_ULT, since ULTs are not concurrently executed, this lock can be avoided. Note that when multiple ULTs execute on multiple kernel threads, we do not use MPI_THREAD_ULT but have to use MPI_THREAD_MULTIPLE, as different ULTs can execute concurrently. This new level, MPI_THREAD_ULT, is proposed to help MPI runtime to distinguish the case specifically when multiple ULTs execute on the same kernel thread but can overlap different MPI calls without executing concurrently.

```
1  int main()
2  {
3    ult_fork(fn_thread, &param, &thread);
4    ult_yield();
5    /* do independent computation */
6    ult_join(thread);
7  }
8
9  void fn_thread()
10 {
11   MPI_Send(buf, count, MPI_CHAR, dest, tag
         , comm);
12   /* other computation or communication */
13 }
```

Listing 1: Example of using ULT with MPI for communication/computation overlap (MPI_Send)

B. Using MPI+ULT to Overlap Computation with Communication

ULT provides an effective way for fast context switch between different tasks. By integrating ULT with MPI, the runtime can provide a lightweight mechanism for computation/communication overlap. Applications can use different ULTs for different computation and communication tasks and can overlap computation and communication by context switching between them.

Listing 1 shows an example of how to use ULT for computation/communication overlap. The main function forks a ULT called thread, which will execute fn_thread to do MPI_Send. It then yields to the new ULT to execute MPI_Send. Inside the MPI runtime, the issued messages in MPI_Send are checked by the progress engine. If the checked messages are still on the fly, the progress engine will yield to other ULTs in order to make effective use of this waiting time. In this case, the ULT thread yields back to the main thread to do computation while waiting for MPI_Send to finish, thus achieviing overlap of computation and communication. With fn_thread, there is no restriction on which MPI call or how many MPI calls can be used. It can be blocking, nonblocking, collective or one-sided communication MPI calls. Moreover, multiple MPI calls can be made inside fn_thread. The data dependence is provided by ult_fork and ult_join. In this example, if the user wants to change the content of buf, it should be placed after ult_join.

C. Implementing MPI+ULT

The idea of MPI+ULT is to fork multiple ULTs for different computation and communication tasks and to do a context switch when one ULT is polling in a blocking MPI call. By switching to other ULTs, the time of waiting for the polling to finish can be used effectively. In order to enable the integration, MPI runtime needs to be aware of the ULT library and be able to do the context switch at the appropriate time.

Wrapper-Based MPI Runtime: ULT can be integrated with MPI in several ways. One way is to use MPI wrappers to convert a blocking call to a nonblocking equivalent and do ULT yielding while waiting for the nonblocking call to finish.

Since changes are made on top of MPI, the MPI runtime does not need to be modified, and this approach can be applied to different MPI implementations. However, every MPI function call needs to be modified. A function that does not have a nonblocking equivalent will not be able to use this approach.

ULT-Aware MPI Runtime: Another approach is to integrate ULT tightly inside the progress engine of MPI runtime, where MPI keeps issuing messages and polling the status of requests. When there is a blocking call inside the progress engine, the runtime will yield. The disadvantage of this approach is that it needs to modify the MPI runtime to get it work.

We choose the second approach to implement MPI+ULT because it has several advantages. First, it has a potentially lower overhead than the first approach. As we explained in Section II-D, MPI needs to poll the progress engine in order to make progress on sending and receiving messages. With the first approach, runtime will yield after each polling. With the second approach, MPI internally will decide how many times MPI should poll the progress engine before yielding, thus reducing the yielding cost. It will benefit MPI collective calls because one collective call will need multiple polls.

Second, MPI blocking calls have been implemented and optimized for a long time. The nonblocking calls have just been implemented, and some of them are still not well optimized. For example, we will later see in microbenchmark tests that `MPI_Alltoall` using ULT will have a lower overhead than `MPI_Ialltoall` does (Section IV-D).

Third, the second approach provides an additional advantage that it supports MPI blocking calls that do not have nonblocking equivalent yet (e.g., `MPI_Win_flush`), thus providing more comprehensive support for MPI calls.

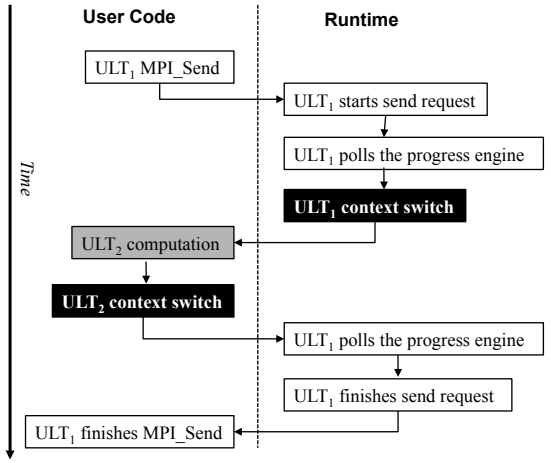


Fig. 2: Timeline of `MPI_Send` with ULT-aware MPI runtime. ULT is integrated with MPICH's progress engine and provides overlap for different ULTs.

Fig. 2 illustrates how ULT is integrated with the progress engine of MPI runtime (here, MPICH). The figure shows part of the timeline of Listing 1, where ULT_1 denotes `thread` and ULT_2 denotes `main`. After ULT_1 started `MPI_Send`, the runtime starts a send request and then polls the request in the progress engine. While waiting for the polling to finish, ULT_1 context switches to ULT_2 to do computation in ULT_2. After ULT_2 finishes its computation, it then context switches back to the position where ULT_1 was polling. After it confirms that the message has been sent, ULT_1 finishes this send request and returns to the user code. The *context switch* between ULT_1 and ULT_2 enables the runtime to make use of the busy polling time, thus providing effective computation/communication overlap.

