Prototyping of Offloaded Persistent Broadcast on Tofu2 Interconnect

Yoshiyuki Morie[1,2], Masayuki Hatanaka[1], Masamichi Takagi[1], Atsushi Hori[1], and Yutaka Ishikawa[1]

1. Riken Advanced Institute for Computational Science, Japan
2. Email: yoshiyuki.morie@riken.jp

Abstract

With the increasing scale of parallel computers, it has become more important to reduce communication time. Overlapping computation and communication is one effective method for hiding communication delay. Although standard non-blocking collective communication is an overlap method, it requires generating a communication command sequence for each collective communication. In contrast, persistent non-blocking collective communication can generate the sequence at initialization and reuse it at the start of collective communication. Moreover, if the sequence can be offloaded to a network device, more efficient execution is possible without using CPU cycles.
In this paper, a persistent non-blocking broadcast is implemented using the offloading functionality of the Tofu2 interconnect on the Fujitsu FX100 supercomputer, the successor to the K computer. We report the performance improvement by offloading persistent non-blocking collective communication in a real machine.

Persistent collective communication

- MPI_(NAME)_init is an initialization call and performs beforehand scheduling of the collective communication by considering dependency.
- MPI_Start() starts the collective communication following its scheduling.
- MPI_Wait() completes the collective communication.
- MPI_Request_free() releases the resources acquired in initialization.

```
MPI_Bcast_init(args, req)

for i = 0 to N do
        MPI_Start(req)
        Compute()
        MPI_Wait(req)
end do

MPI_Request_free(req)
```

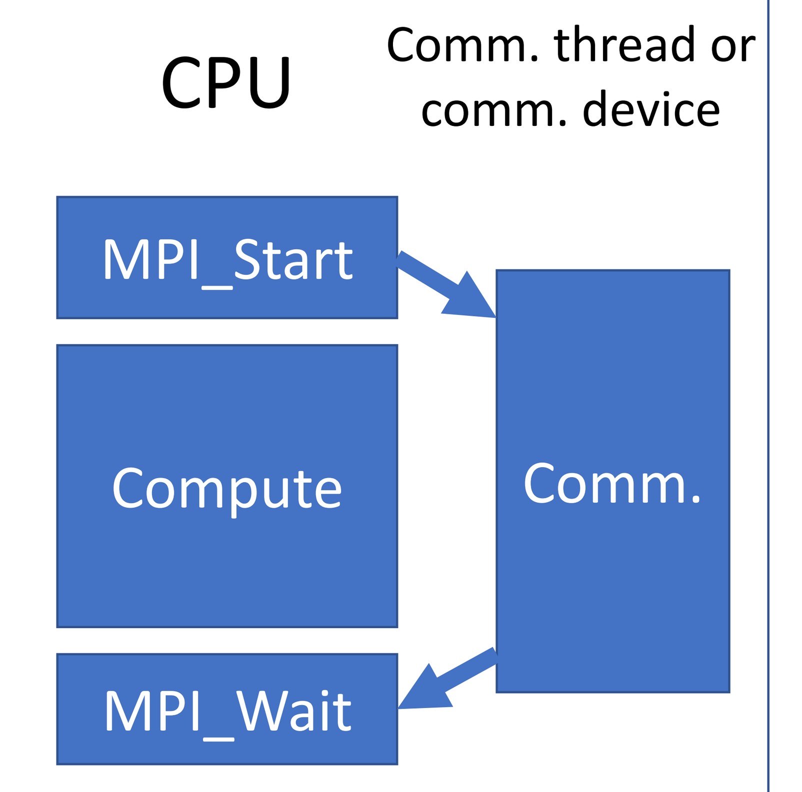


Figure 1. Overview of persistent broadcast

If communication device or communication thread can start the communication without using a processor unit, then computation and communication are able to overlap between MPI_Start() and MPI_Wait().

Offloading mechanism on Tofu2 [1]

- Tofu2 has the session mode CQ (SMCQ) which includes a mechanism for triggering the issue of communication commands.
- As shown in Figure 2, SMCQ consists of the transmit order queue (TOQ) and three pointers: the producer, consumer, and scheduling pointers. The TOQ descriptor has the information for RDMA communication such as command type, source, and destination.
- The producer pointer points to the end of the command sequences enqueued by the application. The consumer pointer indicates the end of the executed commands.
- The consumer pointer cannot overtake the corresponding scheduling pointer on TOQ. This can slow the progress of communication.
- As shown in Figure 3, when an RDMA engine processes an incoming RDMA operation packet, the scheduling pointer can be advanced by the session progress step (SPS) fields of the packet.
- Therefore, SMCQ can progress or stop the sequence of commands by only an incoming RDMA operation packet without using CPU cycles.
- SMCQ can provide a full offloading mechanism; the offloaded MPI neighborhood collectives are implemented using this mode [2].

