Design and Implementation of Portable and Efficient Non-blocking Collective Communication

Akihiro Nomura*†, Yutaka Ishikawa‡, Naoya Maruyama*†, and Satoshi Matsuoka*§
*Global Schientific Information and Computing Center
Tokyo Institute of Technology
Email: nomura.a.ac@m.titech.ac.jp, naoya@matsulab.is.titech.ac.jp, matsu@is.titech.ac.jp
†JST, CREST
‡Depatment of Computer Science
Graduate School of Information Science and Technology
The University of Tokyo
Email: ishikawa@is.s.u-tokyo.ac.jp
§National Institute of Informatics

***Abstract*—Non-blocking communications are widely used in parallel applications for hiding communication overheads through overlapped computation and communication. While most of the existing implementations provide a non-blocking version of point-to-point communications, there is no portable and efficient implementation of non-blocking collectives, partly because application execution contexts need to be interrupted by dependent communications. This paper presents a portable and efficient user-level implementation technique of non-blocking communications. It allows users to design non-blocking collectives by declaring their operations and dependencies using provided APIs without being concerned with complicated management of their progression. While user-level implementations can be less efficient than kernel-level ones due to the cost of OS context switches, we solve this problem by employing the Marcel user-level light-weight thread library when invoking communication operations. More specifically, each communication operation is mapped to one Marcel thread and scheduled to be executed when each operation's dependencies are satisfied by certain events. All executable operations and main user thread are executed simultaneously without any explicit invocations. Performance evaluations with microbenchmarks demonstrate the effectiveness of our proposed technique. Compared to existing OS-thread based method, it reduces CPU load to less than 10% while achieving similar level of communication latencies. We also discuss and compare the descriptive power of internal expressions for non-blocking communications.**

I. Introduction

In parallel applications, there is a growing need for non-blocking collective communication. While the growing scale of high performance computing systems allows for higher compute performance, it is becoming more challenging to efficiently perform collective communication in such large-scale environments. This problem is further amplified by the growing performance gap of computation and intra- and inter-node data movement. Non-blocking collective communication can alleviate the problem by allowing the programmer to overlap communication with computation so as to effectively hide the communication time.

In the next version of the MPI standard, MPI 3.0, non-blocking collective communication APIs is being planned as a part of the standard [1]. However, those APIs cannot be implemented in a straightforward way as was the case for non-blocking point-to-point communication, due to the problem of *progression* [2], [3]. Progression of a collective communication is the process of performing point-to-point communications that are part of the collective communication. Realizing progression with minimum CPU overheads while achieving optimal data transfer performance is the most important challenge in implementing non-blocking collective communication. More specifically, progression can be a source of significant CPU overheads because of the complex interactions of communication event handling and OS scheduling involved in processing of the data dependencies among point-to-point communications.

Past attempts in implementing non-blocking collective communication had performance problems because of the progression. LibNBC [4] is a reference implementation of non-blocking collective communication in the upcoming MPI 3.0 specification. It however consumes significant CPU resources because of its straightforward thread-based approach, and furthermore its communication performance is much slower than existing collective blocking communication implementations. Our past work, KACC, is a more efficient implementation in that it offloads the progression to the OS kernel in order to reduce communication latency and CPU consumption [3]. However, since the original KACC uses Linux kernel modules, which require system administrator's privileges, it is significantly limited in terms of applicability to a wide variety of computing environments, and in fact would be practically impossible to introduce such a module into large-scale production supercomputers. OS kernel-level implementations are also limited in that there is no appropriate and secure way to implement user-defined reduction operations into the kernel-mode context. For the same reason, it is hard to implement collective communication routines for user-defined types.

In this paper, we extend our previous work and propose a new design and implementation of non-blocking collective communication library, uKACC, that is entirely implemented

as a user-level program. The uKACC has the same APIs as the original KACC so that new collective communication algorithms can be introduced without modifying KACC itself. In order to process the progression efficiently, uKACC employs the Marcel user-level thread library [5], [6] and the PIOMan [7] communication scheduler, which are part of the NewMadeleine communication library [8]. Compared to LibNPBC, the light-weight thread scheduling of Marcel allows for significant reduction of the overhead caused by the scheduling of progression threads and main computation threads. Its fine-grained thread priority control reduces the delay in scheduling communication threads so that the overall application performance can be improved. Furthermore, uKACC provides a more powerful mechanism to express complex collective communication algorithms than the LibNBC's API. For example, unlike LibNBC, the start timing of each communication primitive is not bound to a specific timing group. Such a restriction often results in inefficient communication behavior with combination of rendezvous communication.

We compare each method in the performance of non-blocking broadcast. Our uKACC executes non-blocking collective communication almost as fast as the explicit progression method without thread under normalized result. In certain case of TSUBAME2.0, uKACC consumed less than 10% of CPU time while LibNBC consumes more than 70%. We also compare uKACC to original OS kernel-based KACC. UKACC does not outperform original KACC, but marks comparable performance without the restriction of OS kernel implementation.