IV. Micro-Benchmark Results

We designed microbenchmarks to measure the thread overhead in MPI and the potential overlap in ULT-aware MPI calls, including point-to-point, collective, and one-sided tests.

A. Experiment Setup

We conducted our benchmarks on an Intel cluster, named Blues, at the Argonne Laboratory Computing Resource Center. Blues consists of 310 nodes connected with QLogic interconnect, each with two Sandy Bridge Pentium Xeon with 64 GB RAM. Our MPI+ULT library is based on MPICH 3.1.3 with TCP netmod. We use Qthreads [14] as our ULT library. Here `ult_fork`, `ult_join`, and `ult_yield` correspond to `qthread_fork`, `qthread_readFF`, and `qthread_yield`. We choose Qthread because it is one of the most portable user-level thread libraries.

We use `MPI_Wtime` to measure the elapsed time. When measuring point-to-point MPI calls, several warm-up messages were issued before the real measurement started. When measuring collective calls, `MPI_Barrier` is called to limit the interprocess skew before every measurement. Both MPI-only nonblocking calls and MPI+ULT use an interval of 100 μs to poll MPI for progress (see Section II-D). The overhead of a ULT-aware MPI call denotes the time that is spent in the ULT function and in the network stack. It includes forking the threads, polling the progress engine, and joining the threads.

B. Overhead of Kernel Threads vs. ULT

We use the test of `MPI_PROC_NULL` [15] to measure the threading overhead in the MPICH code in the absence of any network communication. If a process executes a send with destination `MPI_PROC_NULL`, MPI will enter this send call, and return immediately. The test is to measure the costs of entering an MPI call for different thread configurations. Pthreads and ULT-multiple are configured to use `MPI_THREAD_MULTIPLE` with "Global" critical-section granularity. ULT-single is configured to use `MPI_THREAD_ULT`. Fig. 3 shows the aggregate message rate of the sending threads or ULT as a function of the number of threads or ULTs. In the case of ULT-single, we use one kernel thread inside MPI and create multiple ULTs inside the kernel thread. Its performance is almost identical to that of MPI-only using one process because ULT does not need a lock to enter an MPI function. With Pthreads, however, there is a considerable decline in message rate because different threads

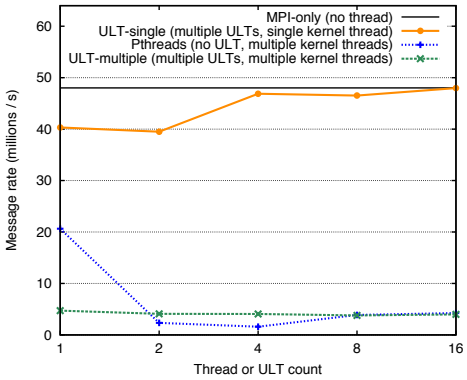


Fig. 3: Message rate for a multithreaded process sending to `MPI_PROC_NULL`.

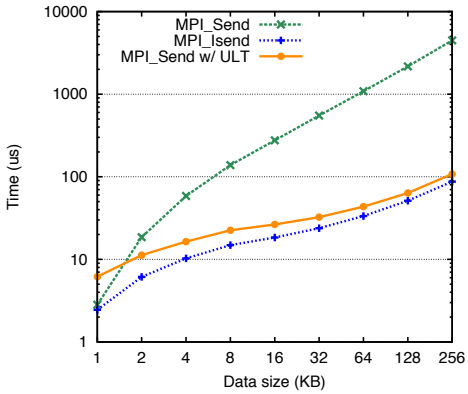


Fig. 4: Overhead of blocking, nonblocking, and ULT MPI send on 2 nodes.

are competing to acquire the critical section on entry to an MPI function, which serializes the access of threads and causes lock contention. In the case of ULT-multiple, we use multiple ULTs executing on multiple kernel threads. We use 16 ULTs in total but distribute them evenly on kernel threads. It suffers the same problem of lock contention as Pthreads and has more overhead to schedule different ULTs on different kernel threads, so its performance is similar to or even worse than that of Pthreads. Note that Pthreads and ULT-multiple use more hardware resources than ULT-single but in fact have a lower message rate because of the lock contention.

C. Overlap in Point-to-Point Communication

Fig. 4 shows the overhead of MPI send using different approaches. As expected, the overhead of `MPI_Isend` is always smaller than that of `MPI_Send`. As the data size grows, the performance gap between `MPI_Isend` and `MPI_Send` becomes larger. The overhead of `MPI_Send` using ULT is more than twice as much as the overhead of `MPI_Isend`. The reason is that ULT adds additional thread creation and yielding cost. As the data size becomes larger, however, the overheads become close because the cost of ULT becomes relatively small compared with the communication overhead

```
1  MPI_Win_allocate(win_size, ..., &win_get);
2  MPI_Win_allocate(win_size, ..., &win_acc);
3  ...
4  MPI_Win_lock_all(0, win_get);
5  MPI_Win_lock_all(0, win_acc);
6  for (i = 0; i < num_iter; i++) {
7    ...
8    MPI_Get(buf, num_elem, MPI_INT, target,
         target_disp, num_elem, MPI_INT,
         win_get);
9    MPI_Win_flush(target, win_get);
10
11   do_comp(buf, num_elem);
12
13   MPI_Accumulate(buf, num_elem, MPI_INT,
         target, target_disp, num_elem,
         MPI_INT, MPI_SUM, win_acc);
14   MPI_Win_flush(target, win_acc);
15   ...
16 }
17 MPI_Win_unlock(win_get);
18 MPI_Win_unlock(win_acc);
19 ...
20 MPI_Win_free(&win_get);
21 MPI_Win_free(&win_acc);
```

Listing 2: Micro-benchmark code for one-sided communication

itself. The gap between `MPI_Send` and `MPI_Send` using ULT shows the maximum potential overlap that can be used for computation.

