An Efficient Inter-node Communication System with Lightweight-thread Scheduling

1st Takuya Fukuoka
The University of Tokyo
fukuoka@eidos.ic.i.u-tokyo.ac.jp

2nd Wataru Endo
The University of Tokyo
wendo@eidos.ic.i.u-tokyo.ac.jp

3rd Kenjiro Taura
The University of Tokyo
tau@eidos.ic.i.u-tokyo.ac.jp

***Abstract*—In the era of multi-/many-core processors, there are increasing needs for middleware of high-performance computing to exploit both inter-node and intra-node parallelism. To overlap communication and computation efficiently, many studies have focused on MPI+ULT, a combination of MPI for inter-node parallelism and user-level threads (ULTs) for intra-node parallelism. However, there are mainly two problems in the existing MPI+ULT implementations. First, the use of MPI_THREAD_MULTIPLE to invoke MPI functions from multiple threads causes a performance bottleneck. Second, some MPI+ULT systems focus on the use of non-blocking communication and programmers have to manage both the start and the end of communication explicitly. To solve these problems, we introduce a high-performance MPI+ULT implementation MPI+myth. MPI+myth focuses on implicit overlapping of communication and computation without any code modifications to the applications. Furthermore, it can avoid the overhead of multi-threaded MPI invocations using a communication dedicated thread and adopts a new scheduling technique which achieves efficient load balancing by avoiding a situation in which a core is occupied by blocking ULTs. In the evaluation, we demonstrate significant performance improvement compared with the existing hybrid programming methods using several microbenchmarks and one mini application miniFE. In addition, we illustrate that MPI+myth has the potential to overlap communication and computation and our new ULT scheduling technique can achieve load balancing more efficiently than existing ULT scheduling techniques.**

***Index Terms*—MPI, user-level thread, thread scheduling, overlap, implicit load balancing**

I. Introduction

The Dennard Scaling [1], which states that the speedup and low power consumption occurs as the MOSFETs get smaller, finished and it entered the era of many cores in one chip. In order to achieve exascale computing systems, it is necessary to adopt a system which consists of millions of nodes each of which contains thousands of cores [2], [3]. Accordingly, there are increasing needs for parallel software technology to combine inter-node and intra-node parallelism efficiently.

Although both inter-node parallelism and intra-node parallelism have been defined by MPI since its early version, in recent years, it is increasingly popular to use MPI for inter-node parallelism and shared memory for intra-node parallelism. Especially, OpenMP [4] is a de-facto standard interface for shared-memory programming because of the simplicity of parallelization by annotating sequential codes with pragma directives.

However, there is a problem in existing hybrid parallelization method simply combining them naively. As the phases of communication and computation are separated, it is difficult to overlap communication and computation. In this case, the communication phase is executed only by one thread, which misses a chance to utilize the computation resources available in the middle of communication. Although the use of non-blocking communication can achieve overlapping communication and computation, it causes poor productivity of programmers.

In order to solve this problem, many studies have focused on MPI+ULT, a combination of MPI for inter-node parallelism and ULTs for intra-node parallelism [5]–[8]. ULT stands for a user-level thread, which is distinguished from a kernel-level thread (KLT) provided by an operating system. ULTs are lightweight compared to KLT and thread creation and context switching can be done with small overhead.

MPI+ULT can achieve communication-computation overlap with taking advantage of the flexibility of ULTs. In order to do this, some systems divide the cores into a communication-dedicated core and computation-dedicated ones in advance. In this case, when the communication occurs, the runtime cut it out as one ULT and force a communication-dedicated core to process it. Other systems are equipped with the mechanism to detect blocking of the communication and make the blocked ULT switch to another ULT immediately. This mechanism keeps the cores always busy and saves the waiting time until the communication finishs.

In addition to communication-computation overlap, MPI+ULT systems usually provide dynamic and automatic load balancing mechanism called work stealing. When a core has completed processing its own ULTs, it can steal a ULT from a busy core which has many ULTs to process. Owing to work stealing, even when an imbalance of tasks in cores occurs, it is possible to utilize computational resources effectively. MPI+ULT systems are effective in a certain application in which computational tasks with various granularity appear irregularly and communication and computation dependency becomes complex.

Although MPI+ULT systems are beneficial as described above, there is no implementation which can withstand for practical use. The problem of existing MPI+ULT systems can be divided into two kinds. First, the performance is degraded by the bottleneck in guaranteeing the thread safety. In order

 DOI: 10.1109/HPCC/SmartCity/DSS.2019.00103

to call MPI functions from multiple threads simultaneously, the thread level has to be set at `MPI_THREAD_MULTIPLE`, but it is known to perform poorly. As we describe in Section II-A2, in order to guarantee thread safety of MPI, it is not enough to acquire the lock at the start of MPI function and release it at the end of it. Because of the difficulty in guaranteeing the thread safety of MPI, MPI+ULT systems with `MPI_THREAD_MULTIPLE` cannot avoid heavy overhead inevitably. Second, some implementations require explicit dependency management for communication-computation overlap. This implementation focuses on the use of non-blocking communication and in this case, programmers have to manage both the start and the end of communication explicitly. While it can improve the performance with detailed tuning, it makes codes complex and reduces the productivity of programmers. TABLE I is detailed comparison of existing MPI+ULT implementations.

In this paper, we introduce a high-performance MPI+ULT implementation MPI+myth. MPI+myth is implemented using a lightweight thread library MassiveThreads [9] and it focuses on implicitly overlapping communication and computation without any code modifications to the applications. MPI+myth can avoid the overhead of invocation of MPI from multiple threads using the method Software Offloading [10]. In addition to this, MPI+myth adopts new scheduling technique to remove blocked ULTs from the ready queue. This scheduling technique achieves efficient load balancing by avoiding a situation in which a core is occupied by blocking ULTs.

To summarize the main contributions, this paper

1) proposes MPI+ULT implementations MPI+myth, which is equipped with two properties described below.
   a) It can avoid the overhead of the MPI invocations from multiple threads
   b) It does not require explicit description for overlapping communication and computation
2) proposes a new ULT scheduling technique called "uncond" and shows that it achieves efficient load balancing.

