LibPhotonNBC: An RDMA Aware Collective Library on Photon

Udayanga Wickramasinghe[1], Ezra Kissel[1], and Andrew Lumsdaine[1,2]

[1]Department of Computer Science, Indiana University, USA, {uswickra,ezkissel}@indiana.edu
[2]Pacific Northwest National Laboratory, Richland WA, USA andrew.lumsdaine@pnnl.gov

December 12, 2016

Abstract

Collectives are a widely utilized programming construct for assembling parallel communication in High Performance Computing (HPC) applications. They are categorized into either non-blocking and blocking invocation, depending on whether the control is returned immediately to a user or not. In particular, non-blocking variety allows applications to perform other useful computations while hiding the network latency of the underlying communication operations. We present LibPhotonNBC, a low-level RDMA aware collective library that enables execution of communication primitives of a collective using one-sided memory semantics. We also utilize LibNBC as a vehicle to support non-blocking collectives algorithms for Photon. This document intends to describe the inner workings of LibPhotonNBC collective library – design, implementation, usage and our initial results.

1 Introduction

Collective communication is a term first popularized by the MPI specification and its subsequent implementations and applications usage. Performance

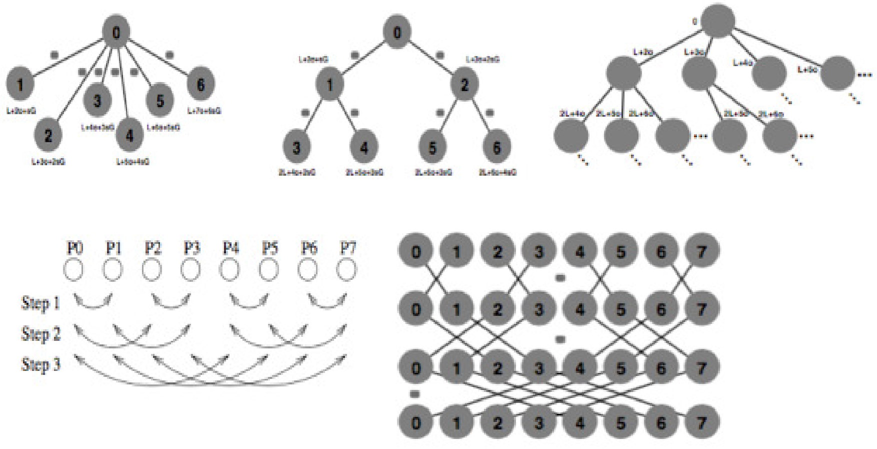

Figure 1: Different collective algorithms in use. (top) – rooted collectives patterns (bottom) – non-rooted collective patterns

characteristics of a synchronizing group communication routine such as a collective involves multitude of factors - scale of the application, network and system features and bottlenecks, etc. Thus each collective operation has to implement multiple algorithms to align with the growing needs of scalability. As shown by Figure 1, different algorithms such as mesh, hypercubes, trees[1, 3, 12] and algorithms catered towards various platform architectures [8, 9] are implemented by different run-time libraries. MPICH [13] and other collective libraries [14, 2, 4, 11] have presented detailed analysis on available collective algorithms and their characterization based on performance considerations such as message sizes, number of processes, data locality and network utilization.

Hoefler et,al's work on LibNBC [6, 7, 5] enabled asynchronous progression of collective communication to better utilize communication and computation overlap of respective applications. LibNBC and its derivatives have since made its way into many MPI implementations and the interface definitions of the MPI Specification.

Photon[10] offers a rich and intuitive programming abstraction in terms of completion events for embedding RDMA aware network transactions in a parallel application. Photon Collectives Implementation is an effort to integrate collective operations and algorithms into Photon RDMA Library. Major portion of the current LibPhotonNBC Library builds on top of libNBC.

There are three primary reasons why this design decision was made. Firstly, almost all regular collective patterns are well known in the literature, and their performance characteristics are adequately studied. Therefore most of the collective algorithms and patterns are boilerplate code that we can reuse. For this purpose LibNBC serves us well – it has a sufficiently large number of built in collective algorithms.