D. Overlap in Collective Communication

Fig. 5 shows the overhead of blocking, nonblocking, and ULT MPI collective communication. In all three collective communications (`MPI_Bcast`, `MPI_Allgather`, and `MPI_Allreduce`), the overhead of nonblocking MPI calls is always smaller than the overhead of doing MPI calls using ULT. For `MPI_Alltoall`, however, the overhead of `MPI_Alltoall` using ULT is the smallest. The reason is that the algorithm for blocking `MPI_Alltoall` has been well optimized during the lifespan of MPICH compared with the newly added `MPI_Ialltoall`. These results show that the overhead of MPI collective communication with ULT is close to or even less than using nonblocking collectives directly.

E. Overlap in One-Sided Communication

To see the benefit of MPI+ULT in one-sided communication, we used a microbenchmark that mainly conducts `MPI_Get`, computation, and `MPI_Accumulate`. Listing 2 shows a simplified code for the microbenchmark. First, we create two windows, called `win_get` and `win_acc`. Each process contributes the same amount of memory to create windows. Then, two windows are locked, and each process carries out the main loop where one-sided communication and computation occur. Since `buf` obtained from `MPI_Get` is used in the computation function, `do_comp`, `MPI_Win_flush` at line 9 is called to complete the outstanding `MPI_Get` operation. Similarly, `buf` is used in the next iteration of `for` loop, and thus `MPI_Win_flush` at line 14 is invoked to complete

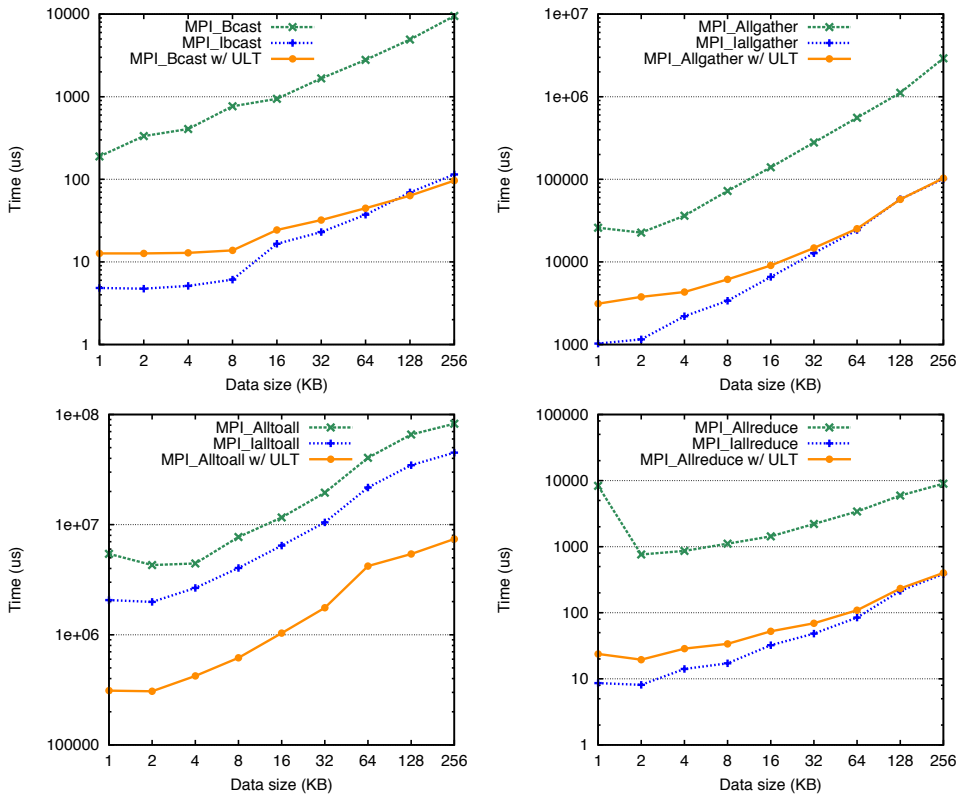


Fig. 5: Overhead of blocking, nonblocking, and ULT MPI collective communication on 1,024 cores.

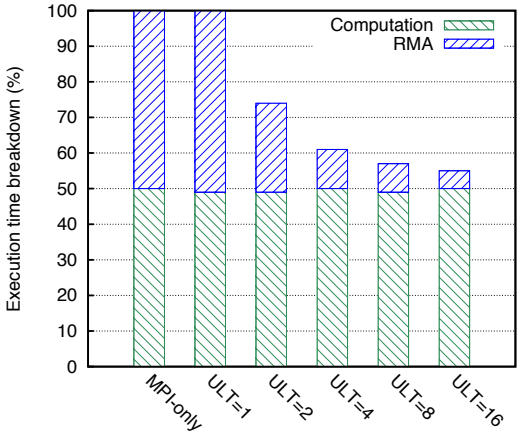


Fig. 6: RMA time breakdown.

the outstanding `MPI_Accumulate` operation. `target` and `target_disp` are changed every iteration in a round-robin manner so that RMA operations access all processes and all memory regions in all windows. For MPI+ULT, we create a number of ULTs that cooperatively execute the `for` loop in Listing 2. In order for each ULT to execute the code, it has its own copy of `buf`. Consequently, MPI+ULT code spends more memory for `buf` according to the number of ULTs used.

Fig. 6 illustrates performance results for MPI-only execution, denoted by MPI-only, and MPI+ULT executions with various number of ULTs, denoted by ULT=X, where X is the number of ULTs used. All execution times are normalized to that of MPI-only. The figure shows the execution time breakdown, which consists of the computation time (Computation) and communication time (RMA). For the experiments, 64 processes were used, and each process contributed 1 GB (64 GB in total) of memory for windows. The data size for each RMA operation was 4 KB.