The rest of the paper is organized as follows. Section II provides the background information on MPI and ULT. Next, we detail the implementation of MPI+myth comparing with other similar techniques in Section III. In Section IV, we evaluate MPI+myth using several microbenchmarks and one mini application miniFE. Through these evaluations, we show that MPI+myth performs better than existing hybrid programming systems such as MPI+Argobots [11] and MPI+Pthreads. In addition to this, we illustrate that MPI+myth can overlap communication and computation and our new ULT scheduling technique achieves load balancing more efficiently than existing ULT scheduling technique. In Section V, we focus on the related work of hybrid programming methods, especially the systems which combine MPI and ULT library. Finally, Section VI draws the conclusions and future work of MPI+myth.

[1]In MPI/SMPSs, communication functions are invoked only from a main thread and there can be no situation in which multiple threads call MPI functions simultaneously.

```
1 MPI_Request req;
2 int flag;
3 MPI_Isend(..., &req);
4 computation_related_to_communication();
5 do{
6   MPI_Test(req, &flag, ...)
7 }while(!flag);
8 computation_not_related_to_communication();
```

Fig. 1. The pseudocode of a non-blocking communication. The computation which requires the result of communication is conducted after the done flag is set by MPI_Test.

Process 0		Process 1	
Thread 0	Thread 1	Thread 0	Thread 1
MPI_Recv (src=1)	MPI_Send (dest=1)	MPI_Recv (src=0)	MPI_Send (dest=0)

Fig. 2. An example causing deadlock when an MPI function is called in multiple threads

II. Background

A. Message Passing Interface (MPI)

1) Blocking and Non-blocking Communication: The functions like `MPI_Send` or `MPI_Recv` are called blocking communication functions because once the communication begins, they are blocked until the end of the communication. When one process issues a blocking function, and the other process is not ready to issue a corresponding function, all the issued processes can do is wait for the partner process to be ready. In order to meet the demand for using this waiting time effectively and overlapping communication and computation, non-blocking communication functions are implemented in MPI. Non-blocking communication functions return immediately with a request object which informs the end of the communication later. The example pseudocode is in Fig. 1.

Although non-blocking communication is prevalent as a means of overlapping communication and computation, it has some problems. One of them is that it degrades the programmer's productivity. As described above, the use of non-blocking communication requires two operations: the start of the communication and the end of the communication, which requires keeping track of the dependency relationship between communication and computation. However, it can be difficult especially when the scale of an application becomes large. Another problem is that the call of a non-blocking function itself does not force the progress of communication. When `MPI_Isend` is called before the issue of corresponding `MPI_Irecv`, `MPI_Isend` do not initialize the data transfer. In this situation, it was not until `MPI_Test` is called that the data transfer is initialized, and this can cause the failure of overlapping.

2) Thread Safety and Thread Level of MPI: There are mainly two problems to build a thread-safe MPI implementation. First, the mutual exclusion, that is, to acquire the lock at the start of MPI function and release it at the end of it, is needed to protect the MPI resources.

In the example of Fig. 2, each processes issues one `MPI_Send` in thread 0, one `MPI_Recv` in thread 1 targeting

TABLE I
DETAILED COMPARISON OF EXISTING MPI+ULT IMPLEMENTATIONS WITH MPI+MYTH

	Avoid Overhead of `MPI_THREAD_MULTIPLE` by the Use of Communication-dedicated Worker	Implicit Dependency Management for Overlapping Communication and Computation
MPI/SMPSs [5]	unsupported [1]	no
HCMPI [6]	yes	no
MPIQ [7]	no	yes
MPI+ULT [8]	no	yes
MPI+myth	yes	yes

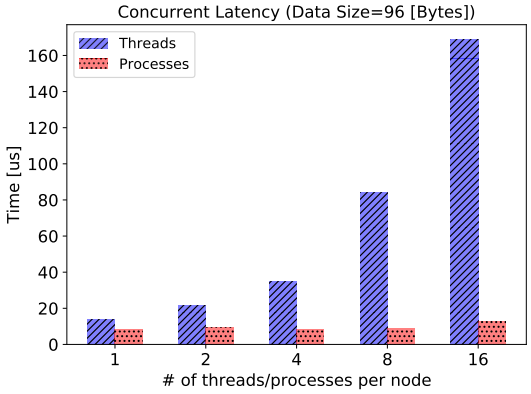

Fig. 3. Concurrent Latency of MPICH on Reedbush-U. While the latency is almost constant with the increase of the number of processes (right bar), the latency increases with the increase of the number of threads (left bar).

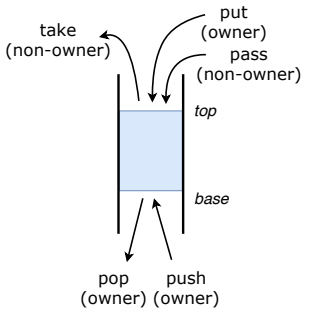

Fig. 4. A ready queue of MassiveThreads

another process. Let us assume that these functions acquire the lock of MPI resources at the function call and release it at the function return. Because they are blocking functions, each function keeps waiting for the issue of the corresponding function in the other process. If thread 0 acquires the lock first in both processes, they issue `MPI_Send` and block and never reach the call of `MPI_Recv`, which causes a deadlock, that is, a suspension of the whole system. This is only one example of the obstacles toward achieving free MPI invocations in multiple threads, and Group and Thakur detailed these problems further in [12].

Because it is expensive to fully assure the thread safety as mentioned above, there are four MPI thread levels for switching the support of thread safety, and only `MPI_THREAD_MULTIPLE` allows multiple threads to call MPI functions simultaneously. Because the implementation of `MPI_THREAD_MULTIPLE` has to overcome the difficulties mentioned above, there is no support for it in some implementations, or even if an implementation exists, it performs poorly. As a preliminary evaluation of multi-threaded MPI, we measured the concurrent latency of MPICH [13] on Reedbush-U [14] shown in Fig. 3. The left bars are the results with multiple threads in one node with `MPI_THREAD_MULTIPLE`, and the right bars are with multiple processes in one node. This graph shows that as the number of threads increases, the performance degrades with the use of `MPI_THREAD_MULTIPLE`.