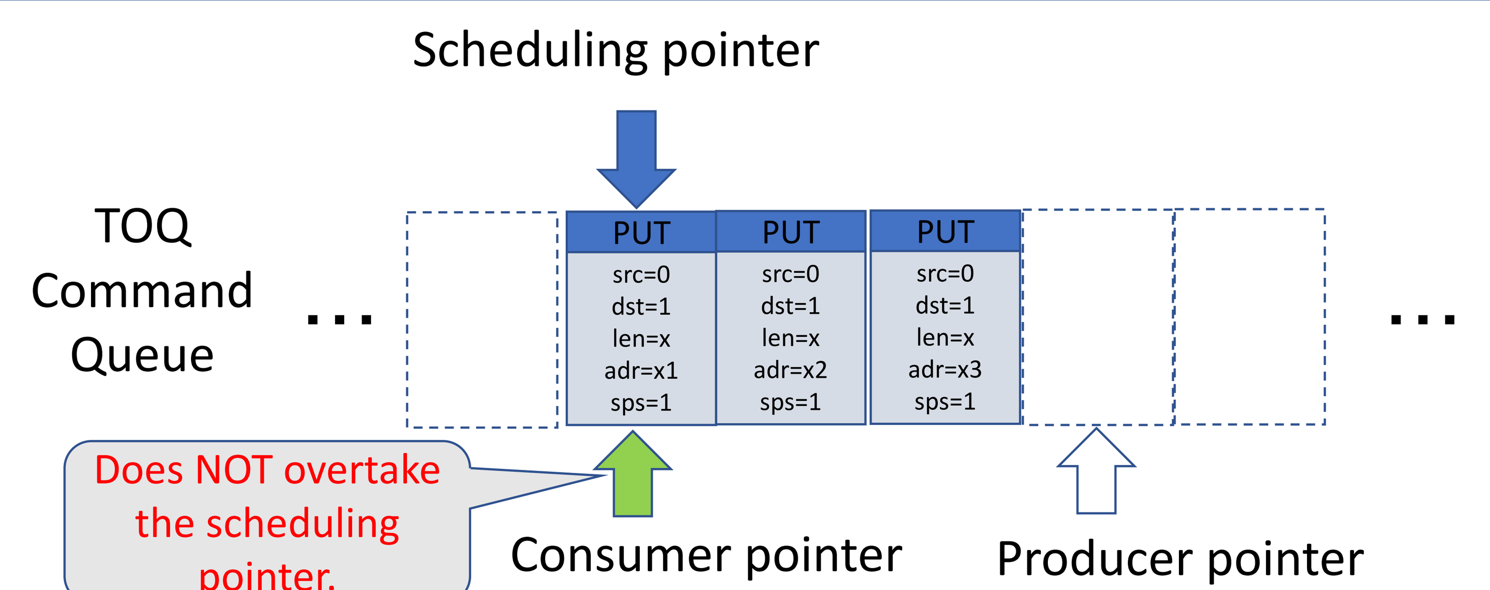


Figure 2. Overview of session mode CQ.

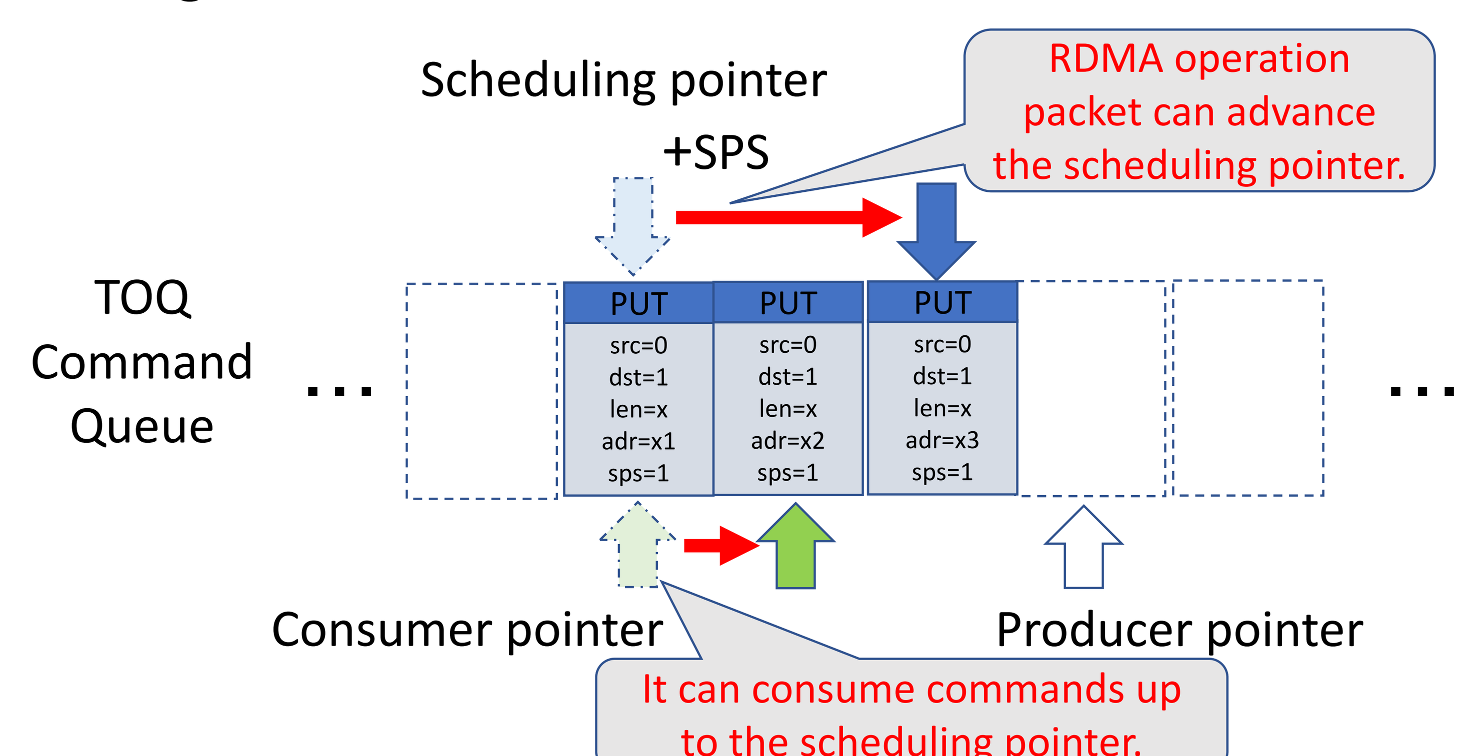
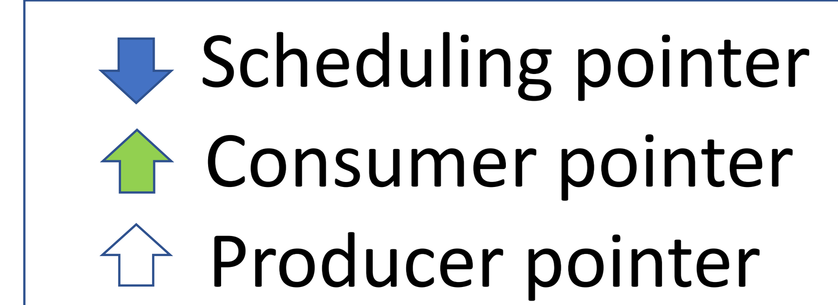


Figure 3. Behavior of the pointers in TOQ when an RDMA operation packet arrives.