II. Related work

A. Software based non-blocking collective communication

LibNBC [4], [9] is a reference implementation of non-blocking collective communication routines in the upcoming MPI 3.0 specification. The earlier version of LibNBC required users to call the library periodically, but this strategy was abandoned later because it cannot ensure that the progression is executed in a sufficient frequency [2]. The current version of LibNBC employs Pthreads to progress their collective communications written in MPI, but does not ensure that communication threads are always prioritized over other application threads because of the limited priority control in Pthreads. More specifically, while Pthreads allows for specifying priority of individual threads, but it is not possible to make arbitrary threads to have higher priority than the prefixed default. The Pthreads-based implementation works well in some cases, because blocking in MPI calls are considered less CPU consuming compared to user calculations and thus OS scheduler prioritizes communication threads. However, in other cases, MPI library employs polling to wait for communication completion; in this case, both the communication thread and user computation thread exhibits high CPU load and OS scheduler cannot prioritize the threads appropriately. This leads to inefficient CPU usage and inappropriate context switch timing, resulting in high latencies in message transmissions and overall performance loss.

A new non-blocking collective I/O interface is proposed by Venkatesan et al. [10] They extend the non-blocking collective communication in MPI 3.0 by relaxing constraints that the amount of data transfer is known a-priori by all MPI processes. In blocking collective communication, *allgather* is often used to exchange the amount of data before calling collective communication. In non-blocking collective communication, however, this might become a bottleneck.

We have proposed KACC [3] that is based on a kernel-level implementation of progression routines. In our previous implementation, we eliminated communication threads by moving its functionality into OS kernel's interrupt handlers and tasklets invoked by the handlers, and demonstrated its effectiveness in improving efficiency. However, kernel-level implementation brings some limitation which will be discussed in the next section. Schneider also proposed a kernel-level implementation of non-blocking collective communication, named GOAL [11], which is similar to our kernel-level implementation with the same limitations.

B. Hardware assisted non-blocking collective communication

Some network interconnects allow for offloading of collective communication operations such as barrier [12]. Such hardware-aided solutions, while limited to a small subset of MPI collective routines, are likely to give better performance than pure software-based implementations. In InfiniBand network, more complicated non-blocking collective communications, such as all-to-all [13] and allreduce [14], are implemented using InfiniBand's management queue [15]. Our uKACC currently does not use such hardware offloading engines, but can be extended for potentially better performance such that it transparently employs hardware mechanisms when available while keeping the same user API for better portability.

III. Issues

A. Inefficiencies in Thread-based Progression Implementations

There are two known approaches in implementing progression in non-blocking collective communication: having the user call progression routine explicitly, or creating progression threads. The former approach requires the user to call routines such as `MPI_Test` periodically after issuing collective communication in order to issue dependent point-to-point communication operations. However, it is impractical to assume that this requirement is always satisfied, and when there are no API calls made during computation, non-blocking communication cannot progress during computation without any overlapping. This in effect almost nullifies the original intent of non-blocking collective communication.

The latter approach implements progression in different execution contexts than main computation threads so that it does not rely on the explicit progression by the programmer. However, this approach causes another problem where the extra threads can conflict with the main application threads for CPU usage when MPI applications use the same number of

threads or processes as the number of CPU cores, resulting in frequent context switches between the communication threads and user threads. This scheduling problem can be mitigated by setting their priority; however, the Pthreads library does not support fine-grained scheduling policies for progression, and the priority APIs in operating systems usually do not have features to give higher priority to specific threads compared to the main thread. Therefore, we cannot guarantee that the communication thread is always called at appropriate timings, which results in the slowdown of communication and inefficient CPU time consumption as we will demonstrate.

B. Inefficiencies in LibNBC's algorithm description

In the LibNBC library [9], collective communication algorithms are described in the *Collective Schedule* structure. The schedule consists of multiple *rounds* to express dependencies among the point-to-point communications. The round is a set of point-to-point communications and reduction operations which can be executed simultaneously. The operations in different rounds may not be executed simultaneously, and thus may have data dependencies. Operations in each round are scheduled to be executed after the completion of all the operations in the previous round.

This round-based method may degrade performance of pipeline-style implementations of collective communications as follows. It is possible to improve performance when transferring large messages by splitting them into small pieces to overlap the receive and transfer, and in fact, the broadcast algorithm in LibNBC for large messages employs this pipeline strategy. In this algorithm, the messages are divided into n small pieces and each piece is transmitted sequentially in a chain topology. On each intermediate node, transfer of kth piece must be executed after the reception of kth piece due to data dependency, while kth send and $(k+1)$th receive can be done simultaneously and therefore can be packed into a single round. In LibNBC's collective schedule structure, these dependencies are expressed as putting each operation into different rounds.

