# Design of Kernel-level Asynchronous Collective Communication

Akihiro Nomura[1] and Yutaka Ishikawa[1]

Dept. of Computer Science, Graduate School of Information Science and Technology,
The University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo, JAPAN
nomura@il.is.s.u-tokyo.ac.jp
ishikawa@is.s.u-tokyo.ac.jp

**Abstract.** Overlapping computation and communication, not only point-to-point but also collective communications, is an important technique to improve the performance of parallel programs. Since the current non-blocking collective communications have been mostly implemented using an extra thread to progress communication, they have extra overhead due to thread scheduling and context switching. In this paper, a new non-blocking communication facility, called KACC is proposed to provide fast asynchronous collective communications. KACC is implemented in the OS kernel interrupt context to perform non-blocking asynchronous collective operations without an extra thread. The experimental results show that the CPU time cost of this method is sufficiently small.
keywords: Non-blocking collective communication, Linux kernel

## 1 Introduction

In parallel applications, the performance and efficiency of communications often dominate the performance of the whole calculation. In addition to blocking point-to-point communication APIs in the MPI (Message Passing Interface) [5], some APIs for non-blocking communication, such as `MPI_Isend` and `MPI_Irecv`, are defined. Non-blocking communication allows calculations to continue during communication. This enables the MPI processes to overlap between calculation and communication.

MPI also defines collective communication APIs, such as `MPI_Reduce` and `MPI_Bcast`, to perform the conventional sets of communications easily and efficiently. The users of the MPI library do not need to know what is going on during the collective communication. The MPI library offers the most efficient algorithms for the requested collective communication with regard to the communication size, topology, and other information. Both APIs are used to efficiently perform the communication.

In the current version of MPI, due to the lack of non-blocking collective communication APIs, users must implement non-blocking communications in order to perform the collective communications asynchronously. For example, in the HPL [8] implementation, a non-blocking version of `MPI_Bcast` was implemented,

but it is hard to maintain the code due to the complexity of the collective algorithms and a mixture of communication and computation routines. Furthermore, the code might be inefficient in some topologies because the broadcast algorithm is based on some assumed network topology. Thus, the introduction of non-blocking collective communication APIs to the MPI standard has been discussed.

In the next version of the MPI standard, MPI 3.0, non-blocking collective communication APIs are to be introduced. There is a reference implementation of those APIs, LibNBC [2, 3]. In the implementation, non-blocking communication operations are implemented using threads for communication progress. The thread implementation has two limitations. Firstly, if an extra thread that performs communications is introduced, it consumes CPU resources due to the overhead of both task scheduling and context switching. For example, if an MPI application runs on an eight-core cluster in which each process runs on each CPU core, sixteen threads are created. Eight threads are for processes, and the other eight threads are for communication progress. This means that the execution of those threads is multiplexed. Secondly, since the timing of communication progress depends on the task scheduling in the operating system, it is not guaranteed that the progress thread runs immediately when the communication processing is ready when a message arrives.

In this paper, a new non-blocking collective communication facility, called KACC, is designed and implemented to overcome the limitations described above. KACC is implemented in the OS kernel interrupt context in order to perform the non-blocking collective operations without an extra thread. Since the communication progress is handled when a message arrives, there is no delay in the progress, and no extra context switching overhead is introduced. The facility has been implemented as a kernel module with a user-level library in the Linux kernel. KACC is evaluated by a benchmark which uses non-blocking broadcast algorithm. The benchmark reveals how much the non-blocking broadcast operation contributes to overlapping communication and computation. Four implementations of the non-blocking broadcast operation are considered: a tree-based broadcast operation written in MPI, a non-blocking point-to-point operation; a tree-based broadcast operation written in KACC; a pipeline-based broadcast operation written in the threads; and a pipeline-based broadcast operation written in KACC.

The CPU waste time depends how often the application program examines the completion of the non-blocking operations. The results of the benchmarks show that 97 % of CPU time is lost in LibNBC, while only 31.8 % of CPU time is lost in KACC during a high frequency of examinations. The total execution time of non-blocking collective operation is also improved. The result shows that the execution time in KACC is 79 to 101% of that in LibNBC.