The execution times of ULT=X in Fig. 6 show that using more than one ULT improves the performance by hiding the communication time. ULT=2, ULT=4, ULT=8, and ULT=16 reduced 26%, 39%, 44%, and 45% of the entire execution time, respectively. From this experiment, we note that using more than eight ULTs does not significantly reduce the execution time. The results indicate that our ULT-aware MPI runtime can efficiently switch ULTs even for one-sided communication and can help reduce the execution time of applications using this kind of communication/computation pattern. Note that although RMA operations such as `MPI_Get` are nonblocking, synchronization calls such as `MPI_Win_flush` are blocking. Therefore, whenever a process has to wait on a synchronization operation, the execution time of application is wasted in waiting. Overcoming this limit of blocking synchronization is not easy, however, because there are no nonblocking counterparts for RMA synchronization operations.

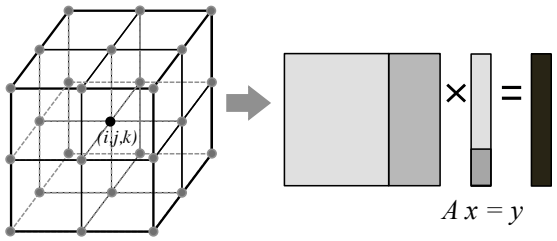


Fig. 7: The 27-point operator in HPCG. The equation at point (i, j, k) depends the value at its location and its 26 surrounding neighbors. The resulting sparse linear system of equations, $Ax = y$, has 27 nonzero entries per row for the interior and 8 to 18 nonzeros entries for boundary equations.

V. Application Results

In this section, we describe how to overlap communication with computation in applications. We look at HPCG, a high-performance conjugate gradient benchmark that is designed to correlate with a broad set of important scientific applications, especially for those governed by partial differential equations. Also, we use MPI+ULT to reduce the runtime overhead in SWAP-Assembler, a genome assembly application that uses MPI+Pthreads programming.

A. High Performance Conjugate Gradient

HPCG [10] is a recently announced benchmark intended to complement the High Performance Linpack (HPL) benchmark currently used to rank supercomputers in the TOP500 list. HPCG is designed to exercise computational and data access patterns that more closely match a broad set of important applications and to give incentive to computer system designers to invest in capabilities that will have impact on the collective performance of these applications. The HPCG benchmark generates a synthetic discretized three-dimensional partial differential equation model problem and computes preconditioned conjugate gradient (PCG) iterations for the resulting sparse linear system.

HPCG has two key communication patterns. One is an all-reduce collective operation, which is used to compute the residual of each iteration. The other is a neighborhood communication, which is used to exchange data between neighbors. These two communication patterns represent essential performance bottlenecks for many real applications. They are prevalent in a variety of methods for discretization and numerical solution of partial differential equations.

Listing 3 shows the simplified main loop of HPCG. The algorithm solves the equation $Ax = b$ using iterative method, where A is usually a large and sparse matrix, b is a known vector, and x is the computed result. The algorithm computes x by iteratively guessing to obtain a good approximation to the solution x until the residual is small enough or the algorithm has reached the maximum number of iterations. The function `MG` denotes a multigrid method used as a preconditioner in HPCG. It provides a powerful technique to accelerate the convergence of iterative solvers for linear systems. The

```
1  for (int k = 1; k <= max_iter && normr >
       tolerance; k++) {
2    MG(A, r, z);
3    if (k == 1) {
4      WAXPBY(1.0, z, 0.0, z, p);
5      DDOT(r, z, rtz);
6    } else {
7      DDOT(r, z, rtz);
8      WAXPBY(1.0, z, beta, p, p);
9    }
10   SpMV(A, p, Ap);
11   DDOT(p, Ap, pAp);
12   WAXPBY(1.0, x, alpha, p, x);
13   WAXPBY(1,0, r, -alpha, Ap, r);
14   DDOT(r, r, normr);
15   normr = sqrt(normr);
16 }
```

Listing 3: The simplified main loop of HPCG

function `DDOT(x, y, r)` computes the dot product of two vectors $r = x \cdot y$. `WAXPBY(alpha, x, beta, y, w)` computes $w = alpha * x + beta * y$, where x and y are vectors and $alpha$ and $beta$ are scalars. `SpMV(A, x, y)` computes the product y of a matrix A and a vector x.

Hiding Global Collective Communication in an Iterative Method: The first key communication pattern is global collective communication. In the original HPCG algorithm, `DDOT` (Listing 3, line 14) is used to calculate the residual $normr$ of each iteration and then compare it with a static $tolerance$ value. The function `DDOT` first computes a dot product of two vectors, and then uses `MPI_Allreduce` to compute the sum of all local dot product results. This global collective communication is expensive and scales badly as the process number increases.

We would like to use ULT to overlap this communication with computation. At first glance, `DDOT(r, r, normr)` happens at the end of the iteration, and there are no other computations to overlap with. To find a potential overlap, we need to "think between iterations". The end of an iteration i is also the beginning of the next iteration $i+1$. We may be able to find some potential computation to overlap with `DDOT(r, r, normr)` at the beginning of the loop. However, the output $normr$ is used in the conditional expression of the loop to determine whether the loop should continue. To do communication hiding between iterations, therefore, we have to delay the decision of this conditional expression. In other words, we will do `MG` speculatively regardless of the value of $normr$ and decide whether to break out the loop after `MG`. Listing 4 shows our HPCG algorithm using MPI+ULT. In line 20, the original `DDOT` is wrapped as a ULT function `ult_fn_dotprod` so it can be executed asynchronously in a ULT named $thread$. The parameters of `DDOT` are passed to `ult_fn_dotprod` in a user-defined $param$ data structure. The $thread$ ULT is joined at line 4 in the next iteration to determine whether the residual computed in the last loop has already met the requirement. Then `ult_join` is put after `MG` so `ult_fn_dotprod` can overlap with `MG`. The overlap is possible because both `MG` and `ult_fn_dotprod` have

```
for (int k = 1; k <= max_iter; k++) {
  MG(A, r, z);
  if (k > 1) {
    ult_join(thread);
    normr = sqrt(param.result);
    if (normr <= tolerance)
      break;
  }
  if (k == 1) {
    WAXPBY(1.0, z, 0.0, z, p);
    DDOT(r, z, rtz);
  } else {
    DDOT(r, z, rtz);
    WAXPBY(1.0, z, beta, p, p);
  }
  SpMV(A, p, Ap);
  DDOT(p, Ap, pAp);
  WAXPBY(1.0, x, alpha, p, x);
  WAXPBY(1,0, r, -alpha, Ap, r);
  ult_fork(ult_fn_dotprod, &param, &thread
      );
  ult_yield();
}
```

Listing 4: HPCG using MPI+ULT.