The performance of this round-based, pipeline-style broadcast suffers from the fact that sending messages before calling corresponding receive APIs is often inefficient. This is the case both for two well-known sending strategies, eager communication and rendezvous communication [16]. Generally, when a message arrives before the issue of `MPI_Recv` or its siblings, MPI library cannot determine which memory part to store the incoming message. In eager communication strategy, the MPI library will allocate memory for such unexpected messages and store the content into temporary buffers, and the content is copied after calling receive APIs. In rendezvous communication strategy, often employed for larger messages, the sender asks the receiver whether the corresponding receive API has been issued and postpones the transfer if it is not ready. Thus, sending messages before calling corresponding receive API results in extra cost in both cases.

In the pipeline-style algorithm, this send-before-receive problem tends to occur for the following reason. In this algorithm, a round that contains kth receive begins almost the same time as the round contains kth send in the previous node, because both of them begin just after finishing the $k-1$th communication. In this case, the send-before-receive problem can occur in some probability because of jitters in message handling. However, this restriction is unessential in description of algorithm, because the receiver's buffer is ready before finishing the previous round of communication and the memory address of receive buffer can be determined. The cause of the problem is that LibNBC's collective schedule structure does not permit to issue receive operations beforehand because they cannot distinguish whether each receive operation has an actual dependency or fake dependency. Thus the round-based pipeline broadcast tends to suffer from slowdown. We eliminate this problem by using *Algorithm Design Graph* as described later.

C. Unfavorable use of OS Kernel modules

The problem in Pthreads-based approaches can be solved by employing the operating system communication and scheduling components for managing progression. For example, the KACC library offloads progression to OS kernel's interrupt handler and OS kernel threads [3], and is demonstrated to ensure optimal scheduling among main computation and communication threads. However, requiring a custom kernel extension prohibits applicability of such an approach in many of production computing systems, including large-scale shared supercomputers, where the advantage of non-blocking communication is more highlighted. In addition, handling all operations in the OS kernel context brings several difficulties. For example, it is difficult to allow user-defined reduction operations and types since OS kernel module code cannot be changed in each execution of user programs. This can be realized by allowing the user-defined code to be called back from OS kernel code. It would however nullify all the performance gains in kernel-level implementations. Furthermore, floating point arithmetic operations in reductions can also be a problem since kernel mode contexts in Linux are not expected to use floating point registers.

IV. Design and Implementation

The user-level KACC, uKACC, is based on the kernel-level KACC, but does not rely on kernel modules. KACC consists of three parts described in Figure 1: the API Layer, the Progress Engine, and the P2P Layer. The latter two layers are originally implemented as an OS kernel module using *tasklets*, which are kernel-level threads in Linux. In uKACC, we implement the same functionality on top of the Marcel light-weight user-level thread library [5], [6] and the PIOMan communication scheduler [7], both of which are being developed at INRIA Bordeaux. We design uKACC within the MadMPI library, which is an MPI implementation provided in the NewMadeleine communication library [8], [17]. In MadMPI, all communications are managed by PIOMan and all threads are managed by Marcel.

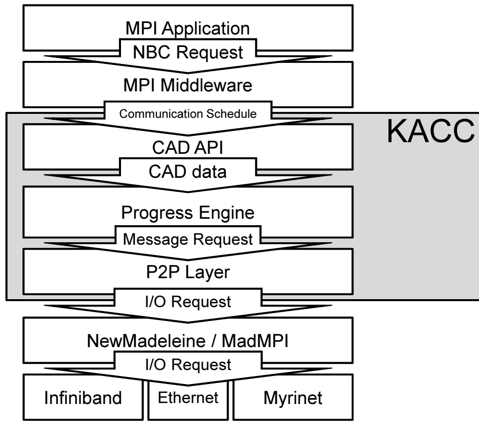

Fig. 1. KACC Facility

A. Algorithm Design Graph

In order to express collective communication algorithms, the KACC has *Algorithm Design Graph (ADG)*, which allows for expressing dependency trees among point-to-point communications and calculations in collective communications. With ADG, collective communication algorithms can be expressed as directed acyclic graphs (DAGs). Each DAG node represents either a point-to-point communication (`SEND` or `RECV`) or a reduction operation (`CALC`), and is connected by edges that represent dependencies among nodes. There are two special nodes, `START` and `END`, which are the common source and sink, respectively.

ADG allows for expressing arbitrary dependencies among point-to-point communications in collective communications. As receive operations can be issued independently from send operation's dependencies, we can avoid the timing problems described in Section III-B. The original KACC employed ADG to pass the algorithm design from user processes to the KACC OS kernel module, and the same facility is also used in uKACC to manage collective communication algorithm and its progress.

B. KACC API to Manipulate ADG

KACC provides API to express collective communications. Users can manipulate ADG structure and tell KACC to execute and query for its completion using the following API:

`Init()`
: Creates a new ADG for collective communication.

`Make{Send|Recv|Calc}Node()`
: Creates ADG nodes for `SEND`, `RECV` and `CALC`, respectively.

`ConnectNode(A, B)`
: Adds the dependency edge from node `A` to node `B`. `START` and `END` nodes are pre-defined.

`Issue()`
: Tells KACC to start communication.

`Query()`
: Queries KACC whether the operation has been completed.