## 2 Issues

Implementation of non-blocking collective communications is not trivial due to progressions. Most of the implementations of collective communications consist of the set of point-to-point communications that are connected by data dependencies. Progression is the procedure to connect these point-to-point communications. The MPI library must issue the communication when all of the dependent communications are completed in order to continue collective communication. Two implementations have been introduced so far: thread implementation and explicit progression.

**Thread Implementation** The straightforward solution to this problem is creating a thread for communication and performing progression in this thread. An example of this method is used in the LibNBC implementation [3]. The advantage of this method is that the communication will execute asynchronously to computation that runs on another thread. Theoretically, the communication thread runs independently from the computation thread, and the progression is always executed at the appropriate timing.

However, the real situation is different due to the limitation of the number of CPU cores. The user usually spawns the same number of MPI processes as the number of CPU cores, because the user often assumes that all cores can be used for computation. In this situation, if the MPI library makes the communication thread, the number of active threads exceeds the number of cores. This results in frequent context switches among these threads. If the context switches are performed by the operating system, and the OS does not know about the dependencies among these threads, then the timing of context switches might not be optimal. This will result in waste of CPU time and delay of communications.

**Explicit Continuation** Another way to implement non-blocking communications instead of creating communication threads is to implement the progressions in the MPI library, that is, the progression of collective communications is only done within the MPI functions, such as `MPI_Test` and `MPI_Wait`, when invoked by the application program. This method does not create any threads, and thus the context switching problem does not happen.

On the other hand, the progression is not processed if the application does not call any MPI functions. This results in no overlapping computation and communication, that is, although a non-blocking collective communication has been posted, the communication does not progress during the computation. If the program waits for the completion of the non-blocking collective communication by issuing `MPI_Wait` when computation is completed, the progression starts. If the user calls the progression too frequently to avoid missing progression timing, this results in a loss of CPU time.

This kind of explicit progression method is used in some MPI applications. For example, in the Linpack benchmark program, non-blocking broadcast is implemented using non-blocking send and receive primitives. For example, during

local computation, Linpack polls whether the broadcast has been completed using the `MPI_Test` primitive.

## 3 Design

In order to solve both the frequent context switching and false asynchronization problems at the same time, the KACC facility is designed and implemented in this paper. In KACC, the progression routine is implemented as an OS kernel's soft-interrupt handler. In this method, the number of threads does not increase, because the progression routine does not have a thread context, and it is called at the appropriate time by the kernel interrupt handler instead of the OS scheduler.

### 3.1 Collective algorithm design

Splitting the collective algorithm from the kernel module is important to design new collective algorithms. However, it becomes a security hole if the program binaries described in the user-level program can be passed to kernel space directly. Instead of sending binary, a data structure, called a CAD (Collective Algorithm Design) structure, is introduced to describe the collective algorithms. The MPI library creates the CAD structure and passes the structure to the kernel module.

Collective communication algorithms can be mapped to directed acyclic graphs which shows dependencies among the point-to-point and reductive operations [3,6]. In the CAD structure, collective algorithms are expressed using these graph structures instead of the binary program. There are three types of nodes: `SEND`, `RECV`, and `CALC`. The `SEND` and `RECV` nodes represent communication. These nodes contain information required for communication: address of data, data size, rank of sender or receiver, and tag information to match the messages. The `CALC` node represents calculation in the MPI reduction function, such as `MPI_Sum` and `MPI_Max`. This node contains information about the reductive calculation operator, memory address, size, and data type. The edges between nodes denote dependencies between each operation.