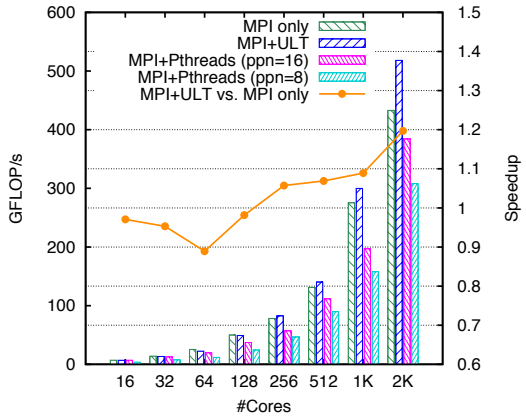
read-only access to r, and MG does not change the value of $normr$. In our new algorithm, we see that one advantage of MPI+ULT is that it can be easily applied to an existing algorithm without making big changes to the original structure of the code. Note that we can also achieve the overlap with MPI nonblocking collective function MPI_Iallreduce, but will need to modify the function DDOT to facilitate the change.

Fig. 8a shows the performance of HPCG using MPI-only, MPI+ULT, and MPI+Pthreads. With a small number of cores, we do not see any benefits of MPI+ULT compared with MPI-only because MPI_Allreduce is fast at a small scale so there is not much to overlap. For example, with 16 cores, DDOT counts only 0.62% of the total execution time, whereas with 2,048 cores, it counts 36.8% of the total time. We even see a little performance loss with the MPI+ULT version compared with the MPI-only version when the core number is smaller than 128 because of the scheduling overhead introduced by the ULT library. When the core number increases, however, the benefit of communication hiding begins to appear. With 2,048 cores, HPCG using MPI+ULT shows a performance improvement of 19.8% compared with the MPI-only version. The performance is gained from reducing the overhead of MPI_Allreduce with ULT. Fig. 8b shows the execution time breakdown of HPCG using MPI-only and MPI+ULT. On 2,048 cores, the DDOT time has been reduced 56.8% from 65.9 seconds to 28.7 seconds (the first two DDOTs remain unchanged). At the same time, the MG time has slightly increased 8.7% from 91.4 seconds to 99.4 seconds. There are three DDOT calls in HPCG main loop. The last DDOT overlaps with MG, and its execution time is counted inside MG. The increased MG time comes from two factors. First, the delayed DDOT slows MG down a little as it competes with MG to use MPI runtime for communication. But the overhead is moderate because the cost of MPI_Allreduce using ULT is relatively small (cf. Fig. 5). Second, because the decision of $normr$ has been delayed, one additional MG is called compared to the original MPI-only implementation. As long as the number of iterations is big, one additional MG is acceptable.

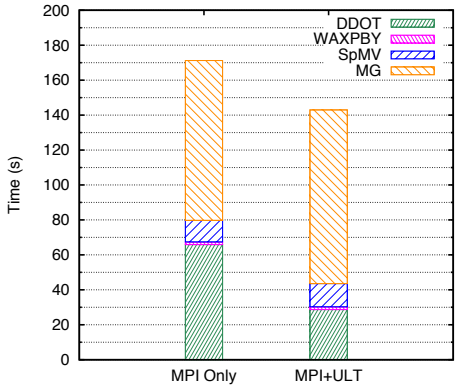
In the "MPI+Pthreads" version in Fig. 8a, ult_fork and ult_join are replaced by pthread_create and pthread_join, and ult_yield is removed. Compared with MPI-only, the runtime overhead of MPI+Pthreads is higher because it needs the thread level MPI_THREAD_MULTIPLE, where MPI calls are protected by locks. The lock used to protect MPI critical sections will cause contention between different threads, thus leading to performance loss. In Fig. 8a, we show the results of testing two configurations of "MPI+Pthreads" HPCG, one with 8 processes per node (ppn=8), the other with 16 processes per node (ppn=16). Since we created an additional thread T for doing MPI_Allreduce, each process has two POSIX threads. With ppn=8, we have 16 POSIX threads on a single node, with one thread for each CPU core. With ppn=16, we have 32 POSIX threads on a single node, and each CPU core is oversubscribed with two POSIX threads. Neither of these two configurations performs better than the MPI-only HPCG. With ppn=8, because thread T is created only for the MPI_Allreduce task. After it is finished, it will be destroyed. Thus, when there is no MPI_Allreduce task, only half of the CPU cores are utilized. With ppn=16, CPU is oversubscribed; when thread T is created, it competes with the main thread and causes lock contention. From the comparison, we see that ULT is better suited than POSIX threads for communication hiding. ULTs do not occupy additional hardware resources because they share the same kernel thread and get executed by context switching.