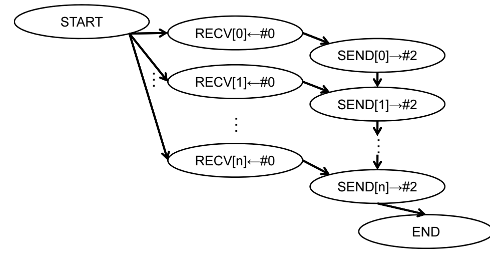

Fig. 2. ADG tree of Pipeline-Style Broadcast Algorithm

```
 1 /* Initializing ADG Structure */
 2 nbc = Init();
 3 sprev = START;
 4 for (i = 0; i < n; i++) {
 5   /* Making Recv and Send Node */
 6   rn = MakeRecvNode(nbc, addr[i], fragsize, prevrank);
 7   ConnectNode(nbc, START, rn);
 8   sn = MakeSendNode(nbc, addr[i], fragsize, nextrank);
 9   ConnectNode(nbc, rn, sn);
10   ConnectNode(nbc, sprev, sn);
11   ConnectNode(nbc, sn, END);
12 }
13 ConnectNode(nbc, sprev, END);
14 /* Issuing NB Coll */
15 req = Issue(nbc);
```

Fig. 3. ADG API Example

For example, an ADG tree corresponding to a pipeline-style broadcast algorithm in rank #1 node is shown in Figure 2. In ADG API, each MPI process generates its own AGD tree independently, and all communication primitives and their dependencies related to the process are placed in the tree. This tree is generated by the code shown in Figure 3.

C. Progression Using Marcel Thread Library

In order to process non-blocking collective communication in the background of main application threads, we employ light-weight user-level threads provided by Marcel. It has similar APIs to Pthreads, but implements multithreading without OS context switches at the user level. Each Marcel thread does not have its own execution context in the underlying operating system, but is scheduled on Virtual Processors(VPs), which are Marcel's abstractions of execution contexts. Since VPs are implemented as user-level components, the cost of managing Marcel threads such as creation and destruction is relatively smaller than that of Pthreads.

In uKACC, each ADG node is mapped into a Marcel thread. Each Marcel thread is started when the corresponding ADG node becomes ready. The threads for the first nodes in ADG are scheduled to run immediately when the collective communication is issued. Each ADG node has reference count of preceding dependencies and it is decremented by the progress engine at the end of each predecessor thread's operation. When the reference count of dependent nodes reaches zero, which means that the corresponding ADG node is ready to run, the Marcel thread is scheduled to run. The completion of the overall collective operation can be checked by using the reference count of `END` node.

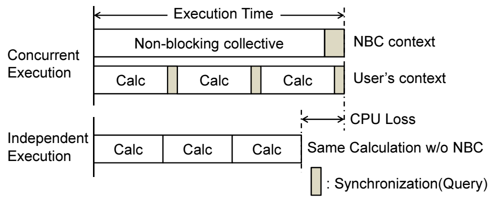

Fig. 4. Relationship between measured times

TABLE I
SPECIFICATION OF EVALUATION CLUSTERS

	TSUBAME2.0	Tateyama
CPU	Xeon X5670	Opteron 2212HE
CPU Frequency	2.93 GHz	2.0 GHz
CPU Cores	2 socket x 6 core x 2 HT	2 socket x 2 core
Hyper threading	Enabled	N/A
Memory	54GB	4GB
OS	Linux 2.6.32(SuSE EL)	Linux 2.6.32(RHEL 6)
Network	2 x Infiniband QDR	Gigabit Ethernet

D. P2P Layer and PIOMan Communication Scheduler

We use the NewMadeleine communication library and the PIOMan communication scheduler to execute each point-to-point communication. The aim of the P2P layer is to execute point-to-point communication as fast as possible and to wake up Progress Engine at appropriate timings. In uKACC, we use MadMPI's `MPI_Send` and `MPI_Recv` functions in the Progress Engine's thread. Marcel-enabled PIOMan creates a waiting queue of Marcel threads, and activates a thread when its dependent message arrives. This feature enables optimal scheduling between communication threads and user computations.

V. EVALUATION

A. Evaluation Scenario

We measure the effectiveness of our user-level implementation of non-blocking collective communication in uKACC. In our benchmark, one non-blocking broadcast involving all MPI processes is initially issued, and then a fixed size computation is executed on each node in a repeated fashion. At the end of each computation, a process queries whether the collective communication is finished, and if not, repeats the computation until so. We measure the length of time taken for the non-blocking collective communication as well as the time for the entire computation during the overlapped computation-communication phase (Figure 4). We employ BLAS `dgemm()` routine for fixed-size calculation, and we also measure the time required for calculation without background collective communication, repeating the computation the same number of times as that was for the collective communication, and use that as a baseline on how much CPU resource is being used purely for calculation, and how much is spent for the other parts, namely communication and thread switching.