A CAD, describing a collective algorithm, is created and executed using the following API:

- `InitCAD()`: creates new CAD
- `MakeSendNode()`, `MakeRecvNode()`, `MakeCalcNode()`: creates CAD nodes for `SEND`, `RECV` and `CALC`, respectively
- `ConnectNode(A, B)`: marks the dependency edges from node `A` to node `B` `START` and `END` nodes are pre-defined
- `IssueCAD()`: tells the system to start communication
- `QueryCAD()`: queries to system whether the operation has been completed

An example of a CAD structure, representing a non-blocking broadcast algorithm, is shown in Figure 1. This structure is generated at rank 1 by the user-level code shown in Figure 2. Note that in this broadcast implementation, each `SEND`/`RECV` node sends/receives a fragment of the message. The message

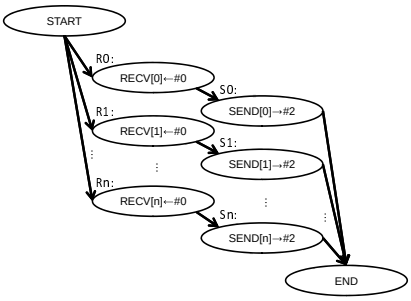



**Fig. 1.** CAD tree example

```
 1 /* Initializing CAD Tree */
 2 cad = InitCAD();
 3 /* Making R0 and S0 Node */
 4 rn = MakeRecvNode(cad, addr[0], fragsize, 0);
 5 ConnectNode(cad, START, rn);
 6 sn = MakeSendNode(cad, addr[0], fragsize, 2);
 7 ConnectNode(cad, rn, sn);
 8 ConnectNode(cad, sn, END);
 9 /* Making R1 and S1 Node */
10 rn = MakeRecvNode(cad, addr[1], fragsize, 0);
11 ConnectNode(cad, START, rn);
12 sn = MakeSendNode(cad, addr[1], fragsize, 2);
13 ConnectNode(cad, rn, sn);
14 ConnectNode(cad, sn, END);
15 ....
16 /* Making Rn and Sn Node */
17 rn = MakeRecvNode(cad, addr[n], fragsize, 0);
18 ConnectNode(cad, START, rn);
19 sn = MakeSendNode(cad, addr[n], fragsize, 2);
20 ConnectNode(cad, rn, sn);
21 ConnectNode(cad, sn, END);
22 /* Issuing CAD Tree */
23 req = IssueCAD(cad);
```

**Fig. 2.** Code generating CAD Tree

is stored in the `addr` memory area that is defined as an array for simplicity. In the code shown in Figure 2, a data structure to store CAD tree is allocated by `InitCAD` in line 2 at first. Then, `RECV` and `SEND` nodes for each fragment are created by `MakeRecvNode` and `MakeSendNode`. Each `RECV`/`SEND` node has corresponding `SEND`/`RECV` node which is created by the code in corresponding rank. After that, each `SEND`/`RECV` nodes and special `START` and `END` nodes are connected using `ConnectNode` to form dependencies shown in Figure 1. Finally, the CAD tree is fixed and sent to the KACC system using `IssueCAD` in line 23. The progression routine, which will be introduced in the following section, starts communication in CAD tree at this time. The user can query the completion of issued CAD using `QueryCAD`.

## 4 Implementation

### 4.1 Structure of KACC System

As shown in Figure 3, the KACC facility consists of three layers: the CAD API, the Progress Engine (PE), and the point-to-point (P2P) communication interface. The latter two layers are implemented inside the Linux kernel as a kernel module. The CAD API described in the previous section is implemented as a user-level library. Using the API, the CAD structure is created in a special memory area shared by MPI processes and the kernel module, so that no structure copy between the user and kernel memory spaces is required.

The Progress Engine (PE) plays two roles. The first role, invoked at the user-level, is to start communication between the CAD structures and to report its progress. The second role, invoked by the P2P communication interface, is to

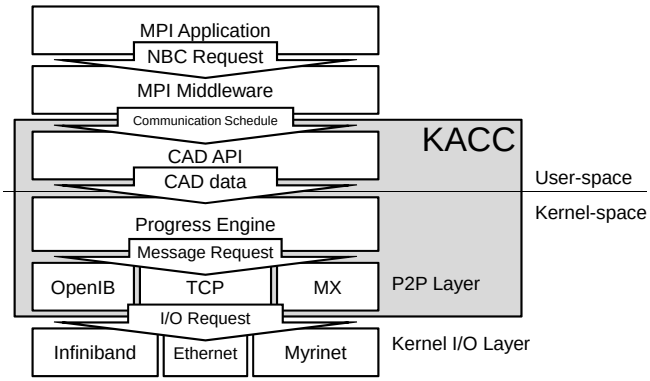



**Fig. 3.** Structure of KACC facility

perform the communication algorithm on the CAD structure. The latter role is implemented as a *tasklet* in the Linux kernel, that is, it runs under the kernel context triggered by the interrupt routine. Thus, the implementation of KACC does not have threads.