Hiding Neighborhood Communication: The second communication pattern in HPCG is neighborhood communication, which is used in the SpMV kernel. Modeled as a 3D 27-stencil grid (Fig. 7), each process has to do halo exchange with up to 26 neighbors before it computes its own submatrix. After the halo exchange, the kernel will do sparse matrix-vector multiplication on its submatrix. The halo exchange only exchanges the edge data; thus, the submatrix can be divided to two parts: internal and external. While the external part needs communication before computation, the internal part can be performed independently. In our MPI+ULT SpMV algorithm, we create a ULT for each neighbor to do halo exchange and the computation related to that neighbor. In order to partition the matrix by neighbors, the external matrix part has to change its format from compressed sparse row (CSR) to compressed sparse column (CSC). Also, another ULT is assigned to do internal computation only, with periodically yielding for MPI progress. In this way, the halo exchange with neighbors can be overlapped with the internal computation.

Fig. 9 shows the speedup of SpMV using MPI+ULT compared with MPI-only SpMV. As the proportion of halo exchange increases, the benefit of communication hiding becomes more obvious. On 4,096 cores, the performance of MPI+ULT SpMV improves 14.8% compared with the MPI-



(a) Performance of HPCG using MPI-only, MPI+ULT and MPI+Pthreads. Each MPI process uses a grid size of 128^3.



(b) HPCG time breakdown for 2,048 cores.

Fig. 8: HPCG performance.

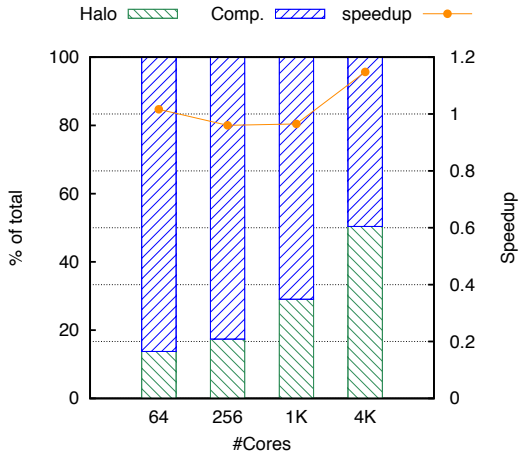


Fig. 9: SpMV speedup (MPI+ULT vs. MPI-only). Each MPI process has a grid size of 128^3.

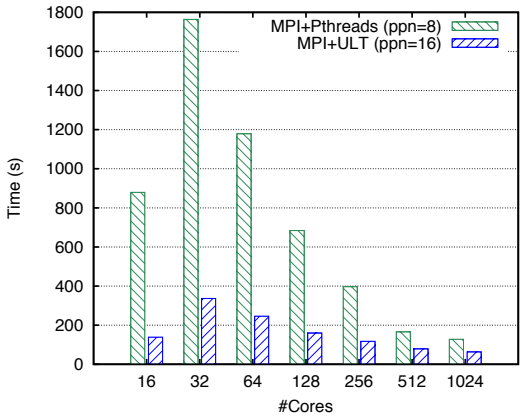


Fig. 10: Performance of SWAP-Assembler.

only SpMV.

B. SWAP Genome Assembler

SWAP-Assembler is a scalable and efficient genome assembler designed for processing massive sequence data on thousands of cores [16], [11]. It is one of the few parallel genome assemblers that run on distributed memory systems. The genome sequence data is distributed among different processes. In order to access the data from other processes and serve the requests from other processes, each process spawns two threads, one as a client and the other as a server, to communicate with each other. During the edge merge stage, because multiple processes can request the same read at the same time, SWAP uses a set of "Lock-Computing-Unlock" steps to operate on a read. If two processes request the same read at the same time, they will use a back-off algorithm to avoid collision. The server thread and the client thread can call MPI functions independently. However, because MPI functions are protected by a global lock[1], only one thread can enter the MPI function call at a time, which means the client thread and the server thread may spend much of their time competing the lock to enter the MPI function.

To reduce the thread overhead, we replace both server thread and client thread with ULTs, so the server ULT and client ULT can run on the same kernel thread. With this simple change, the MPI+ULT SWAP-Assembler can double the number of processes used on a node. Moreover, we can use `MPI_THREAD_ULT` instead of `MPI_THREAD_MULTIPLE` for this application, eliminating the use of locks for MPI function calls. Fig. 10 shows the performance of SWAP-Assembler with MPI+Pthreads and MPI+ULT. We performed a strong-scaling experiment with a synthetic sequence of 5 million reads, where each read contains 36 nucleotides. The result of MPI+Pthreads with ppn=16 (i.e., oversubscription) is not shown in the figure, because its lock contention is too heavy to allow the application to finish in a reasonable

[1]IBM MPI provides an option for better fine-grained lock support, but it is only available for Blue Gene systems.

time. MPI+ULT is between 2.0 and 6.3 times faster than MPI+Pthreads (ppn=8), depending on the number of cores used. With a smaller number of cores, SWAP-Assembler is more likely to encounter collision, so the overhead of Pthreads library is higher. By replacing Pthreads with ULT, the speedup of MPI+ULT over MPI+Pthreads is relatively higher for smaller number of cores. With 16 cores, the overhead of pthreads library is 25.4% of the total execution time while with 1,024 cores the overhead is 9.9%. By replacing pthreads with Qthreads, the overhead of the thread library is reduced to 8.2% with 16 cores and 4.4% with 1,024 cores.

VI. Related Work

Hybrid programming has long been a research topic with the popularity of SMP clusters [17]. A decade ago, the total number of CPU cores on a SMP node was still small, so that two-level parallelism was not obvious. As computer architectures moved toward many-core processors, however, the trend for hybrid programming became more prominent [18], [19], [20]. Thus it is considered as a promising programming model for future exascale systems [1]. Considerable research has been done in this direction. Most of the research focuses on how applications can use "MPI+X" model to improve the application performance [21]. Other research has focused on runtime improvement of the hybrid model [8], [9], with some investigations looking at better utilization of thread idle time and other investigations exploring ways to minimize lock contention. Our work focuses on using user-level threads to overlap communication.