Implementation of offloaded persistent broadcast

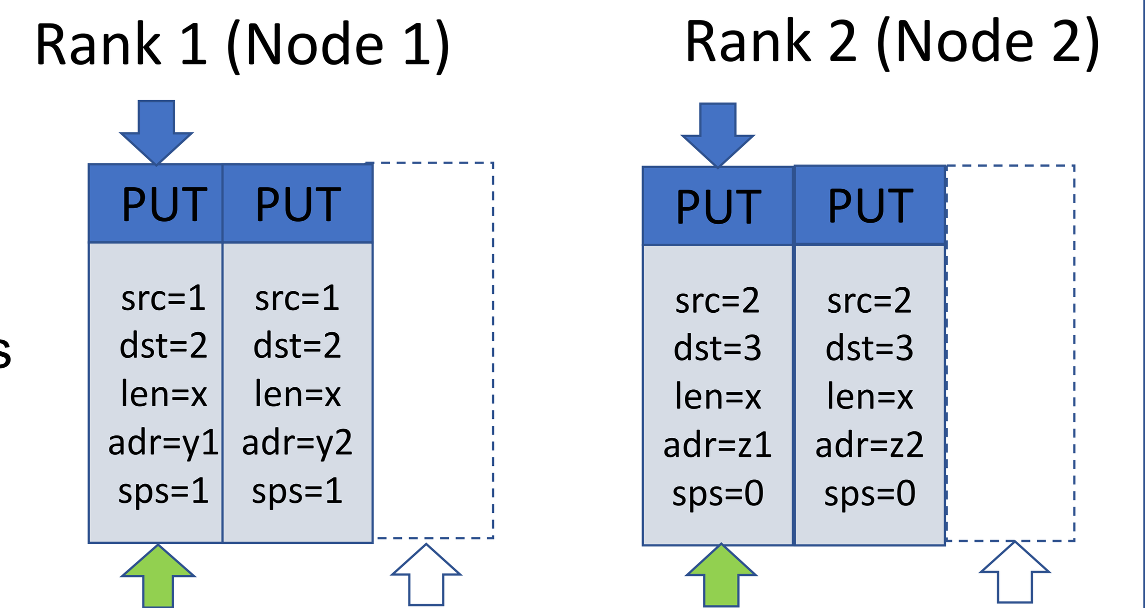
As shown in Figure 4, broadcast data are divided into three fragments, each of which is transmitted using a pipelined broadcast using three network devices on Tofu [3][4]. Blue Gene/L uses a similar algorithm[5].



Single-path pipelined broadcast offloaded using SMCQ.

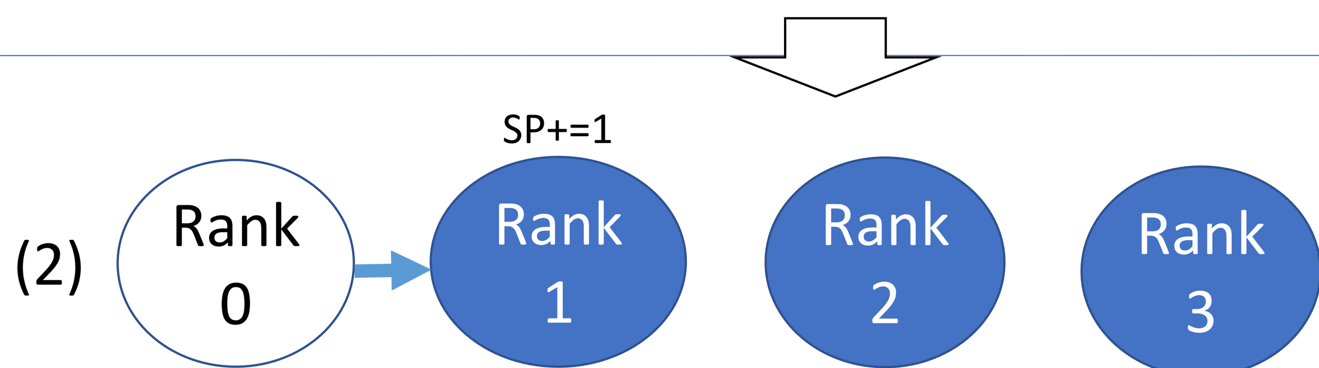
A pipelined broadcast, used by the Trinaryx3 algorithm, is implemented using SMCQ. Here we assume that four processes participate in the broadcast and that its message is packetized into two segments.

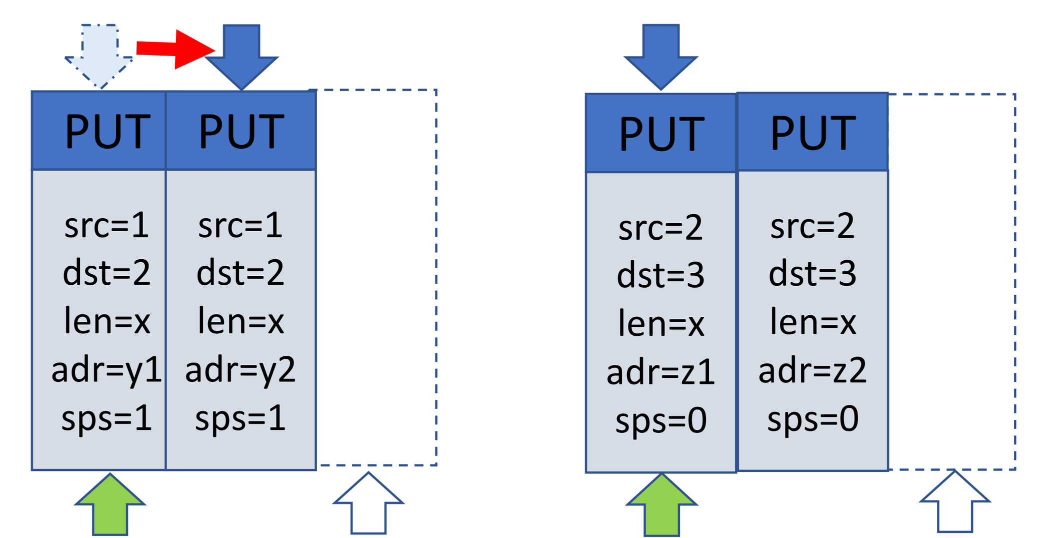
(1)



- After enqueueing command sequences into TOQ except ranks 0 and 3, all processes must report ready to rank 0. A simple way of implementing this is to use a barrier function of Tofu2, but our implementation also uses SMCQ. Due to the space limitations of this presentation, this synchronization method is not described.

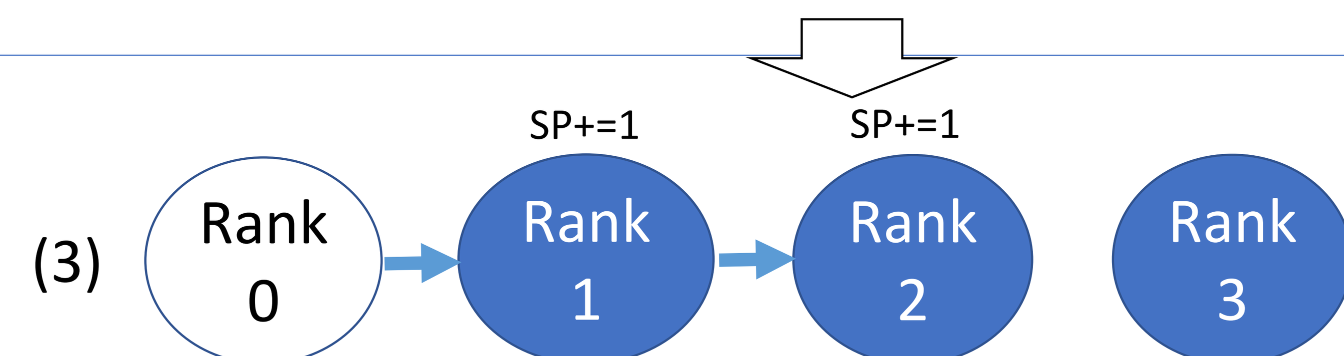
(2)