The Linux tasklet runs on the same core as the tasklet is scheduled on. If the network interrupts always trigger one specific core, all of the PE routine is executed on that core and fully serialized. In order to avoid this serialization and balance the PE tasklets among CPU cores, we use inter-processor interrupts(IPI) to schedule them on arbitrary core. This method enables each PE event to be executed simultaneously and reduces total execution time. On the other hand, CPU time loss increases due to IPI costs.

The point-to-point (P2P) layer offers communication APIs to the PE layer. The P2P APIs are independent from the network devices and are similar to MPI's non-blocking point-to-point communication APIs. The difference between them is the notification method of completion. The P2P layer invokes PE's call-back routine immediately after completion, instead of offering a polling interface of completion. Thus, the PE's routine is always called at the appropriate timing.

In the current implementation, only the kernel-mode non-blocking TCP has been implemented in the P2P layer, since the recent interconnect devices often have an IP interface [4, 7]. It is possible to implement the P2P layer using native APIs for Myrinet or InfiniBand instead of using IP compatibility layer of these interfaces. The interface between PE and P2P layer is message-oriented, it will fit nicely to the message-oriented communication in Myrinet MX. If we use the native implementation, the communication will be faster and the CPU time loss will be smaller.

Currently, the network connection used in P2P layer is established independently from connection of MPI library in order to distinguish KACC's traffic from other traffics. We are planning to implement communication device interface for MPICH2 or other MPI implementations. This interface also provides

point-to-point non-blocking communication API and all communications in MPI programs are executed by KACC facility.

## 5 Evaluation

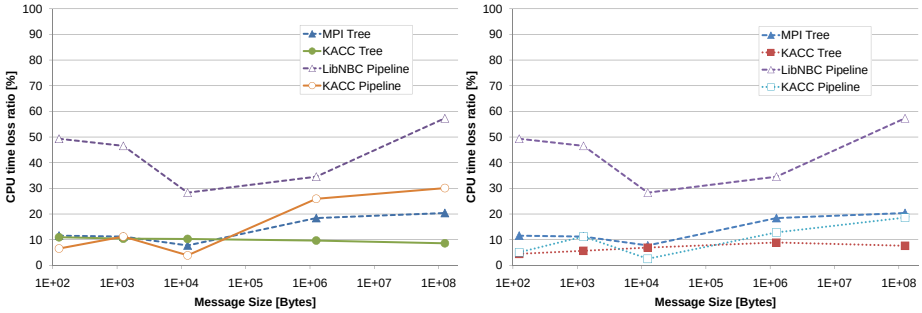
The performance of KACC facility is compared to the other implementation, LibNBC, using a non-blocking broadcast communication in this section. A benchmark program is designed based on the HPL benchmark [8] in order to show how much the non-blocking operation contributes to overlapping communication and computation. The benchmark is named the HPL codelet because it is not a real Linpack benchmark, but is a code snippet from HPL that performs broadcast communication and calculation simultaneously. In this benchmark, after issuing a non-blocking broadcast operation, the fixed amount of calculation is computed repeatedly until the broadcast operation is completed, that is, at every end of calculation, the completion of the operation is examined. The calculation is the matrix multiplication whose size is specified at the run time so that the examination frequency of completion is programmable.

Four implementations of the non-blocking broadcast communication have been carried out. The first implementation is the original HPL non-blocking broadcast communication implementation, using MPI point-to-point operations, whose communication algorithm is based on a binary tree, which will be denoted as *MPI Tree*. The second implementation is the same algorithm written using KACC, which will be denoted as *KACC Tree*. The third implementation is the version used in the LibNBC's library, that is based on pipeline processing, which will be denoted as *LibNBC Pipeline*. The fourth implementation is the same algorithm as LibNBC but written in KACC, which will be denoted as *KACC Pipeline*.