MPI provides nonblocking calls to overlap communication with computation. Nonblocking point-to-point MPI calls have been added to the MPI standard since MPI-1.0. Recent work [6], [22] adds nonblocking collectives to the MPI-3.0 standard [5], extending the use of MPI nonblocking operations to collective communication patterns. However, some one-sided synchronization operations such as `MPI_Win_flush` still do not have a nonblocking equivalent to be used asynchronously. With MPI+ULT, one-sided synchronization calls can be easily overlapped by wrapping them in an ULT, overlapping them with useful computation. The use of ULT has been explored for task parallelism on shared-memory machines [23], [24], [25]. In this paper, we focus on overlapping communication and computation in distributed-memory systems. To make computation/communication overlap easier, Marjanović et al. proposed MPI/SMPSs [26], integrating MPI with a task-based programming model called SPMSs [27]. MPI/SPMSs focused on improving the programming model by extending C/Fortran programming languages with a set of pragmas/directives to easily generate tasks for communication. In our work, we focus on runtime improvement by effectively integrating ULT with MPI.

A rich body of literature exists on the conjugate gradient (CG) algorithm because of the importance of the problem. In CG, there are two communication patterns: global collective communication and neighborhood communication. Demmel [28] proposed overlapping the global reduction with computation algorithmically. With MPI+ULT, we can easily achieve this without big changes to the code structure of HPCG. Ghysels and Vanroose [29] recently proposed a pipelined CG that has only a single nonblocking reduction per iteration, but the method requires extra floating-point operations and significant change to the algorithm. Hoefler et al. [7] optimized the neighborhood communication in CG with libNBC [6] but did not optimize the global reduction. In our work, we have optimized both the global collective communication and neighborhood communication and successfully hidden them with MPI+ULT. Recent studies of HPCG on Intel Phi [30] and Tianhe [31] focus mainly on shared-memory optimizations. In our paper, on the other hand, we focus on improving multinode scalability by hiding the communication in HPCG. Also, one can optimize communication by offloading it to hardware. Kandalla et al. [32] offloaded the `MPI_Iallreduce` operation on InfiniBand clusters, successfully improving PCG performance 21%. With MPI+ULT, we have been able to achieve comparable benefits with runtime improvement.

VII. Conclusion and Future Works

In this paper we present MPI+ULT, a new approach to support asynchronous MPI communication using ULT. Our runtime not only supports hiding the communication of MPI point-to-point and collective calls but also can be used for overlapping MPI one-sided synchronization calls. Compared with other runtime systems, MPI+ULT provides several advantages. With ULT, overlap of communication and computation can be achieved easily at a low cost. We have evaluated our runtime with various microbenchmarks and applications. Experiments show that MPI+ULT can help applications hide different communication patterns such as global collective communication and neighborhood communication and improves the application performance up to 19.8% on 2,048 cores. Also, by replacing pthreads with ULT in a parallel genome assembly application, we have improved its performance by 2.0 to 6.3 times.

MPI+ULT can be further investigated in several directions. First, we plan to integrate both ULT and kernel thread with MPI in the future. Each kernel thread will fork multiple ULTs to overlap computation and communication. Second, we will further improve communication hiding by reducing the ULT library overhead. Third, we have seen benefits from MPI one-sided micro-benchmarks that ULT can be effectively used to overlap MPI one-sided synchronization. This potential has not been fully explored, and more MPI one-sided applications may benefit from MPI+ULT. Finally, the conjugate gradient method is a building block for Krylov subspace methods and many partial differential equation applications. The improvement of HPCG in this paper can be applied to more applications that follow the same communication pattern.

Acknowledgments

This material was based upon work supported by the U.S. Department of Energy, Office of Science, Office of

Advanced Scientific Computing Research, under Contract DE-AC02-06CH11357. We gratefully acknowledge the computing resources provided on Blues, a high-performance computing cluster operated by the Laboratory Computing Resource Center at Argonne National Laboratory.

REFERENCES

[1] J. Dongarra *et al.*, "The international exascale software project roadmap," *Int. J. High Perform. Comput. Appl.*, vol. 25, no. 1, pp. 3–60, feb 2011.

[2] T. Saif and M. Parashar, "Understanding the behavior and performance of non-blocking communications in mpi," in *Euro-Par 2004 Parallel Processing*, ser. Lecture Notes in Computer Science, M. Danelutto, M. Vanneschi, and D. Laforenza, Eds. Springer Berlin Heidelberg, 2004, vol. 3149, pp. 173–182.

[3] C. Bell, D. Bonachea, R. Nishtala, and K. Yelick, "Optimizing bandwidth limited problems using one-sided communication and overlap," in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, April 2006.

[4] T. S. Abdelrahman and G. Liu, "Cluster computing," R. Buyya and C. Szyperski, Eds. Commack, NY, USA: Nova Science Publishers, Inc., 2001, ch. Overlap of Computation and Communication on Shared-memory Networks-of-workstations, pp. 35–45.

[5] M. P. I. Forum, *MPI: A Message-Passing Interface Standard (Version 3.0)*, MPI Forum Std., Spetember 2012. [Online]. Available: http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf

[6] T. Hoefler, A. Lumsdaine, and W. Rehm, "Implementation and performance analysis of non-blocking collective operations for MPI," in *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, ser. SC '07. New York, NY, USA: ACM, 2007, pp. 52:1–52:10.

[7] T. Hoefler, P. Gottschling, A. Lumsdaine, and W. Rehm, "Optimizing a Conjugate Gradient Solver with Non-Blocking Collective Operations," *Elsevier Journal of Parallel Computing (PARCO)*, vol. 33, no. 9, pp. 624–633, Sep. 2007.