B. Evaluation Environments and Target Algorithm

We evaluated our systems on two machines, the TSUBAME2.0 supercomputer cluster and the Tateyama cluster. TSUBAME2.0 is a large-scale production supercomputer at Tokyo Institute of Technology, being fifth fastest on the Top500 as of November, 2011. TSUBAME2.0 embodies dual-rail QDR Infiniband, and as such has tremendous network injection bandwidth, supported by very fast CPUs supporting up to 24 Intel Hyper-Threads per node. However, since TSUBAME2.0 is a production machine, the users are never given root privileges, and moreover, installation of arbitrary OS kernel module to the nodes is very difficult unless it is shown to be fully tested and debugged, and with widespread benefit to the 2000 or so user base. On the other hand Tateyama is an older laboratory-owned private cluster at the University of Tokyo, and there we have administrator privileges to install arbitrary OS kernel modules. We evaluated the user-level approach, namely uKACC, LibNBC and manual progression, on both TSUBAME and Tateyama, while the kernel-level approach was executed on Tateyama for comparisons with our original kernel-level implementation KACC. The specifications of the machines are described in Table I.

For uKACC we use the MadMPI MPI library, which is a part of NewMadeleine and also upon which uKACC is dependent. Unfortunately, we are forced to use OpenMPI for LibNBC, because we were unable to make LibNBC work on MadMPI. Because different MPI implementations would exhibit non-uniform baseline performances, in order to conduct a fair assessment on how much overhead is actually incurred by the collectives, we pre-measure each MPI implementation for its non-blocking point-to-point communication performance, and use that as a baseline to normalize the result,

In MadMPI, we use the following compile options (excerpt):

```
pkg marcel --enable-keys
  --enable-maintainer_mode
  --disable-static --with-topo=mono
pkg pioman --with-marcel
pkg nmad --with-pioman --enable-mpi
```

In this setting, Marcel does not use Pthread to execute their threads in parallel, but executes them with its own time-sharing scheduling, with blocking communication information being derived from PIOMan. We explicitly designate a CPU core for each MPI process using MadMPI's launcher's option or `numactl` command to avoid contention between processes on the same node. To be more specific, we assigned one physical CPU core to each MPI process, that is 12 processes per node in TSUBAME2.0 and 4 processes per node in Tateyama.

C. Evaluation Results

We measured the execution time and CPU consumption for non-blocking broadcast of 1.25MB data for the four systems being compared: *uKACC* is our implementation running in MadMPI, *LibNBC* is the Pthread-based implementation running on OpenMPI, *MadMPI* and *OpenMPI* are manual MPI point-to-point based implementations for the corresponding MPIs, requiring explicit and periodic calls of progression routines. The algorithms of collective communication are

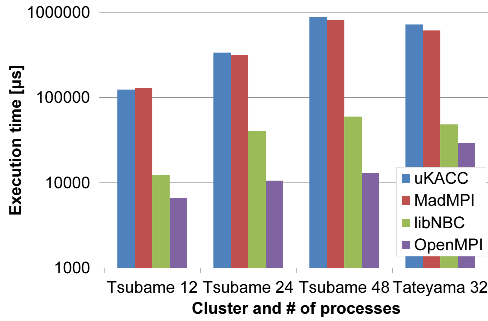

Fig. 5. Execution time of non-blocking broadcast (Message size: 1.25MB)

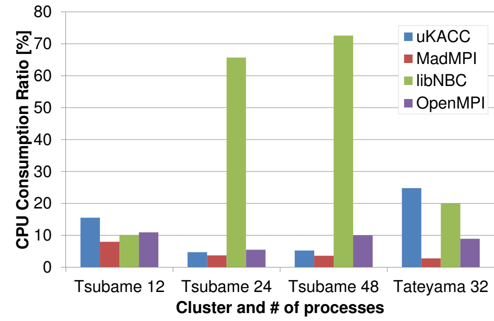

Fig. 7. CPU usage ratio in non-blocking broadcast (Message size: 1.25MB)

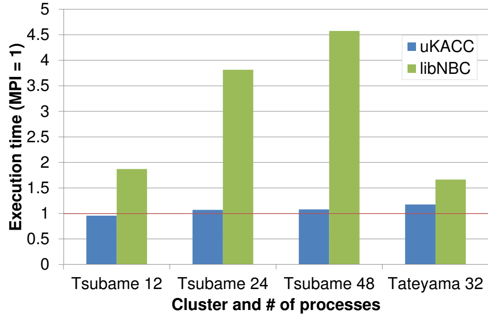

Fig. 6. Relative execution time of non-blocking broadcast (Message size: 1.25MB)

identical over all the systems: 1D chain pipeline broadcasting. Here, the root node divides the messages into small pieces and sends each piece to the next node sequentially, while an intermediate node relays a received piece immediately after reception. In uKACC, the ADG tree of this algorithm on the intermediate node #1 is shown in Figure 2. In LibNBC, each kth send and the successive $k+1$th receive on the node are placed into the same round as described in section III-B where we must have the receive be issued before the corresponding send. In the manual MPI implementations, all `MPI_Irecv` receive requests are issued at the beginning of a non-blocking broadcast, and the progression routine is called periodically. In the progression routine, completion of the uncompleted receive with the smallest index is queried for, and if completion is detected, then the corresponding `MPI_Isend` is issued.