B. User-level thread (ULT)

A thread (or a "thread of execution") can be implemented as either a kernel-level thread (KLT) or a user-level thread (ULT). One of the examples of KLTs is Pthreads, and threads in KLTs are scheduled by an operating system. On the other hand, ULTs are threads implemented in user space and multiple ULTs can be mapped into one KLT.

To exploit intra-node parallelism, the use of ULTs has two advantages compared to that of KLTs. First, ULTs are more lightweight and thread creation and context-switching can be done at low cost because they can avoid the invocations of system calls whose overheads are known to be heavy. Second, ULTs are more flexible with scheduling because they are not interfered by the preemption. Most of ULT libraries are equipped with work stealing for efficient scheduling. There are numerous exmples of ULT libraries such as MassiveThreads [9], Argobots [15], Qthreads [16], Habanero-Java [17], Cilk [18] and TBB [19].

A ULT library usually generates multiple KLTs and binds each KLT to one core, and multiple ULTs are executed in one KLT concurrently. The term "worker" is defined as one KLT which executes multiple ULTs, and each worker has its own ready queue where executable ULTs can be enqueued and dequeued. When a worker has no ULT in its ready queue, it can steal a ULT from other workers in whose queue there are many ULTs.

1) Ready Queue of MassiveThreads: A ready queue of MassiveThreads is double-ended queue implemented as an array. As Fig. 4 shows, each queue keeps track of two indices, *base* and *top*, and it can perform the following five operations:

- *pop* is an owner function[2] which extracts one ULT from *top* direction of the queue.
- *push* is an owner function which inserts one ULT into *top* direction of the queue.
- *put* is an owner function which inserts one ULT into *base* direction of the queue.
- *pass* is a non-owner function which inserts one ULT into *base* direction of the queue.
- *take* is a non-owner function which extracts one ULT from *base* direction of the queue.

Before a queue performs an operation accessing *base*, it has to take the lock associated with the queue. When the queue takes operations accessing *top*, it does not have to take the lock except in a situation that the number of ULTs in the queue is less than two.

When a new ULT is spawned, the runtime stores the thread context to the ULT which was originally executed, *push*es it to the queue, and begins to execute the new ULT immediately. When the execution of a ULT completes, the runtime *pop*s a ULT from the queue if it exists. If there is no ULT in the queue, it tries to steal a ULT from another worker's queue using *take*. If the steal succeeds, the worker executes the stolen ULT immediately, and if it fails, it tries to steal again.

2) Uncond API of MassiveThreads: MassiveThreads recently implemented with a new mechanism called "uncond" to reduce the scheduling overhead. `myth_uncond_wait` is a function which blocks the ULT and excludes it from the ready queue. It takes as an argument an uncond object, which is an opaque structure defined internally in MassiveThreads. When a ULT calls `myth_uncond_wait`, it suspends its execution and removes itself from the ready queue. At the same time, the execution state of the ULT is stored in the uncond object.

`myth_uncond_signal` is a function which awakes the ULT which was blocked by `myth_uncond_wait`. It takes as an argument an uncond object which stores the execution state of the ULT to be awakened. When `myth_uncond_signal` is called, the ULT associated with the uncond object is inserted to the `top` of the ready queue to which the caller ULT belongs. That is, the queue performs *push* operation without taking the lock of the queue. If there is no ULT associated with the uncond object, `myth_uncond_signal` blocks and keeps monitoring the uncond object by a busy loop.

While these API functions are lightweight with no necessity of any extra mutex variables unlike condition variables, it is the user's responsibility to implement a means to resolve the race condition. Therefore, they are useful in such a situation that the event related to blocking a ULT always occurs before the event related to awakening the ULT.

C. Software Offloading

Software Offloading [10] is a useful method to avoid the overhead of `MPI_THREAD_MULTIPLE` by creating a communication-dedicated thread to which all of the MPI communication operations are offloaded. All of the invocations of the MPI functions are serialized in the dedicated thread. Because software offloading enables multiple threads to invoke the MPI functions at the same time, it can reduce the cost of explicitly putting communication together into a master thread and improve productivity.

[2]An owner function is a function which is executed only by the owner (= the worker) of the queue.

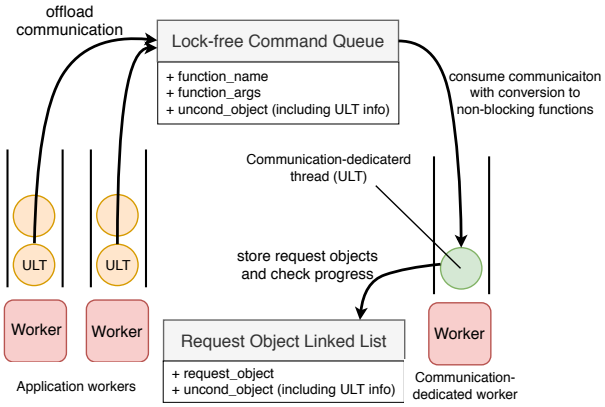

Fig. 5. The architecture of MPI+myth

Software Offloading needs two main infrastructures; a command queue and a communication-dedicated thread. The command queue contains entries each of which includes the name of the MPI function and its arguments. It is implemented as a lock-free object and can be enqueued by multiple threads and dequeued by the communication-dedicated thread concurrently. For convenience, all of the threads except for the dedicated thread are called application threads. When an application thread invokes an MPI function, the runtime replaces the invocation with an insertion into the command queue and the application thread waits for the completion by polling the done flag. After that, the communication-dedicated thread consumes the contents of the command queue, and when the communication finishes, it sets the done flag to notify the completion of the communication to the application thread. We call the system which adopts Software Offloading to Pthreads as MPI+Pthreads+Offloading.

III. Implementation of MPI+myth

In this section, we propose the design of our MPI+ULT implementation MPI+myth. Our system achieves implicit communication-computation overlap inside MPI blocking calls without any explicit code modifications to the applications. The core idea is applying Software Offloading to ULTs and adopting new ULT scheduling technique to remove a blocked ULT from the ready queue. Fig. 5 shows the architecture of MPI+myth. We used the implementation of a lock-free command queue presented in [20].