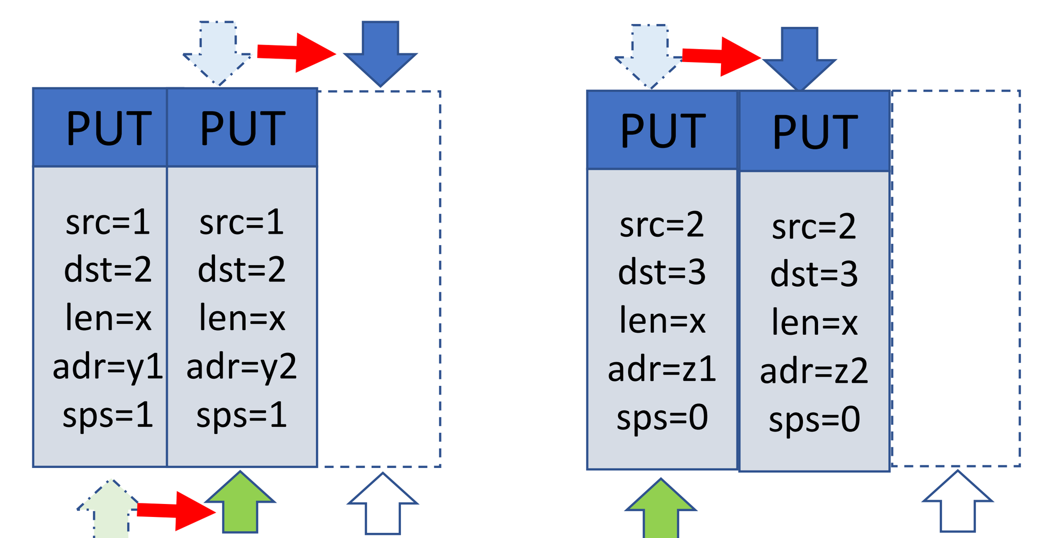




- After rank 0 recognizes all process's as ready, rank 0 enqueues two commands in CQ whose mode is normal (shown on the right)
- Because the CQ in rank 0 is normal mode, commands enqueued in CQ are handled immediately.
- The consumer pointer in rank 0 is advanced after sending one segment to rank 1.
- When rank 1 receives this segment, data are stored and the scheduling pointer is incremented by 1 because the SPS of the command issued in rank 0 is 1.
- Then, rank 1 is able to issue the first command of CQ, e.g., send one segment to rank 2.

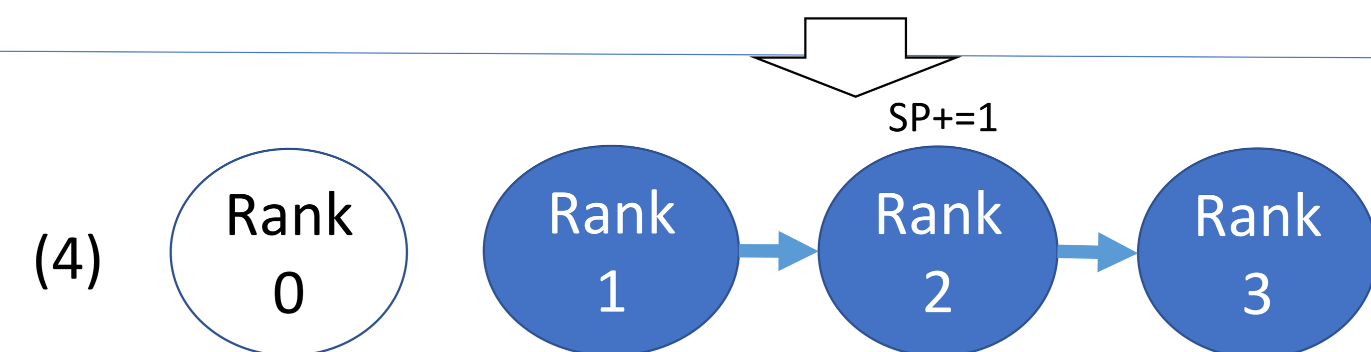
(3)

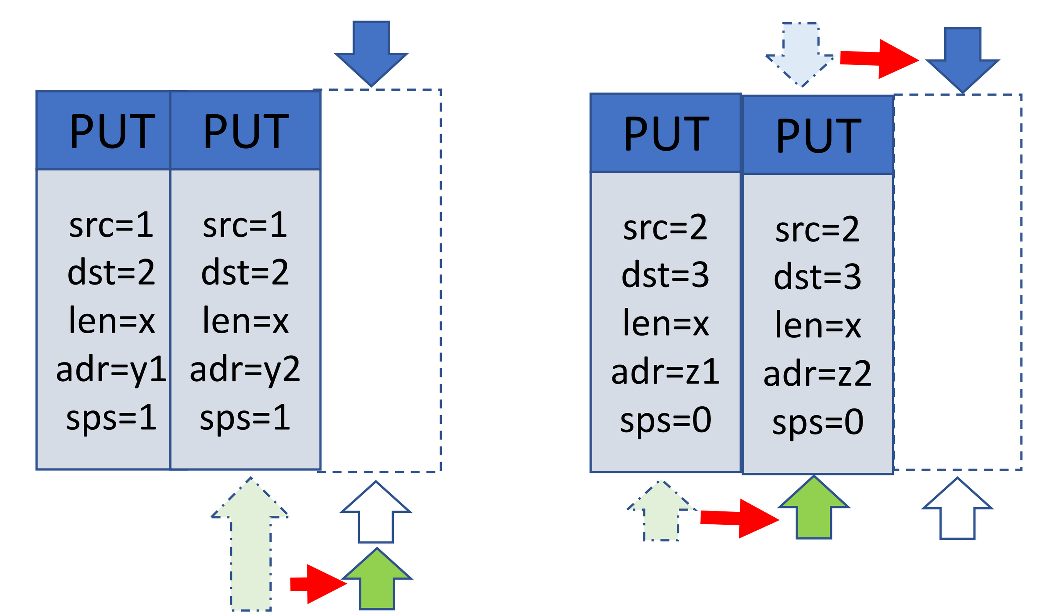




- The consumer pointer in rank 0 is advanced after sending the second segment to rank 1.
- When rank 1 receives this segment, data are stored and the scheduling pointer is incremented by 1 because the SPS of the command issued in rank 0 is 1.
- Then, rank 1 is able to issue the first command of CQ, e.g., send one segment to rank 2.
- The consumer pointer in rank 1 is advanced after sending one segment to rank 2.
- When rank 2 receives this segment, data are stored and the scheduling pointer is incremented by 1 because the SPS of the command issued in rank 0 is 1.
- Then, rank 2 is able to issue the first command of CQ, e.g., send one segment to rank 3.