The execution time of the benchmark was measured for the four implementations. The experimental environment consists of eight computing nodes connected by a 1Gbps Ethernet. Each computing node has two dual-core 2GHz Opteron CPUs; thus, there are 32 cores in this cluster. MPICH2/TCP 1.0.6 [1] is used as the base MPI environment.

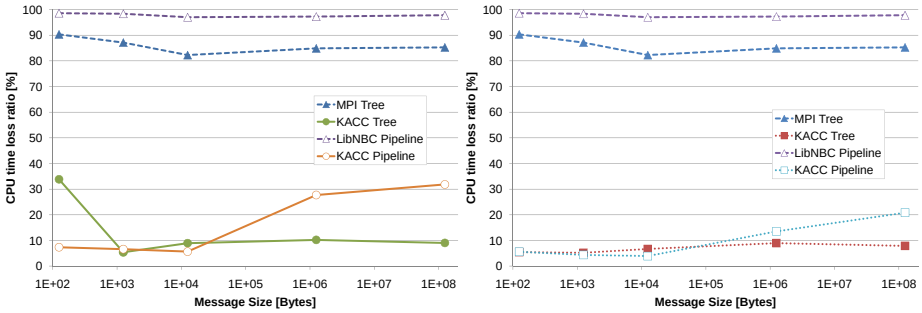
The percentage of CPU time spent for communication is estimated by comparing the number of calculation to the ideal number of calculations without communication. The effect of the frequency of examining the completion of the communication operation is revealed by varying both computation and communication lengths. For the granularity of the computation, two different matrix sizes for calculation are considered: $40 \times 40$ matrices for coarse-grained testing and $4 \times 4$ matrices for fine-grained testing.

The results of CPU time loss are shown in Figures 4 and 5. In order to show the cost of tasklet load-balancing by IPI, the CPU time loss in KACC without load-balancing is also shown in graph (b) of each figures. In the coarse-grained workload shown in Figure 4, the frequency of polling is less, and thus, in this case, the CPU is not consumed by polling communication activity. However, in LibNBC, 28.3 to 57.3% of CPU time is always spent for the communication




(a) With load balancing using IPI (b) Without load balancing using IPI

**Fig. 4.** CPU time loss during broadcast with coarse-grained workload



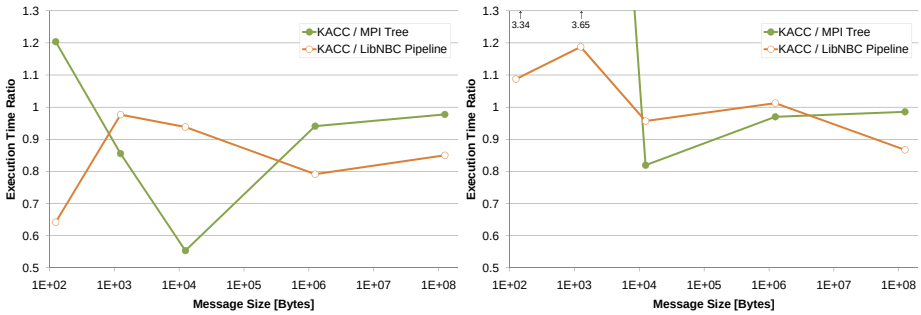

(a) With load balancing using IPI (b) Without load balancing using IPI

**Fig. 5.** CPU time loss during broadcast with fine-grained workload

thread, and, therefore, the effect from the communication computation overlap is relatively small. On the other hand, in the same pipeline algorithm with KACC facility, the CPU time spent for communication is limited to about half of the case with LibNBC, and about half of the CPU time is due to the cost from IPI handling.

When the granularity of computation is fine as shown in Figure 5, the frequency of tests for the communication's completion is high. In this case, 97.0% of the CPU time is lost to the communication thread in LibNBC. On the other hand, KACC's CPU loss rate is limited to 31.8% even if load-balancing using IPI is enabled. This result shows that the users can continue calculation effectively during collective communication under the KACC facility.