Secondly we think that LibNBC allows a flexible internal layer which enables seamless integration of new collective patterns and operations and allows easy performance tuning of collectives. It comes in the form of creation of collective schedules – A dependency graph of send/recv operations for each collective operation created at the initialization time. Since for a single collective operation, the same schedule can be seen repeating many times during an applications lifetime, performance cost of creating schedules will approach zero. Third and most importantly we intend to provide true asynchronous progression of collectives with LibPhotonNBC which perfectly matches with LibNBC's intent. All these aspects contribute as positives for using LibNBC as a substrate for LibPhotonNBC Photon Collective Implementation.

Outline The remainder of this article is organized as follows. Section 2 gives an account of overall design and implementation of the LibPhotonNBC Library. We will discuss various design aspects as well as the interfaces provided by the library in detail. Next we discuss the details of experiments performed using Photon collectives and micro-benchmark performance results in Section 3. Finally, Section 4 presents our position on the current state of LibPhotonNBC library and possible improvements for the future.

2 Implementation

Main design objective of the LibPhotonNBC Library has been to introduce collective operations to the existing Photon RDMA Library. We try to achieve this with pluggable interfaces and easily programmable high-level **API**s for LibPhotonNBC end users. Each algorithm contains a point to point to communication schedule. Thus each primitive communication routine in LibPhotonNBC maps directly to a Photon RDMA Write operation – Photon **PWC** (Put witch completion) [10]. A Photon **PWC** operation is an RDMA remote memory *write* operation accompanied by a user-defined *local* and *remote* completion identifier. Photon pushes a *local* completion event to its local internal event queue whenever a RDMA operation has successfully completed (ie:, whenever it is safe to re-use the local buffer again). Similarly, at the remote end, Photon pushes a *remote* completion event to its internal event queue, whenever message has been successfully received. We use these Photon specific identifiers to transport necessary information on the state of the collective schedule.

A. *Collective Interfaces*

Photon Collective implementation encompasses an internal interface for integrating arbitrary number of back-end implementations. The Interface **PHOTON_COLL_IFACE_NBC** correspond to LibNBC library based back-end. External interface corresponds to the user facing API – we discuss these in the next sections.

Each collective call site primarily involves invoking two interfaces namely **photon_collective_init()** and the respective collective operation **photon–_collective_run()**. As the name suggests the API **photon_collective_init()**, prepares a Photon request required to begin a collective operation. It also need photon local completion identifier which will be used at a later time for probing for respective operation's completion.

Listing 1 shows a simple example of a photon collective invocation. API **photon_collective_run()** accepts number of parameters for an operation depending on a particular operation. First, this invocation requires a initialized photon request (as a parameter) for storing information of a collective operation in-flight. It also accepts input and output parameters depending on the operation invoked[1]. As stated earlier all LibPhotonNBC collective invocations are non-blocking and thus return immediately.

Photon checks completion of its RDMA operations via completion events. Similarly LibPhotonNBC operations need to probe for the local completion[2] of the respective collective request. Listing 2 report the code listing for checking completion of a collective. The function **run_nonblocking_progress** accepts an additional function pointer argument which allows simultaneously

[1]Each collective operation have different argument sizes and a optional computation function

[2]Each collective reach a local view of completion when all its send/recv and compute operations are finalized. This is different from a global view of completion where we need to guarantee that all nodes have completed – thus require barrier synchronization.

progressing the network and the compute task. LibPhotonNBC doesn't restrict probing to a single thread, therefore user is entitled to use as many threads as required to progress the collective.

```
// compute heavy kernel/algorithm
extern void compute_kernel(short *done);

application_kernel(){
  ...
  // input/output select
  paramters.sendbuf = in;
  paramters.recvbuf = out;
  // data type select
  paramters.datatype = PHOTON_INT64
  // global non blocking join
  int op = PHOTON_COLL_OP_SUM;
  paramters.op = op;

  // track coll request
  photon_rid req;
  // identifier for completion
  photon_cid lid;
  lid.u64 = PHOTON_TAG;

  //initialize an Allreduce request and a completion id
  rc = photon_collective_init(PHOTON_COLL_IALLREDUCE, lid, &req);
  if (rc != PHOTON_OK) {
    fprintf(stderr, "Could not initialize collective operation\n");
        goto err_exit;
  }

  rc = photon_collective_run(req, &parameters);
  if (rc != PHOTON_OK) {
    fprintf(stderr, "Could not execute Allreduce join\n");
        goto err_exit;
  }
  run_nonblocking_progress(lid, compute_kernel);
}
```