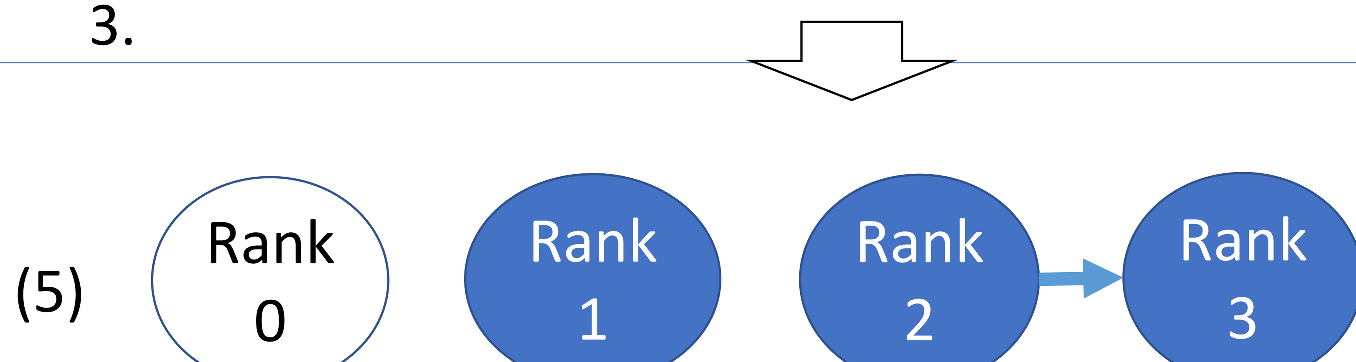
(4)

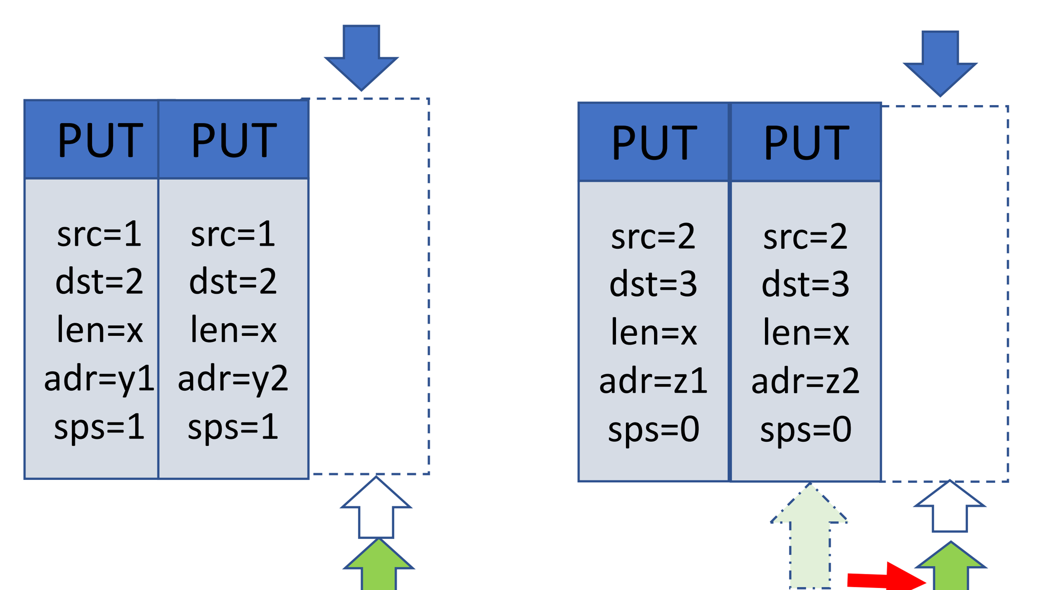




- The consumer pointer in rank 1 is advanced after sending the second segment to rank 2.
- When rank 2 receives this segment, data are stored and the scheduling pointer is incremented by 1 because the SPS of the command issued in rank 1 is 1.
- Then, rank 2 is able to issue the first command of CQ, e.g., send one segment to rank 3.
- The consumer pointer in rank 2 is advanced after sending one segment to rank 3.

(5)





- The consumer pointer in rank 2 is advanced after sending one segment to rank 3.

Implementation of offloaded persistent broadcast for Tofu2

- The Trinaryx3 algorithm is an extension of the Trinary algorithm in which broadcast data are divided into three fragments, each of which is transmitted using the Trinary algorithm as shown in Figure 4.
- Tofu2 has multiple DMA engines, called TNIs, each of which has CQs.
- The Trinaryx3 algorithm utilizes three TNIs.
- A Trinary algorithm is implemented in a way similar to single-path pipelined broadcast except that the data transmission branches. The three pipeline paths of the trinary tree are mapped to the 3D torus in Figure 4. Each pipeline communication is offloaded and executed by SMCQ.

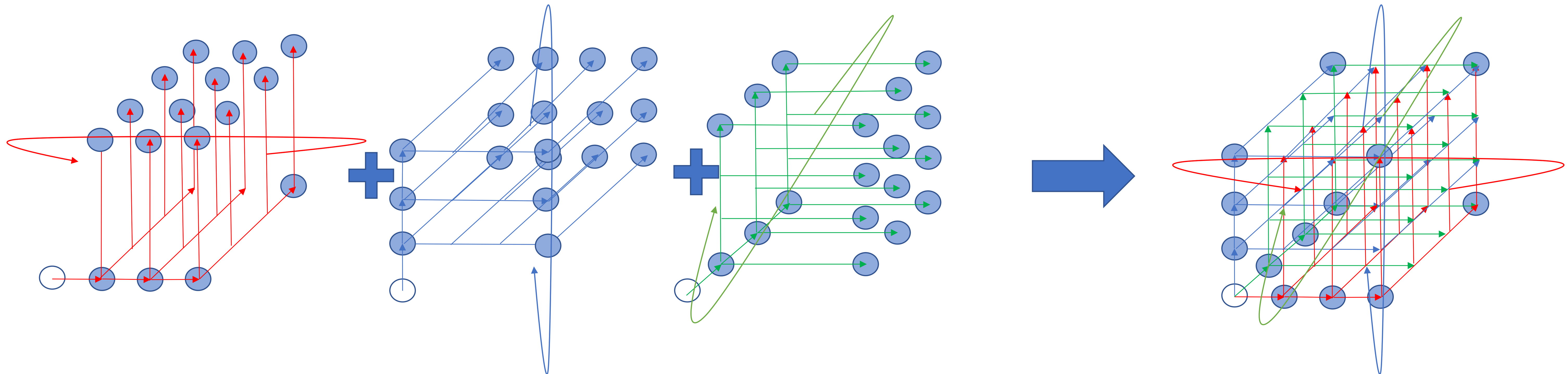


Figure 4. Pipeline paths with Trinaryx3 algorithm.