When Software Offloading is combined with ULT, one ULT is dedicated for communication and is bound to one worker. For convenience, we name this ULT as a communication-dedicated ULT and this worker as a communication-dedicated worker. We also name all of other ULTs as application

```
1  MPI_Send(){
2      initialize(uncond_object);
3      create_value_object("MPI_Send",
           arguments_list, uncond_object);
4      enqueue(value_object);
5      wait_for_communication(uncond_object);
6  }
```

Fig. 6. The implementation of MPI_Send in MPI+myth. The value object is created from a command name such as "MPI_Send", its arguments and an uncond object. For waiting for the completion of communication, `myth_uncond_wait` is used. This function suspends the execution of the ULT associated with the uncond object and excludes it from the ready queue.

```
1  offloading_thread(){
2      while(True){
3          command_node *p = dequeue();
4          if(p){
5              issue_nonblocking_call(p);
6          }else{
7              check_progress();
8          }
9      }
10 }
```

Fig. 7. The implementation of the communication-dedicated thread. When there exists no command in the command queue, this thread checks the completion of communications issued before.

ULTs and the workers except for the communication-dedicated worker as application workers. Compared with the case in which Software Offloading is not adopted, the number of application workers decreases by one.

When the application ULT invokes an MPI function in MPI+myth, the function name, its arguments and (a pointer to) the uncond object are enqueued into the command queue instead of calling the original MPI function. Fig. 6 is an example of implementing `MPI_Send`. In MPI+myth, blocking an application ULT is done by removing it from the ready queue instead of setting a done flag as in MPI+Pthreads+Offloading. When the communication finishes, restarting the ULT is conducted by re-inserting the ULT into the ready queue. The other method to block the ULT are described next.

A. Management of a Blocked Application ULT

There are two ways to manage a blocked application ULT. The first method is that the ULT keeps invoking `yield` until the communication finishes. The `yield` function is available in many threading libraries such as Pthreads, MassiveThreads and Argobots. In a ULT library, `yield` suspends the execution of the caller ULT and inserts its context to the ready queue. Inserting to the queue is implemented as *put* in MassiveThreads to avoid deadlocking. Every time the turn of the blocked ULT comes around, it checks the done flag, which is set by the communication-dedicated thread atomically, to detect the completion of the communication. The second method is that the application ULT is excluded from the ready queue when it blocks, and when the communication finishes, the communication-dedicated thread returns back the ULT to the ready queue. These operations are implemented by `uncond`, described in Section II-B2. For convenience, we call the former the yield method, and the latter the uncond method in this paper.

While many runtime systems adopt the yield method including MPI+Argobots [11], MPI+myth adopts the uncond method for several reasons. First, the uncond method can reduce the overhead of reading the done flag and invoking `yield`. Although the uncond method itself has its own overhead, the cost of reading flags and yielding ULTs exceeds the cost of the uncond method if the duration of communication is relatively long. Second, the yield method can prevent work stealing when all the ULTs in one ready queue are blocked. Since the work stealing is not conducted with no ULT in the ready queue, when there are any blocked ULTs in the ready queue, it does not steal ULTs from another queue. In this situation, blocked ULTs keep issuing `yield` call and does not do useful work until one of the blocked ULTs finishes communication.

B. Communication-dedicated Thread in MPI+myth

We detail the behavior of a communication-dedicated thread here. The communication-dedicated thread is implemented as a ULT, spawned in the initialization phase and fixed at one core. The main loop of the communication-dedicated thread is shown in Fig. 7. As the figure shows, depending on the condition of the command queue, it takes two different actions. When the command queue has commands to be executed, the dedicated thread dequeues one of them, which includes the command name, the arguments, and the uncond object. Then, the dedicated thread executes the command while replacing a blocking call with the corresponding non-blocking call. The request object returned by the non-blocking call is inserted into the linked list as an element with the uncond object.

When the command queue is empty, the dedicated thread traverses the linked list of the request elements and check the completions of all of the non-blocking calls with `MPI_Test`. `MPI_Test` not only checks the communication completion, but also forces the progress of the non-blocking calls. If one of the requests is completed, the request element is removed from the list, and `myth_uncond_signal` is invoked with the uncond object, which resumes the blocked ULT by inserting it into the ready queue.

IV. Evaluation

A. Experimental Environment

All the experiments were conducted in Reedbush-U [14], whose hardware configuration is in TABLE II. As an MPI library, we built MPICH [13] enabling the multi-threading option by ourselves.

B. Microbenchmarks

1) Concurrent Latency: First, concurrent latencies were measured using the test suite [21] developed at Argonne National Laboratory (ANL). In this measurement, we allocate two processes and each process spawns multiple threads. Each thread sends a message to the corresponding thread in the other process. In order to distinguish the messages, thread IDs are

TABLE II
HARDWARE CONFIGURATION OF REEDBUSH-U

Total number of nodes	420
Processor name	Intel Xeon E5-2695v4 (Broadwell-EP)
Number of processors	2
Number of cores	36
Memory capacity	256 GB
Memory bandwidth	153.6 GB/sec
Interconnections	InfiniBand EDR 4x (100 Gbps)

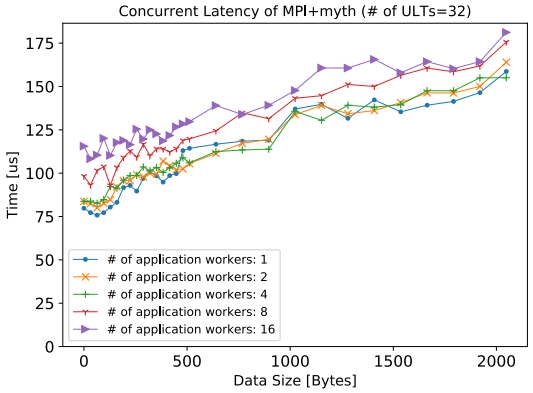

Fig. 8. Concurrent latency of MPI+myth changing the number of workers and fixing the number of ULTs at 32

passed to MPI as tags. We measure the average time for the communication completion.

Fig. 8 shows the concurrent latency of MPI+myth with different message sizes. The number of ULTs was fixed at 32 and the number of application workers was changed. As the graph shows, the larger the number of workers was, the longer the latency was observed. This is due to the contention of the command queue, which adopts CAS instructions.