[8] M. Si, A. J. Peña, P. Balaji, M. Takagi, and Y. Ishikawa, "MT-MPI: multithreaded MPI for many-core environments," in *2014 International Conference on Supercomputing, ICS'14, Muenchen, Germany, June 10-13, 2014*, 2014, pp. 125–134.

[9] A. Amer, H. Lu, Y. Wei, P. Balaji, and S. Matsuoka, "MPI+threads: Runtime contention and remedies," in *Proceedings of the 20th ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPoPP '15, 2015.

[10] "HPCG project homepage." [Online]. Available: https://software.sandia.gov/hpcg

[11] J. Meng, B. Wang, Y. Wei, S. Feng, and P. Balaji, "SWAP-assembler: scalable and efficient genome assembly towards thousands of cores," *BMC Bioinformatics*, vol. 15, no. Suppl 9, pp. –2, 2014.

[12] D. E. Knuth, *The Art of Computer Programming*. Addison-Wesley, 1997, vol. 1.

[13] "The Go programming language." [Online]. Available: http://golang.org/

[14] K. Wheeler, R. Murphy, and D. Thain, "Qthreads: An api for programming with millions of lightweight threads," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, 2008, pp. 1–8.

[15] P. Balaji, D. Buntinas, D. Goodell, W. D. Gropp, and R. Thakur, "Fine-grained multithreading support for hybrid threaded mpi programming," *Int. J. High Perform. Comput.Appl.*, vol. 24, pp. 49–57, Feb 2010.

[16] J. Meng, J. Yuan, J. Cheng, Y. Wei, and S. Feng, "Small world asynchronous parallel model for genome assembly," in *Network and Parallel Computing*, J. Park, A. Zomaya, S.-S. Yeo, and S. Sahni, Eds. Springer Berlin Heidelberg, 2012, vol. 7513, ch. Lecture Notes in Computer Science, pp. 145–155.

[17] F. Cappello and D. Etiemble, "Mpi versus mpi+openmp on the ibm sp for the nas benchmarks," in *Supercomputing, ACM/IEEE 2000 Conference*, Nov 2000, pp. 12–12.

[18] H. Gahvari, W. Gropp, K. Jordan, M. Schulz, and U. Yang, "Modeling the performance of an algebraic multigrid cycle using hybrid mpi/openmp," in *Parallel Processing (ICPP), 2012 41st International Conference on*, Sept 2012, pp. 128–137.

[19] P. D. Mininni, D. Rosenberg, R. Reddy, and A. Pouquet, "A hybrid mpi–openmp scheme for scalable parallel pseudospectral computations for fluid turbulence," *Parallel Computing*, vol. 37, no. 6–7, pp. 316 – 326, 2011.

[20] R. Rabenseifner, G. Hager, and G. Jost, "Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes," in *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on*, Feb 2009, pp. 427–436.

[21] X. Wu and V. Taylor, "Performance characteristics of hybrid mpi/openmp implementations of nas parallel benchmarks sp and bt on large-scale multicore supercomputers," *SIGMETRICS Perform. Eval. Rev.*, vol. 38, no. 4, pp. 56–62, Mar. 2011.

[22] T. Hoefler and A. Lumsdaine, "Overlapping communication and computation with high level communication routines," in *Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid*, ser. CCGRID '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 572–577.

[23] S. L. Olivier, A. K. Porterfield, K. B. Wheeler, and J. F. Prins, "Scheduling task parallelism on multi-socket multicore systems," in *Proceedings of the 1st International Workshop on Runtime and Operating Systems for Supercomputers*, ser. ROSS '11. New York, NY, USA: ACM, 2011, pp. 49–56.

[24] J. Nakashima and K. Taura, "Massivethreads: A thread library for high productivity languages," in *Concurrent Objects and Beyond*, ser. Lecture Notes in Computer Science, G. Agha, A. Igarashi, N. Kobayashi, H. Masuhara, S. Matsuoka, E. Shibayama, and K. Taura, Eds. Springer Berlin Heidelberg, 2014, vol. 8665, pp. 222–238.

[25] B. Barrett, J. Berry, R. Murphy, and K. Wheeler, "Implementing a portable multi-threaded graph library: The mtgl on qthreads," in *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, May 2009, pp. 1–8.

[26] V. Marjanović, J. Labarta, E. Ayguadé, and M. Valero, "Overlapping communication and computation by using a hybrid mpi/smpss approach," in *Proceedings of the 24th ACM International Conference on Supercomputing*, ser. ICS '10. New York, NY, USA: ACM, 2010, pp. 5–16.

[27] J. Perez, R. Badia, and J. Labarta, "A dependency-aware task-based programming environment for multi-core architectures," in *Cluster Computing, 2008 IEEE International Conference on*, Sept 2008, pp. 142–151.

[28] J. W. Demmel, M. T. Heath, and H. A. van der Vorst, "Parallel numerical linear algebra," *Acta Numerica*, vol. 2, pp. 111–197, 1 1993.

[29] P. Ghysels and W. Vanroose, "Hiding global synchronization latency in the preconditioned conjugate gradient algorithm," *Parallel Computing*, vol. 40, no. 7, pp. 224 – 238, 2014, 7th Workshop on Parallel Matrix Algorithms and Applications.

[30] J. Park *et al.*, "Efficient shared-memory implementation of high-performance conjugate gradient benchmark and its application to unstructured matrices," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14, 2014. [Online]. Available: http://dx.doi.org/10.1109/SC.2014.82

[31] X. Zhang, C. Yang, F. Liu, Y. Liu, and Y. Lu, "Optimizing and scaling hpcg on tianhe-2: Early experience," in *Algorithms and Architectures for Parallel Processing*, ser. Lecture Notes in Computer Science, 2014, vol. 8630, pp. 28–41.

[32] K. Kandalla *et al.*, "Designing non-blocking allreduce with collective offload on infiniband clusters: A case study with conjugate gradient solvers," in *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, May 2012, pp. 1156–1167.