Performance evaluation

- The performance of offloaded persistent broadcast is compared with a non-blocking broadcast implementation involving progressing of the same algorithm, Trinaryx3. Table 1 shows the evaluated environment and Figures 5 and 6 show results.
- The offloaded persistent broadcast is better than the non-blocking broadcast because the persistent non-blocking collective is able to complete creating the sequence of communication commands and exchanging the remote memory information at MPI_Bcast_init().
- Non-blocking broadcast requires processing equivalent to MPI_bcast_init() at every execution. Therefore, its performance deteriorated due to the associated cost.

Table 1. Evaluation environment.

Machine name	Fujitsu PRIMEHPC FX100
CPU	SPARC64XIfx 1.975 GHz 32 cores
Memory	32 GB
# of nodes	36
Network	Tofu2 (12.5 GB/s)
Topology	6-dimensional mesh/torus

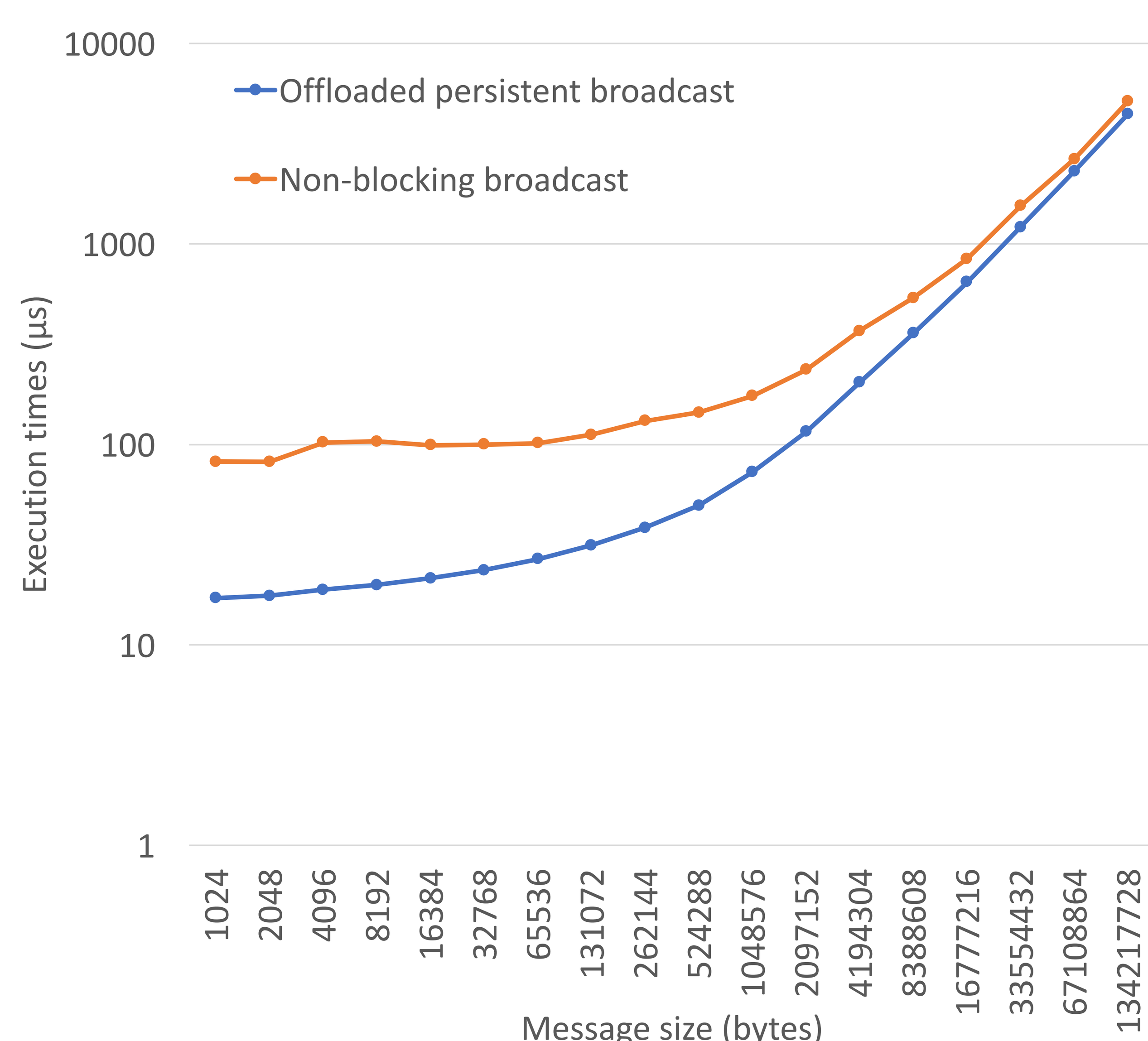


Figure 5. Execution time on Tofu2 (12 processes).