The execution time for non-blocking broadcast is shown in figure 5. As described above, the relative difference in the MPI implementation performance is apparent, where OpenMPI is generally an order of magnitude faster than MadMPI, requiring normalization for fair comparison. When we normalize uKACC and libNBC with respect to the manual MPI non-blocking point-to-point API implementation as is shown in Figure 6, we see that uKACC is far superior to LibNBC, almost matching that of the manual MPI implementations.

CPU usage ration is also compared in figure 7. This shows that uKACC and manual MPI consume much less CPU than LibNBC. We attribute the difference in how the asynchronous messages are treated, either by the kernel or using threads with differing scheduling strategies. For manual MPI, non-blocking communication is issued in the computation thread and is executed in the background, while no threads dedicated for communication is created. For LibNBC, a communication thread is created at the beginning of collective communication for each rank, and all communications are issued as non-blocking in the respective thread. Each communication thread in turn waits for completion of the communication immediately after it is issued, effectively blocking with `MPI_Waitany`. However, many MPI libraries including OpenMPI wait for the communication completion by polling instead of the blocking wait in functions like `MPI_Waitany` for better performance, because the CPU core is considered to be available to use. In this case, this polling wait consumes the CPU core which should be used by overlapping computation and it leads to performance degradation. On the other hand, for uKACC each communication is mapped into a Marcel thread and each thread issues the communication as blocking wait. Internally within NewMadeleine, the blocking communication is implemented as non-blocking communication immediately followed by waiting for the request. Although this would seem similar to LibNBC case, the MPI library does not misunderstand availability of CPU core, because for PIOMan the communication library and scheduler mutually know the communication state of each thread, and acts appropriately. More specifically in NewMadeleine, the PIOMan communication scheduler knows the relationship between threads and messages, and can schedule execution of a thread just after the arrival of the message it is waiting for. On the other hand, in Pthread, no such relationship exists, resulting in the anomalous behavior described above.

The same experiment conducted with varying message sizes on TSUBAME2.0 using 48 processes are shown in Figures 8, 9 and 10. The uKACC exhibited almost the same execution time compared to manual MPI implementations in the worst case, and if fact 24.2% faster for small message size of 12.5KB. Contrastingly, LibNBC shows substantial overhead, even for small message sizes. We believe the reasons for this could be twofold: one could be that the thread creation and context switching cost is far larger for LibNBC, incurring constant overhead. The other reason could be that the difference in performance of MadMPI and OpenMPI might be deflating the relative overhead incurred for uKACC. However, the latter

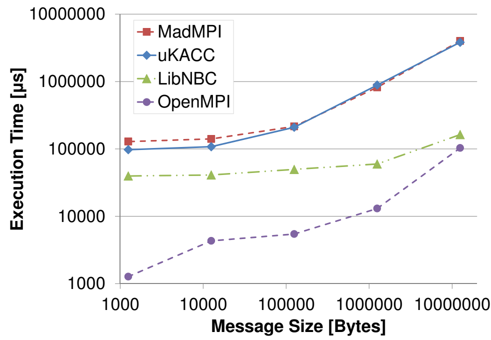

Fig. 8. Execution time of non-blocking broadcast in Tsubame 48 processes

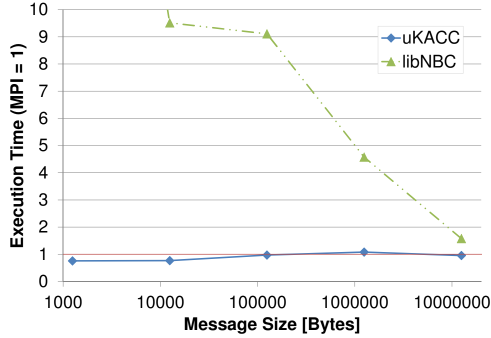

Fig. 9. Relative execution time of non-blocking broadcast in Tsubame 48 processes