Listing 1: **Executing a Photon collective operation**

```
run_nonblocking_progress(photon_cid cid, tast_t task){
 photon_cid req;
 short task_done = 0;
 do {
    photon_probe_completion(PHOTON_ANY_SOURCE, &flag, NULL, &req,.., NULL,
                                     PHOTON_PROBE_ANY);
    if ((flag > 0) && (req.u64 == cid.u64)) {
        //collective operation is completed at this point
         while(!task_done){
           task(&task_done);
         }
         break;
    }
    // overlap computation task
    task(&task_done);
   }while(1)
}
```

Listing 2: **Non blocking progress with a Photon collective operation**

Photon internal collective Interface allows many collective implementations to co-exist in the library. Users can configure which implementation to choose, with the photon configuration parameter **photon_config_t#coll**. The interface definition for collectives in Photon is reported in Listing 2. Here interfaces for specific collective operations are straightforward, however others require some explanation. Interface **init()** and **comm_create()** are for initialization purposes. Photon collective implementation register pinned buffers using respective RDMA back-end implementations such as **IBVerbs**, **LibFabric**, **Cray-ugini**, etc during initialization and also perform additional tasks required to setup a group communicator to facilitate collective operations. The act of probing for completion events for collective communication is served by the API **probe()**.

All requests and completion events of a collective operation are internal to Photon and therefore opaque to the user (ie;, These events won't be visible to Photon end users). Therefore all control events pertaining to a collective operation are handled through **cid_handler()** Interface.

```
typedef struct photon_coll_interface_t {
  // collective internal functions..
  int   (*init)       (photonConfig cfg);
  int   (*comm_create)(void *c, void *active, int num_active, int total);
  void* (*comm_get)   ();
  int   (*probe)      (int proc, int *flag, photon_cid *c);
  int   (*cid_handler)(int proc, pwc_command cmd, photon_cid cid,
                     void *data, int size);

  // collective operations ..
  int   (*barrier)    (void *comm);
  int   (*ibarrier)   (void *comm, photonRequest req);
  int   (*allreduce)  (void *in, void *out, int count, void *datatype,
                                                     void *op, void *comm);
  int   (*iallreduce) (void *in, void *out, int count, void *datatype,
                                        void *op, void *comm, photonRequest req);
  int   (*scan)       (void *in, void *out, int count, void *datatype,
                                                   void *op, void *comm);
  int   (*iscan)      (void *in, void *out, int count, void *datatype,
                                      void *op, void *comm, photonRequest req);
  ...
} photon_coll_interface;
```

Listing 3: **Photon collective Interface**

B. *Collective Progression*

The two key functions that perform network progression of a collective operation in LibPhotonNBC includes collecting Photon events and driving the collective schedule accordingly. The function **nbc_cid_handler()** collects all *local* and *remote* completion event descriptors from CQs and forward them to a internal FIFO queue. The **nbc_probe()** function would then probe this queue and check for completion of NBC collective schedules[7] using NBC test interfaces. Each **NBC_Request**[6] that is sent over RDMA is marked with a unique event tag that directly correspond to its **NBC_Handle**. This allows event descriptors to match upon probed data of the collective operation without invoking any external routines.

Figure 2 explains the overall design and the general description of what a LibPhotonNBC operation resemble. Each collective operation is delegated to a **NBC_Handle** to tracks its progress. **NBC_Handle** acquires progress