The ratio of total execution time in broadcast on KACC compared to normal MPI tree and LibNBC pipeline implementations is shown in Figure 6. If the ratio is smaller than 1, corresponding method is faster than the compared case, MPI Tree or LibNBC Pipeline. In the best case, it took 2.15 milliseconds for 12.5kB broadcast in MPI and 1.19 milliseconds in the same algorithms with KACC facility and thus execution time ratio is calculated as 0.55. Similarly, it took 22.6




(a) With a coarse-grained workload (b) With a fine-grained workload

**Fig. 6.** Total execution time ratio in non-blocking broadcast

milliseconds for 1.25MB broadcast in LibNBC and 17.9 milliseconds in the same algorithms with KACC facility and execution time ratio is calculated as 0.79. For various communication size and workload granularity, the execution time in the KACC is 55 to 98% of that in MPI implementation and 79 to 101% of that in LibNBC except for small message size. However, if the message size is small, KACC implementation is slower than normal MPI or LibNBC implementation. We have to improve the performance of small messages in the future.

## 6 Conclusions

Non-blocking collective operations have been proposed and implemented. If non-blocking collective operations are performed asynchronously with the main computation, the communication latency can be hidden, and more time can be made available for computation. However, these implementations are not always scaled due to the extra cost of asynchronous communication management using threads.

This paper proposed the KACC facility, which performs collective communications in the OS kernel's interrupt context. Since non-blocking collective operations are performed in the kernel interrupt context, no extra threads are introduced. Furthermore, a collective operation is progressed immediately when a message for the operation arrives. These two features contribute to the application program spending more CPU time in computation.

A benchmark was used to test four different implementations, including the thread implementation in LibNBC [2,3], of the non-blocking broadcast operation. The results show that the thread implementation, LibNBC, consumes almost half of the CPU when less frequent polling of completion is requested or 97% of CPU time when frequent polling is performed, while KACC consumes only 10 to 30% of CPU time for both cases while keeping its execution time up to 79 to 101% of that in LibNBC.

There are four limitations in the current implementation. Firstly, the user-defined operations have not yet been considered. The execution of operations must be safe in terms of security. There are two potential approaches to this

problem: interpreter and thread approaches. In the interpreter approach, the function is compiled to virtual machine code, and this code is interpreted in the kernel. Another approach is to introduce a thread for executing the operations. In this approach, an extra thread is created. However, unlike the thread for communication progress, the extra thread is only activated during the operation of collectives, and the overhead is expected to be small.

Secondly, the current implementation has not yet been optimized for the intra-node communication, since intra-node communications are handled by the TCP connections. If messages are directly copied by the memory copy operation, performance is improved.

Thirdly, the P2P layer supports only TCP in the current implementation. Porting to other hardware, such as InfiniBand and Myrinet, is being considered.

Fourthly, the current implementation is slow if the messages are small. Improving the performance for small messages is strongly needed. We may recommend not to use non-blocking collective communications if the message size is small. In this case, we should show reasonable method to determine the threshold of message size to use non-blocking collective communications.

## References


1. Argonne National Laboratory: MPICH2 : High-performance and widely portable MPI. http://www.mcs.anl.gov/research/projects/mpich2/
2. Hoefler, T., Lumsdaine, A.: Design, Implementation, and Usage of LibNBC. Tech. rep., Open Systems Lab, Indiana University (Aug 2006)
3. Hoefler, T., Lumsdaine, A., Rehm, W.: Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI. In: Proceedings of the 2007 International Conference on High Performance Computing, Networking, Storage and Analysis, SC07. IEEE Computer Society/ACM (Nov 2007)
4. Kashyap, V.: IP over InfiniBand (IPoIB) Architecture. RFC 4392 (Informational) (April 2006), http://www.ietf.org/rfc/rfc4392.txt
5. MPI Forum: Message passing interface. http://www.mpi-forum.org/
6. MPI Forum: MPIplans - an alternative for all other collectives proposals? https://svn.mpi-forum.org/trac/mpi-forum-web/wiki/MPIplans
7. Myricom, inc.: IP over myrinet. http://www.myri.com/scs/documentation/mug/ip/
8. Petitet, A., Whaley, R.C., Dongarra, J., Cleary, A.: HPL - a portable implementation of the high-performance linpack benchmark for distributed-memory computers. http://www.netlib.org/benchmark/hpl/