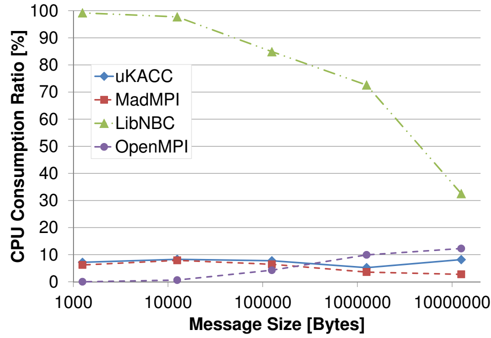

Fig. 10. CPU usage ratio in non-blocking broadcast in Tsubame 48 processes

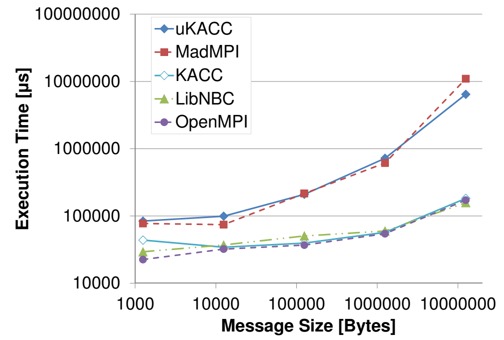

Fig. 11. Execution time of non-blocking broadcast in Tsubame 48 processes

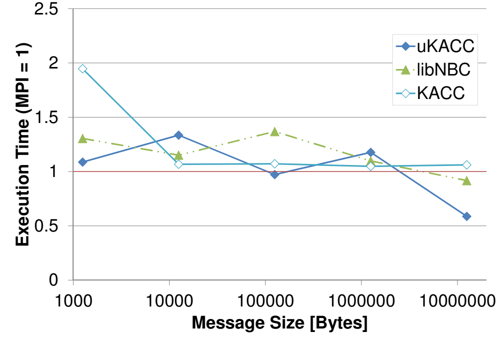

Fig. 12. Relative execution time of non-blocking broadcast in Tsubame 48 processes

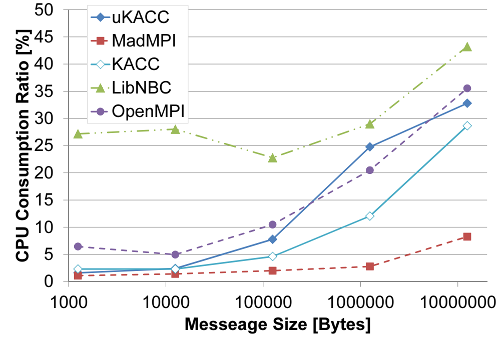

Fig. 13. CPU usage ratio in non-blocking broadcast in Tsubame 48 processes

might not hold, as the increase in execution time for uKACC is clearly visible for small message sizes whereas for LibNBC the increase is rather minimal. In fact the manual OpenMPI performance is rising sharply; this would indicate that LibNBC would seem less responsive to the performance variance of the underlying MPI.

D. Comparison to Kernel-mode Implementation

We compared the performance between uKACC and the original KACC on the Tateyama cluster using 32 processes (Figures 11, 12 and 13). Comparing execution time is somewhat difficult due to the performance instability introduced by thread scheduling. As kernel-mode KACC employs Linux tasklet, its performance is very stable, whereas uKACC and LibNBC exhibits larger variance. Overall, however, the relative execution time is in the same ballpark. Another indicator is the CPU consumption ratio, where the consumption by uKACC is much smaller than that of LibNBC, and is being slightly higher than KACC. The results show that the OS kernel based implementation by all means is the best choice in terms of performance, but our uKACC implementation in user space can be competitive.

VI. Conclusion

We showed inefficiencies in the Pthread-base implementation of non-blocking collective communications. OS thread scheduler which does not have a knowledge of waiting message of blocking threads produces large CPU overhead and communication delay. Inefficient message handling caused by the design of collective communication algorithm description

also makes the communication slower. In order to show the potential of faster and light-weight implementations of non-blocking collective communication, uKACC is proposed. In uKACC, communication algorithm is expressed in dependency graph to provide maximum level of expressiveness. Non-blocking collective communications are handled in communication-aware thread scheduler and are scheduled with appropriate timings.

We evaluated uKACC by comparing communication time and CPU consumption during non-blocking collective communication. Our uKACC performed better compared to LibNBC in large-scale production supercomputer, TSUBAME2.0, and comparable to kernel-based implementation of our predecessor KACC.

Current implementation depends on a specific MPI and communication library, MadMPI in NewMadeleine. Therefore, it cannot be used with arbitrary MPI implementations such as OpenMPI and MPICH2. It is an open problem whether provides portable implementation of KACC, independent of both OS kernel and base MPI library.

The other open problem is performance. Currently, the communication layer in uKACC is based upon MadMPI, but NewMadeleine has internal interfaces that can be used to implement the non-blocking collective communication, and how much we could exploit them to eliminate the potential overhead we are incurring for the MadMPI layer is not clear. Eventually, we should establish the minimal requirement of communication thread and its scheduler and implement them in a portable fashion so that we can adapt to any MPI implementations without potential overhead. In addition to that, our P2P layer should be extended to employ interconnect-specific communication offloading mechanisms, such as InfiniBand's management queue, to treat communication more efficiently.

ACKNOWLEDGMENT

The authors would like to thank Dr. Alexandre Denis for his grateful help to making NewMadeleine work on TSUBAME2.0 and for lots of his useful suggestions in implementation of uKACC.

This work was partially supported by JST, CREST through its research program: "Highly Productive, High Performance Application Frameworks for Post Petascale Computing."

This work was partially supported by the ANR-JST project, FP3C: "Framework and Programmng for Post Petascale Computing."

REFERENCES

[1] MPI Forum, "Mpi 3.0 standardization effort," http://meetings.mpi-forum.org/MPI_3.0_main_page.php.