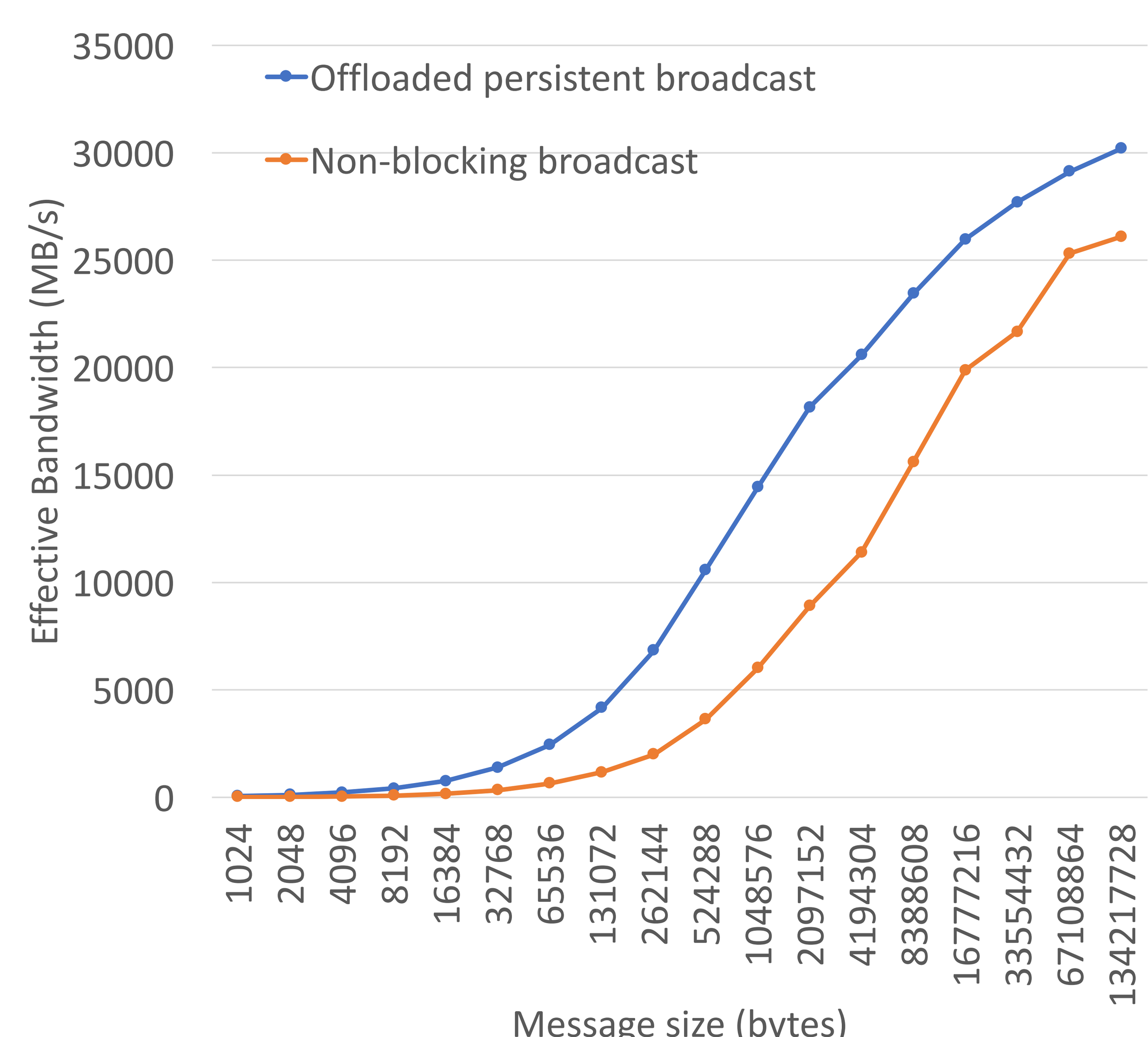


Figure 6. Effective bandwidth on Tofu2 (12 processes).

Execution time of MPI_Start

- The execution time of MPI_Start() is 2-8 µs (Figure 7).
- The MPI_Start() execution time is proportional to the number of segments but is extremely short.
- The execution time of MPI_Bcast_init() is 60-251 µs, which is a relatively high cost (Figure 7).

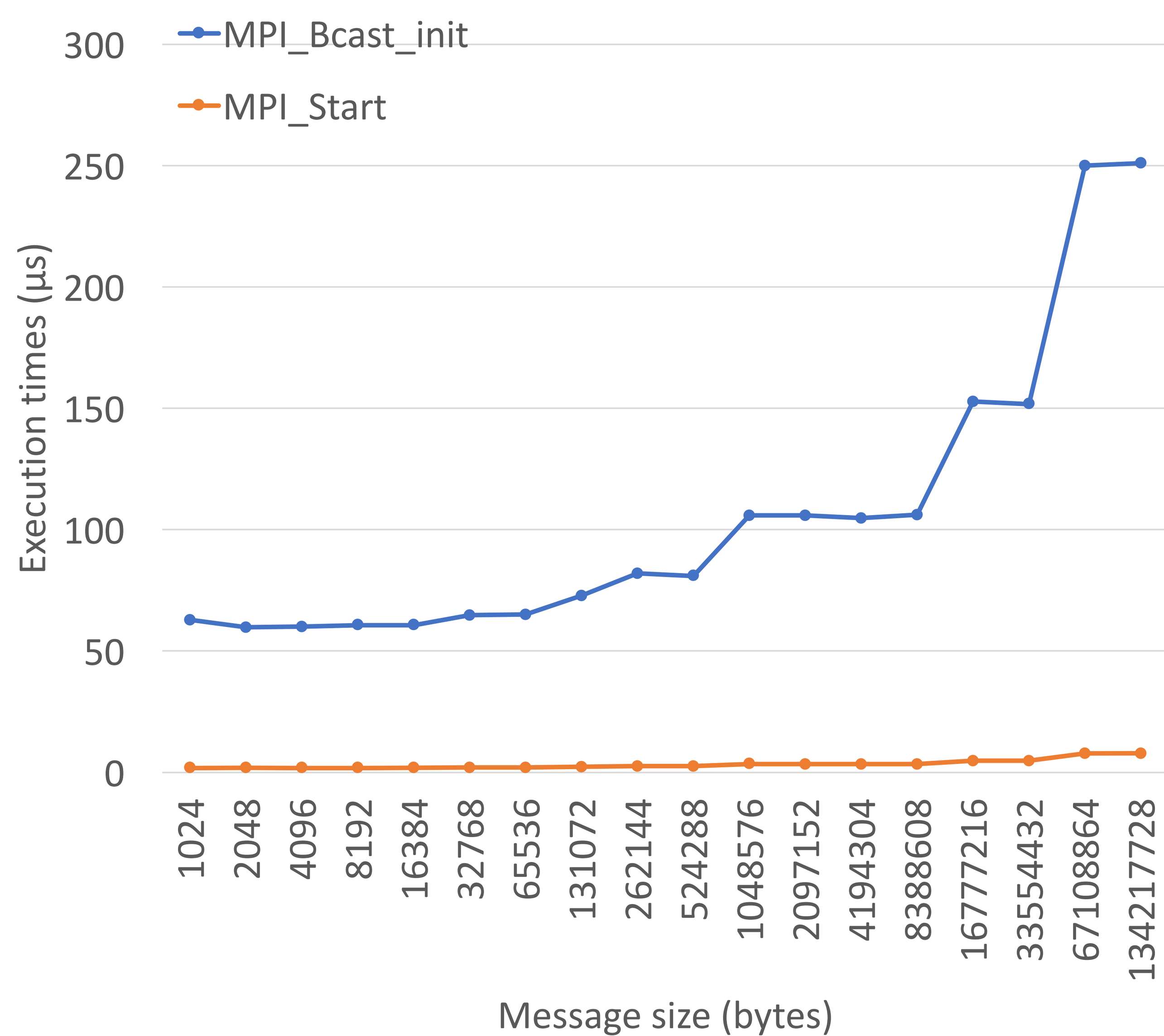


Figure 7. Execution time of MPI_Bcast_init and MPI_Start (12 processes).