Fig. 9 illustrates the comparison of concurrent latency between MPI+myth and MPI+Argobots [11]. MPI+Argobots, developed at ANL, is one of the existing MPI+ULT implementations. In the case of MPI+Argobots, we fixed the number of workers at 2, and in the case of MPI+myth, we tested the

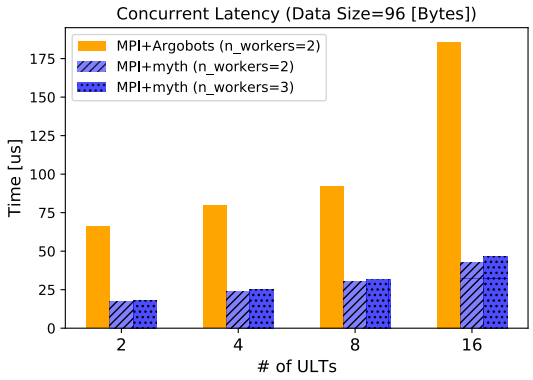

Fig. 9. Comparison of concurrent latency between MPI+Argobots and MPI+myth. An application worker is a worker except for the communication-dedicated thread.

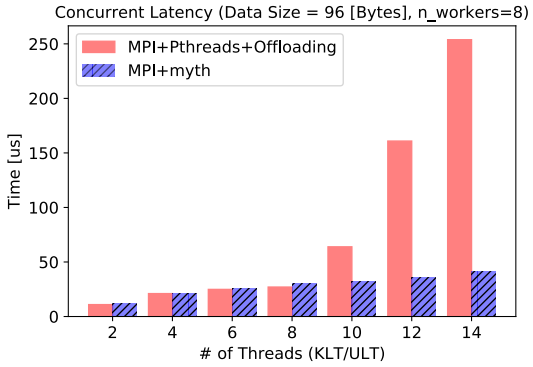

Fig. 10. Comparison of concurrent latency between MPI+Pthread+Offloading and MPI+myth.

number of workers at either 2 or 3. `n_workers` means the number of workers including both application workers and a communication-dedicated worker. The concurrent latency of MPI+myth becomes longer as the number of ULTs increases because the number of MPI functions processed in the dedicated worker increases. Compared with MPI+Argobots, MPI+myth with 2 workers achieved performance improvement by 3.0 to 4.6 times and MPI+myth with 3 workers achieved performance improvement by 2.9 to 3.9 times depending on the number of ULTs. MPI+myth was able to avoid contention of MPI resources by creating a communication-dedicated thread while MPI+Argobots degraded performance due to the use of `MPI_THREAD_MULTIPLE`.

We have also implemented MPI+Pthreads+Offloading as explained in [10]. In this implementation, when a thread calls an MPI function and blocks, it waits for the communication completion by polling the done flag to be set by an offloading thread as described before. Fig. 10 shows the comparison of concurrent latency between our MPI+Pthread+Offloading implementation and MPI+myth. In this experiment, we fixed the number of workers at 8 and changed the number of threads including application threads and a communication-dedicated thread. When the number of application threads was under the number of workers, the latencies of both models were almost the same. However, as the number of application threads increased, the performance of MPI+Pthreads+Offloading model degraded. This is because of the difference of thread scheduling policy. In addition to this, threads are scheduled by lightweight threading in MPI+myth, but in MPI+Pthreads+Offloading, threads are scheduled by the OS incurring a high cost.

2) Communication-Computation Overlap: In order to measure the communication-computation overlap, we made a new microbenchmark. The pseudocode is in Fig. 11. In this benchmark, two MPI processes are generated and communicate with each other. Each MPI process spawns many ULTs, which execute a communication, a computation, and a communication again. When a ULT has a thread number $\mathrm{thread\_num}$, it targets the ULT in the other process assigned as not $\mathrm{thread\_num}$ but $\mathrm{nthreads} - \mathrm{thread\_num} - 1$. This

```
computation(){
    for (i=0; i<50; i++)
        for (j=0; j<MAT_SIZE; j++)
            A[j] = B[j] * C[j];
}

overlap(){
    if (mpi_rank == 0) {
        tag = thread_rank;
        MPI_Send(dest=1, tag=tag);
        computation();
        MPI_Recv(src=1, tag=tag);
    } else if (mpi_rank == 1){
        tag = nthreads - thread_rank - 1;
        MPI_Recv(src=0, tag=tag);
        computation();
        MPI_Send(dest=1, tag=tag);
    }
}
```

Fig. 11. The pseudocode executed in one ULT in the overlap benchmark

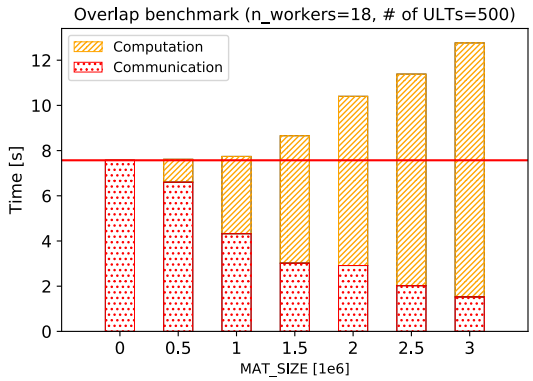

Fig. 12. Communication-computation overlap in overlap benchmark changing the amount of computation. Each bar shows the time of communication and computation and the red line shows the communication time when there is no overlap.

purpose is to increase the irregularity in the communication pattern. We fixed the message size of the communication at one megabyte and changed the amount of computation. The number of ULTs was fixed at 500 and they were distributed in 18 workers (17 application workers and 1 communication-dedicated worker) through work stealing.

The result of the overlap benchmark in MPI+myth is in Fig. 12. In addition to the time with both communication and computation, we measured the computation time alone by commenting out the communication calls in this benchmark. The communication time was calculated as the difference of the whole time and the computation time. The graph shows that with the increase of the amount of computation, the time for communication decreased, and it is observed that MPI+myth achieved communication-computation overlap. When the computation size was small, the whole time was almost constant with the change of the amount of communication because almost all the computation was able to be covered up by the communication.