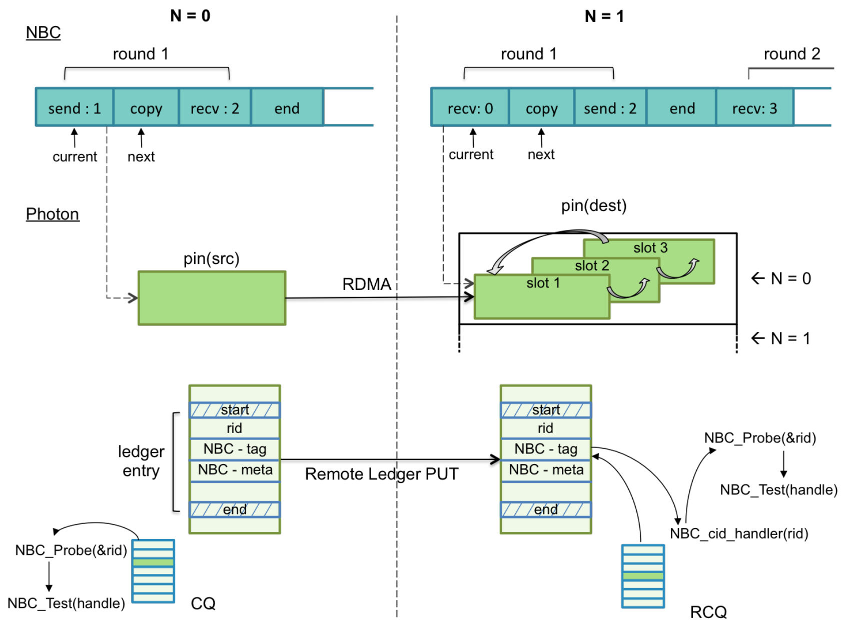

Figure 2: **Photon Collective Design Overview** - A NBC collective communication schedule drives progression of a collective operation through Photon **PWC** primitives. A per peer lookup table contains a pipeline of circular buffers used for collective RDMA operations. Each LibPhotonNBC event descriptor is handled by a special handler and ultimately ends up on a probe routine with the resepective **NBC_Handle**

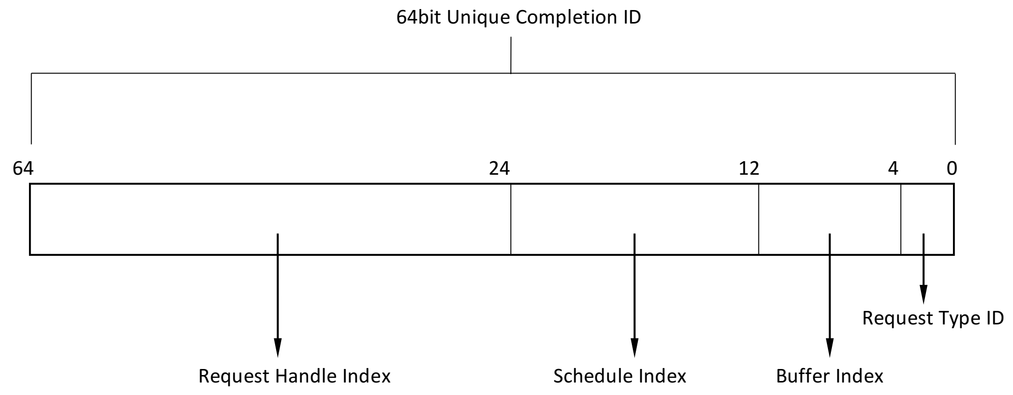

Figure 3: LibPhotonNBC 64bit Completion Identifier used for collective request tracking

updates using **NBC_Test()** interface. Once testing is completed (ie:, operation completed) a Photon completion request is pushed into the local PWC queue. Non blocking progression works by user checking the state of the collective request within this completion queue using the externally facing interface **photon_probe_completion**. User will be able to probe for this as local completion event for a collective operation.

As shown in the design diagram, NBC collective schedule may contain more than one iteration of **SEND/RECV** groupings. Therefore **NBC_Test()** will terminate only when all request iterations of a collective are tested for completion. For example, a collective communication schedule of a 4 node broadcast (binomial tree) tree in LibPhotonNBC works as follows – (each node separated by a comma, numerical value is the node number and each iteration/round is in brackets) **1**:[**send**$\Rightarrow$**2**] [**send**$\Rightarrow$**3**], **2**:[**recv**$\Leftarrow$**1**][**send**$\Rightarrow$**4**], **3**:[**recv**$\Leftarrow$**1**], **4**:[**recv**$\Leftarrow$**2**].

C. *Request Tracking*

Each iteration round of a **NBC_Schedule** is tracked using individual collective operation. Function routines in **Photon_PWC_Coll_Test_some()** of LibPhotonNBC takes the task progressing subset of requests. Whenever a *remote* completion event is encountered we match using a unique 64-bit completion event identifier (Figure 3) and if a match is successful we increment the completion count for a iteration and this step is repeated per event basis until all iterations are completed.