[2] T. Hoefler and A. Lumsdaine, "Message Progression in Parallel Computing - To Thread or not to Thread?" in *Proceedings of the 2008 IEEE International Conference on Cluster Computing*. IEEE Computer Society, Oct. 2008.

[3] A. Nomura and Y. Ishikawa, "Design of kernel-level asynchronous collective communication," in *The 17th European MPI User's Group Meeting (EuroMPI 2010)*, Sep 2010, pp. 92–101.

[4] T. Hoefler and A. Lumsdaine, "Optimizing non-blocking Collective Operations for InfiniBand," in *Proceedings of the 22nd IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, Apr. 2008.

[5] V. Danjean and R. Namyst, "Controling Kernel Scheduling from User Space: an Approach to Enhancing Applications' Reactivity to I/O Events," in *Proceedings of the 2003 International Conference on High Performance Computing (HiPC '03)*, Hyderabad, India, Dec. 2003. [Online]. Available: http://www.springerlink.com/link.asp?id=r99f1x65v4gw07pp

[6] S. Thibault, R. Namyst, and P.-A. Wacrenier, "Building Portable Thread Schedulers for Hierarchical Multiprocessors: the BubbleSched Framework," in *Proceedings of the 13th International Euro-par Conference*, ser. Lecture Notes in Computer Science, vol. 4641, ACM. Rennes,France: Springer, 8 2007. [Online]. Available: http://hal.inria.fr/inria-00154506

[7] F. Trahay and A. Denis, "A scalable and generic task scheduling system for communication libraries," in *Proceedings of the IEEE International Conference on Cluster Computing*. New Orleans, LA: IEEE Computer Society Press, Sep. 2009. [Online]. Available: http://hal.inria.fr/inria-00408521

[8] F. Trahay, É. Brunet, and A. Denis, "An analysis of the impact of multi-threading on communication performance," in *CAC 2009: The 9th Workshop on Communication Architecture for Clusters, held in conjunction with IPDPS 2009*. Rome, Italy: IEEE Computer Society Press, May 2009. [Online]. Available: http://hal.inria.fr/inria-00381670

[9] T. Hoefler and A. Lumsdaine, "Design, Implementation, and Usage of LibNBC," Open Systems Lab, Indiana University, Tech. Rep., Aug. 2006.

[10] V. Venkatesan, M. Chaarawi, E. Gabriel, and T. Hoefler, "Design and Evaluation of Nonblocking Collective I/O Operations," in *Recent Advances in the Message Passing Interface (EuroMPI'10)*, vol. 6960. Springer, Sep. 2011, pp. 90–98.

[11] T. Schneider, S. Eckelmann, T. Hoefler, , and W. Rehm, "Kernel-Based Offload of Collective Operations - Implementation, Evaluation and Lessons Learned," in *Proceedings of the 17th international conference on Parallel processing - Volume Part II*, ser. Euro-Par'11. Springer-Verlag, 2011, pp. 264–275.

[12] G. Almási, P. Heidelberger, C. J. Archer, X. Martorell, C. C. Erway, J. E. Moreira, B. Steinmacher-Burow, and Y. Zheng, "Optimization of mpi collective communication on bluegene/l systems," in *Proceedings of the 19th annual international conference on Supercomputing*, ser. ICS '05. New York, NY, USA: ACM, 2005, pp. 253–262. [Online]. Available: http://doi.acm.org/10.1145/1088149.1088183

[13] K. Kandalla, H. Subramoni, K. Tomko, D. Pekurovsky, S. Sur, and D. K. Panda, "High-performance and scalable non-blocking all-to-all with collective offload on InfiniBand clusters: a study with parallel 3d FFT," in *International Supercomputing Conference (ISC)*, vol. 26, 2011, pp. 237–246.

[14] K. Kandalla, U. Yang, J. Keasler, T. Kolev, A. Moody, H. Subramoni, K. Tomko, J. Vienne, and D. K. Panda, "Designing non-blocking allreduce with collective offload on InfiniBand clusters: A case study with conjugate gradient solvers," in *International Parallel and Distributed Processing Symposium (IPDPS '12)*, 2012.

[15] R. L. Graham, S. Poole, P. Shamis, G. Bloch, N. Bloch, H. Chapman, M. Kagan, A. Shahar, I. Rabinovitz, and G. Shainer, "Connectx-2 infiniband management queues: First investigation of the new support for network offloaded collective operations," in *Cluster Computing and the Grid*, 2010, pp. 53–62.

[16] R. Milner, *A Calculus of Communicating Systems*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1982.

[17] O. Aumage, É. Brunet, N. Furmento, and R. Namyst, "Newmadeleine: a fast communication scheduling engine for high performance networks," in *CAC 2007: Workshop on Communication Architecture for Clusters, held in conjunction with IPDPS 2007*. Long Beach, California, USA: IEEE Computer Society Press, March 2007, also available as LaBRI Report 1421-07 and INRIA RR-6085. [Online]. Available: http://hal.inria.fr/inria-00127356