3) Comparison of scheduling techniques: Fig. 13 shows the performance comparison of two scheduling techniques, the yield method and the uncond method. As we described in Section III-A, when the yield method is adopted, a blocked ULT checks whether the communication has finished and if it has not finished, the ULT is inserted into *base* of the original ready queue. On the other hand, when the uncond method is adopted, once the ULT is blocked, the ULT is removed from the ready queue until the communication finishes. In this experiment, the time of overlap benchmark was measured in the condition that the number of ULTs was fixed at 100, and we changed the number of workers from 2 to 16. It is noted that the number of workers in the figure includes one communication-dedicated worker and the number of application workers is one less than the number which the figure shows. The message size of communication was fixed at one megabyte which is the same as the previous measurement, and in computation part, MAT_SIZE was fixed at 4 million.

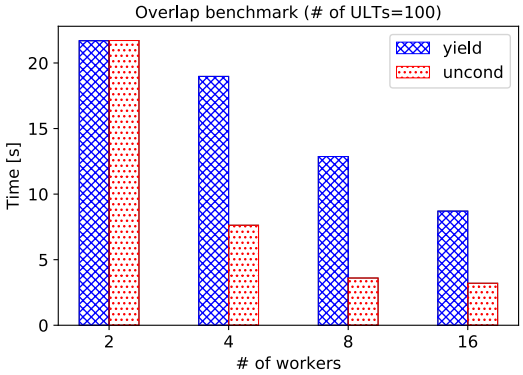

Fig. 13. The comparison between uncond method and yield method using overlap benchmark.

When there are two workers consisting of one application worker and one communication-dedicated worker, the results of two scheduling techniques were almost the same. However, as the number of workers increased, the uncond method performed better than the yield method. This is because the yield method cannot fully utilize computational resources when a worker has more than one ULT and all of them are blocked.

Fig. 14 shows how many computation parts each application worker processed using the yield method in the overlap benchmark with the number of ULT fixed at 100 and the number of workers fixed at 16. Because there is no preemption in ULT systems, the computation part of one ULT was processed in the same worker from the beginning to the end. The graph illustrates that worker 10 calculated far more parts than other workers. This load imbalance of computation occurred when a worker had more than one ULT and all of them were blocked. In this situation, all ULTs continued to call `yield` function until one of them became unblocked, which could lose the chance of utilizing computation resources. Such a situation can be avoided with the uncond method, and as Fig. 15 shows, the computation parts were evenly distributed into each worker.

C. Application Benchmark

miniFE is a mini-app from the Mantevo benchmark suite [22] which mimics the finite element generation, assembly,

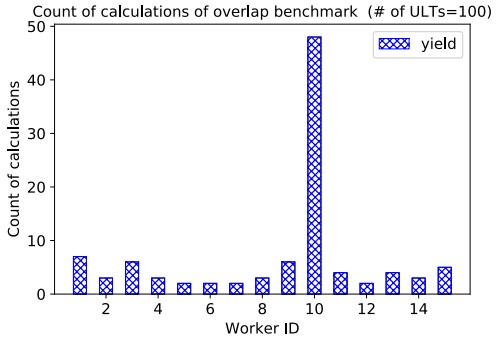

Fig. 14. The count of calculation tasks processed in each core with yield method

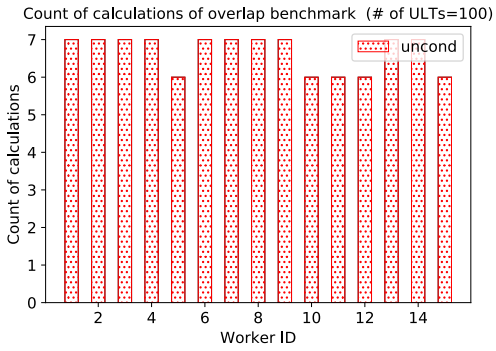

Fig. 15. The count of calculation tasks processed in each core with uncond method

and solution for an unstructured grid problem. This benchmark is written in C++ and parallelized by MPI. The conjugate gradient solver in miniFE is mainly composed of three parts; MATVEC, WAXPBY, DOT. MATVEC calculates a matrix-vector product, and it includes the exchange of the elements with other MPI processes. WAXPBY computes `alpha * x + beta * y`, where alpha and beta are scalars and x and y are vectors. It includes only a local vector computation and does not communicate. Finally, DOT calculates an inner product of two vectors. Each process first calculates the inner product of local vectors, and then issues MPI_Allreduce to get the sum.

In order to exploit the parallelism of multiple threads, we modified the original MATVEC code to spawn a communication thread per neighbor. We distributed two cores to each processes, and we selected the cores so that they were in the same NUMA node as possible. In this experiment, the problem size was fixed at 100^3, and we measured the execution time of MPI+myth and MPI+Pthreads with the change of the number of MPI ranks.

In Fig. 16, the left bars indicate the execution times of MPI+Pthreads and the right bars indicate those of MPI+myth. While the execution time of MPI+Pthreads hit the ceiling once the number of MPI rank exceeds 27, the result of MPI+myth showed the strong-scaling as the number of MPI ranks increased. With 72 MPI ranks, MPI+myth achieved performance improvement by 2.9 times. When we look in

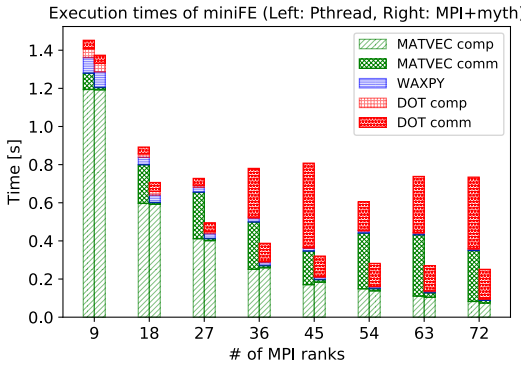

Fig. 16. The execution time of miniFE with 100^3 problem size. The left bars indicate the results of MPI+Pthreads and the right bars indicate those of MPI+myth.

more detail, it is observed that while the computation time was almost the same between two systems, MPI+Pthreads took more time to communicate than MPI+myth. In the case of MATVEC, the performance degradation of MPI+Pthreads was caused by the contention of communication resources. On the other hand, MPI+myth was able to avoid this problem by serializing the invocations of MPI functions. In the case of DOT, the difference of communication time was affected by the MATVEC communication time. The reason why is that the communication phase of DOT was composed of collective functions and the ending time of the communication was bound to the arrival of the lattermost process at this function. With the different number of threads created according to the process in MATVEC, the longer time of MATVEC communication caused the larger difference of the end time of communication in MATVEC, which also caused the larger difference of arrival of collective communication in DOT. Depending on the MPI rank count, MPI+myth was between 2.4 to 5.1 times faster than MPI+Pthreads in the communication phase.