Execution time of MPI_Wait

Figure 8 shows the execution time of MPI_Start followed by MPI_Wait without any computation. The execution time of MPI_Wait in this case is just the wait time for completion of offloaded communication. Thus, the execution time of MPI_Wait is the amount of time to overlap computation between MPI_Start and MPI_Wait. This is about 4452 µs of computation time in the case of a 128 MB broadcast.

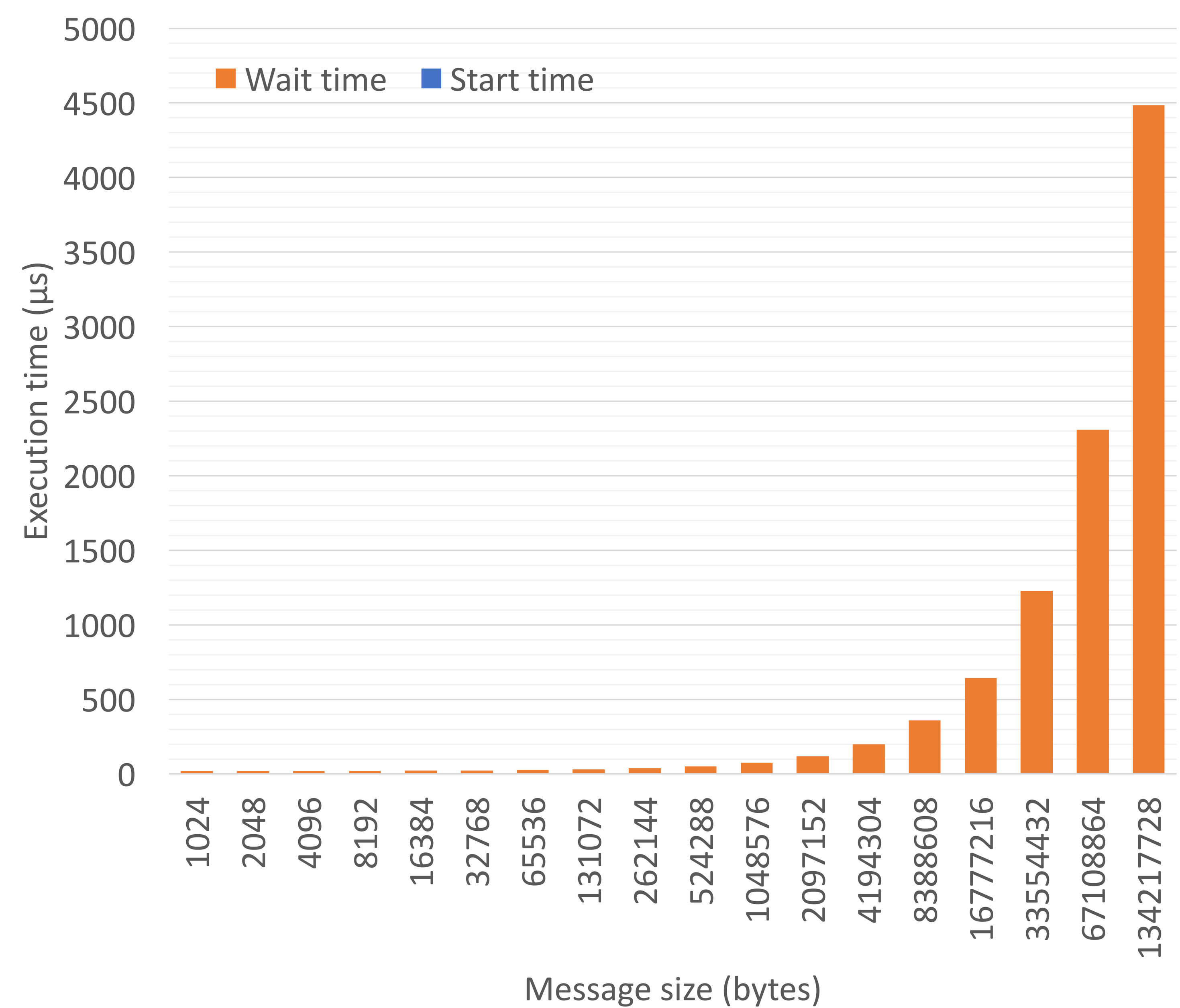


Figure 8. Execution time of MPI_Start followed by MPI_Wait without any computation (12 processes).

Conclusions and future work

- Conclusions
  - The effectiveness of offloading the persistent non-blocking collective communication using Tofu2 interconnect is shown.
    - The offloaded non-blocking broadcast outperforms the non-blocking broadcast.
    - The relationship between the generation of the communication command sequence and the cost of offloading the command sequence of collective communication is shown.
- Future work
  - The evaluation of the implemented offloaded persistent non-blocking broadcast will be evaluated on a large-scale machine.
  - The other collective communications will be ported into offloaded persistent communications on Tofu2.

References

[1] Y. Ajima, T. Inoue, S. Hiramoto, S. Ando, M. Maeda, T. Yoshikawa, K. Hosoe, and T. Shimizu, “Tofu Interconnect 2”, In 2014 IEEE 22nd Annual Symposium on High-Performance Interconnects, pp. 57-62, Aug. 2014.

[2] M. Hatanaka, M. Takagi, A Hori, and Y. Ishikawa, “Offloaded MPI persistent collectives using persistent generalized request interface”, In EuroMPI/USA 2017, Sep. 2017 (to appear).

[3] N. Shida, S. Sumimoto, and A. Uno, “MPI library and low-level communication on the K computer”, Fujitsu Scientific & Technical Journal, Vol. 48, No. 3, pp. 324-330, Jul. 2012.

[4] T. Adachi, N. Shida, K. Miura, S. Sumimoto, A. Uno, M. Kurokawa, F. Shoji, and M. Yokokawa, “The design of ultra scalable MPI collective communication on the K computer”, Computer Science - Research and Development, Vol. 28, Nos. 2-3, pp. 147-155, May 2013.

[5] G. Almasi, 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 Proc. of SuperComputing 2005, pp. 253-262, 2005.