V. Related Work

With the needs for flexible use of many cores in communication, the performance improvement of `MPI_THREAD_MULTIPLE` is an urgent task. In order to handle this problem, many studies are conducted in three different viewpoints; lock granularity [23], [24], lock ownership passing latency [25], and lock arbitration [26], [27].

There exists a large amount of work in the direction of avoiding the use of `MPI_THREAD_MULTIPLE`. As described before, Vaidyanathan et al. adopted an effective method called Software Offloading to avoid MPI function calls from multiple kernel-level threads [10]. This idea is being transported into the low-level communication with user-level threads [28]. MPI endpoints is a promising interface proposed by MPI Forum to enable multiple threads to communicate freely [29], [30].

Here, we introduce several hybrid programming systems combining MPI with user-level threading. Although all of them

are closely related to our research, they have problems respectively or their focuses are different from our implementation.

A. Hybrid MPI/SMPSs

Marjanović et al. proposed MPI/SMPSs approach to avoid the programming cost of using asynchronous communication primitives in communication-computation overlap [5]. The SMPSs is a task-based shared-memory programming model which defines tasks with inserting pragma directives above function declarations and manages dependency relationship of the variables using the information of input and output of the functions.

In hybrid MPI/SMPSs system, as many kernel-level threads as cores in the node are created for computation. In addition to this, a kernel-level thread is created for the communication, and each MPI call is encapsulated as one task and assigned to this communication thread. When the MPI call blocks, the communication thread yields the CPU to the computation thread. In order to avoid the deadlock, *highpriority* is set at the communication thread with `setpriority` system call.

This approach expects no nested parallelism and only supports the MPI invocation from the main stream, which leaves the work of serializing MPI call to programmers.

B. HCMPI (MPI + Habanero-C)

Chatterjee et al. proposed HCMPI, which is a hybrid programming system integrating MPI with Habanero-C (HC) [6]. HC is a task-based programming model based on Habanero-Java [17] and X10 [31]. While MassiveThreads adopts the fork-join model, it uses `async` clause and `finish` clause in task management. Tasks are created with `async` clause, executed asynchronously, and synchronized with `finish` clause.

For inter-node communication, an application calls a function prefixed with `HCMPI_` rather than an original MPI function. A HCMPI function internally creates a task containing only one corresponding MPI call, which is executed asynchronously in the communication-dedicated worker. The programmer has to explicitly manage synchronization of all HCMPI functions with `finish` clause regardless of whether they are blocking or non-blocking calls, which needs a significant rewrite of existing MPI applications.

C. MPIQ (MPI + Qthreads)

Stark et al. proposed MPIQ which combines a low-level task parallel runtime called Qthreads [16] with MPI to effectively handle fine-grained parallel communication and communication-computation overlap [7]. In this model, a blocking MPI call is internally converted to the corresponding non-blocking call, and the runtime keeps calling `yield` until the communication has finished. As the advantage of this model, they insisted that little rewrite of MPI application codes was needed. Application programmers do not need to specifically annotate MPI codes for MPIQ because MPIQ intercepts standard MPI calls.

This system experiences performance degradation with the MPI application which requires MPI invocation in multiple threads. As the system requires the same thread level as the original MPI application does, the free invocation of MPI is limited because of the large overhead of `MPI_THREAD_MULTIPLE`.

D. MPI+ULT

Lu et al. proposed MPI+ULT hybrid programming implementation to overlap communication and computation by enabling ULT context switching inside the MPI progress engine [8]. This model focuses on the specific situation, where one MPI process includes only one kernel-level thread and multiple ULTs in the kernel-level thread invokes MPI functions concurrently. For this situation, a new thread level `MPI_THREAD_ULT` is implemented to avoid the overhead of `MPI_THREAD_MULTIPLE`, and on the other hand, to enable concurrent invocation from multiple ULTs mapped into one kernel-level thread. In this implementation, the yield call is embedded in the progress engine of MPI to enable immediate detection of the stagnation of progress.

Because this model demands the mapping of one MPI process to one core, it has difficulty in the effective use of shared memory especially when the number of cores per CPU increases.

VI. Conclusion and Future Work

A. Conclusion

In this paper, we introduced a high-performance MPI+ULT implementation MPI+myth. MPI+myth focuses on implicit overlapping of communication and computation using blocking MPI functions without making codes complex. Our implementation adopts a communication-dedicated worker to avoid the overhead of `MPI_THREAD_MULTIPLE` and adopts a new ULT scheduling technique called the uncond method, which achieves efficient load balancing by removing a blocked ULT from the ready queue. In the evaluation, we demonstrated significant performance improvement compared with existing hybrid programming systems using several microbenchmarks and one mini application miniFE. In addition to this, we illustrated that MPI+myth has the potential to overlap communication and computation and that the uncond method can achieve load balancing more efficiently than the yield method.

B. Future Work

At the moment, MPI+myth degrades performance in NUMA architecture and this phenomenon is related to work stealing phase. When the stealing application worker and the communication-dedicated worker are on a different CPU, this stealing operation is at a higher cost than when they are on the same CPU. One of the solutions to this performance deterioration is that the implementation of `myth_uncond_wait` is fixed to insert unblocked ULT to the original ready queue instead of the ready queue of the worker which call `myth_uncond_wait`. While this implementation can reduce the work stealing across different CPUs and also improve

cache locality, a contention of queue lock can occur when unblocked ULT is inserted into *base* of the original ready queue with *put* operation. It is a future work to modify the implementation of uncond method and evaluate considering pros and cons as described above.

The implementation for non-blocking MPI functions in MPI+myth is also required. Although this system mainly focuses on blocking MPI functions, there are many applications which include non-blocking MPI functions.

Remote Memory Access (RMA) is the one-sided communication interface and the extension of MPI+myth for RMA functions is also future work.

ACKNOWLEDGEMENT

This work is partially supported by a project commissioned by the New Energy and Industrial Technology Development Organization (NEDO).

REFERENCES

[1] M. Bohr, "A 30 Year Retrospective on Dennard's MOSFET Scaling Paper," *IEEE Solid-State Circuits Newsletter*, vol. 12, no. 1, pp. 11–13, 2007.

[2] V. Sarkar, W. Harrod, and A. E. Snavely, "Software challenges in extreme scale systems," *Journal of Physics: Conference Series*, vol. 180, no. 1, p. 012045, jul 2009.

[3] J. Shalf, S. Dosanjh, and J. Morrison, "Exascale Computing Technology Challenges," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2011, vol. 6449 LNCS, pp. 1–25.

[4] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," Tech. Rep. 1, 1998.

[5] 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 - ICS '10*. New York, New York, USA: ACM Press, 2010, p. 5.

[6] S. Chatterjee, S. Tasirlar, Z. Budimlic, V. Cave, M. Chabbi, M. Grossman, V. Sarkar, and Y. Yan, "Integrating Asynchronous Task Parallelism with MPI," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE, may 2013, pp. 712–725.

[7] D. T. Stark, R. F. Barrett, R. E. Grant, S. L. Olivier, K. T. Pedretti, and C. T. Vaughan, "Early Experiences Co-Scheduling Work and Communication Tasks for Hybrid MPI+X Applications," in *2014 Workshop on Exascale MPI at Supercomputing Conference*. IEEE, nov 2014, pp. 9–19.

[8] H. Lu, S. Seo, and P. Balaji, "MPI+ULT: Overlapping communication and computation with user-level threads," in *Proceedings - 2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security and 2015 IEEE 12th International Conference on Embedded Software and Systems, H*. IEEE, aug 2015, pp. 444–454.

[9] J. Nakashima and K. Taura, "MassiveThreads: A Thread Library for High Productivity Languages," 2014, pp. 222–238.

[10] 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 *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '15*, 2015, pp. 1–12.

[11] "MPI+Argobots," https://wiki.mpich.org/mpich/index.php/MPI+Argobots, 2017.

[12] W. Gropp and R. Thakur, "Issues in Developing a Thread-Safe MPI Implementation," in *Recent Advances in Parallel Virtual Machine and* .... Springer, Berlin, Heidelberg, 2006, pp. 12–21.

[14] "Reedbush supercomputer system," https://www.cc.u-tokyo.ac.jp/en/supercomputer/reedbush/service, 2019.

[13] "MPICH," http://www-unix.mcs.anl.gov/mpi/mpich, 2019.

[15] S. Seo, A. Amer, P. Balaji, C. Bordage, G. Bosilca, A. Brooks, P. Carns, A. Castello, D. Genet, T. Herault, S. Iwasaki, P. Jindal, L. V. Kale, S. Krishnamoorthy, J. Lifflander, H. Lu, E. Meneses, M. Snir, Y. Sun, K. Taura, and P. Beckman, "Argobots: A Lightweight Low-Level Threading and Tasking Framework," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 3, pp. 512–526, mar 2018.

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

[17] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar, "Habanero-Java," in *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java - PPPJ '11*. New York, New York, USA: ACM Press, 2011, p. 51.

[18] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An Efficient Multithreaded Runtime System," *Journal of Parallel and Distributed Computing*, vol. 37, no. 1, pp. 55–69, aug 1996.

[19] G. Contreras and M. Martonosi, "Characterizing and improving the performance of Intel Threading Building Blocks," in *2008 IEEE International Symposium on Workload Characterization, IISWC'08*, 2008, pp. 57–66.

[20] J. D. Valois, "Implementing Lock-Free Queues," Tech. Rep., 1994.

[21] R. Thakur and W. D. Gropp, "Test Suite for Evaluating Performance of MPI Implementations that Support MPI THREAD MULTIPLE," *EuroPVM/MPI 2007*, vol. 35, pp. 608–617, 2007.

[22] P. S. Crozier, H. K. Thornquist, R. W. Numrich, A. B. Williams, H. C. Edwards, E. R. Keiter, M. Rajan, J. M. Willenbring, D. W. Doerfler, and M. A. Heroux, "Improving performance via mini-applications." Sandia National Laboratories (SNL), Albuquerque, NM, and Livermore, CA (United States), Tech. Rep., sep 2009.

[23] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, and R. Gropp, "Fine-grained multithreading support for hybrid threaded MPI programming," Tech. Rep. 1, 2010.

[24] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, and R. Thakur, "Toward efficient support for multithreaded MPI communication," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 5205 LNCS. Springer, Berlin, Heidelberg, 2008, pp. 120–129.

[25] M. Chabbi and J. Mellor-Crummey, "Contention-conscious, locality-preserving locks," in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming - PPoPP '16*, vol. 51, no. 8. New York, New York, USA: ACM Press, 2016, pp. 1–14.

[26] A. Amer, H. Lu, Y. Wei, P. Balaji, and S. Matsuoka, "MPI+Threads: runtime contention and remedies," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming - PPoPP 2015*. New York, New York, USA: ACM Press, 2015, pp. 239–248.

[27] H.-V. Dang, S. Seo, A. Amer, and P. Balaji, "Advanced Thread Synchronization for Multithreaded MPI Implementations," in *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2017, pp. 314–324.

[28] W. Endo and K. Taura, "Parallelized Software Offloading of Low-Level Communication with User-Level Threads," in *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region - HPC Asia 2018*, 2018, pp. 289–298.

[29] J. Dinan, P. Balaji, D. Goodell, D. Miller, M. Snir, and R. Thakur, "Enabling MPI interoperability through flexible communication endpoints," in *Proceedings of the 20th European MPI Users' Group Meeting on - EuroMPI '13*. New York, New York, USA: ACM Press, 2013, p. 13.

[30] S. Sridharan, J. Dinan, and D. D. Kalamkar, "Enabling Efficient Multithreaded MPI Communication through a Library-Based Implementation of MPI Endpoints," in *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, vol. 2015-Janua, no. January. IEEE, nov 2014, pp. 487–498.

[31] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10," in *Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications - OOPSLA '05*, vol. 40, no. 10. New York, New York, USA: ACM Press, 2005, p. 519.