Completion identifiers assist request processing by encoding useful information about the collective schedule. As shown by Figure 3, first 4 bits encode the type of the request – which encodes request to be either a local *send* or a remote *receive*. For send or receive requests we check for its Photon local completion request type and then increment the respective completion count. Bits 4 - 11 indexes directly into the photon RDMA buffer table to copy data to target location, therefore avoids any buffer matching semantics. Bits 12 - 23 encodes schedule offset within a operation. In case the request belongs to a future iteration, we store the relevant buffer information in a temporary stash for future use. Finally we assign each request a unique handle id and encodes them in bits 24 to 63. We currently set the **PWC** default completion event size to 8 bytes (64bits). However this value is configurable - we could easily shrink or scale the completion identifier length[3] if we require further optimizations in future.

D. ***Flow Control***

RDMA message transfer semantics require some form of flow control to achieve synchronization of multiple operations. We use sender side credit tracking and special type of completion id (with no data) for this purpose. Each sender will manage a table of credits received from a peer and will only perform send operations to a particular destination if it has enough credits reserves. Number of total Photon **PWC** requests will increase linearly with the number of nodes used in a photon collective operation. Therefore a single RDMA buffer per destination can cause a bottleneck when either sender or receiver has to synchronize. For example a delay in request matching can propagate through the network causing significant performance degradation (ie;, stagnate some senders/receivers for a schedule on certain nodes) for the collective operation.

We employ standard request pipe-lining techniques to avoid this issue. The basic idea is to have indexed per peer circular RDMA buffers (cf. Figure 2 - as shown in **pin(dest)**) to increase the throughput of the system and also employ credit based flow control for synchronization using special credit requests. Therefore, a node will only stall for a Photon PWC operation

[3]There is a trade-off between amount of information we would like to encode in **PWC** identifiers and Photon **PWC** performance – After certain threshold, longer the identifiers slower the Photon **PWC PUT** operations will be.

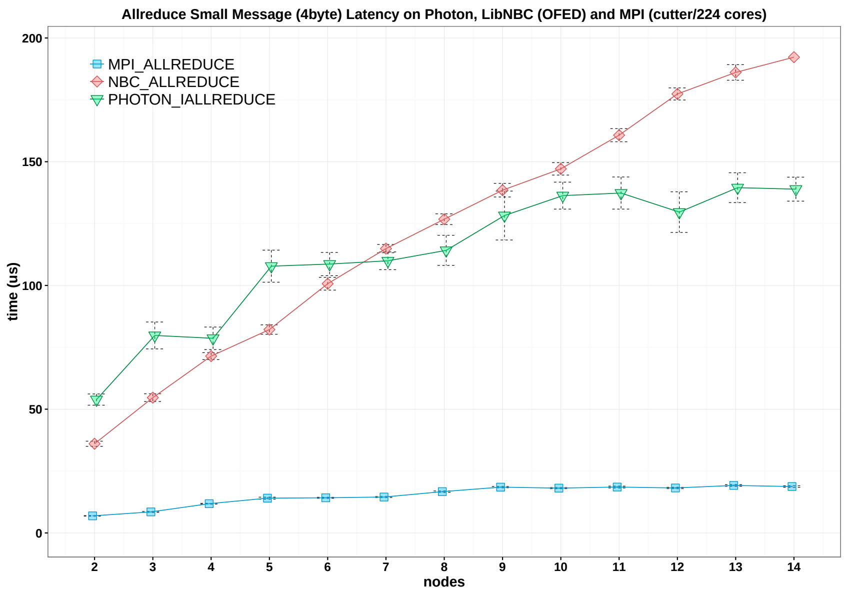

Figure 4: '**Allreduce**' scaling performance on cutter for small messages (4 bytes) with Photon, MPI and LibNBC

when peer credits reach some threshold value and thus all peer buffer slots are being exhausted (due to previous send/recv operations still on flight). We use 50 slots for default number of buffers in the pipeline.

3 Results

We conducted two (broadcast and allreduce) micro-benchmark experiments with LibPhotonNBC Library. All experiments were carried out thus far are, of a small scale and, was on a private HPC cluster called 'Cutter' located at Indiana University. At this scale we carried out experiments on up to 14 nodes/224 cores on Intel Xeon E5 16 core 2.1GHz processors supported by an Infiniband/Mellanox Network. We tested scaling characteristics of the aforementioned collective operations with different implementations available including LibPhotonNBC collectives, non-blocking (MPI 3.0) version of OpenMPI (default IB btl network) collectives and the LibNBC (using default network transport) version 1.1.0.

Figures 4, 6 report latency for 'Allreduce' and 'Broadcast' operations for

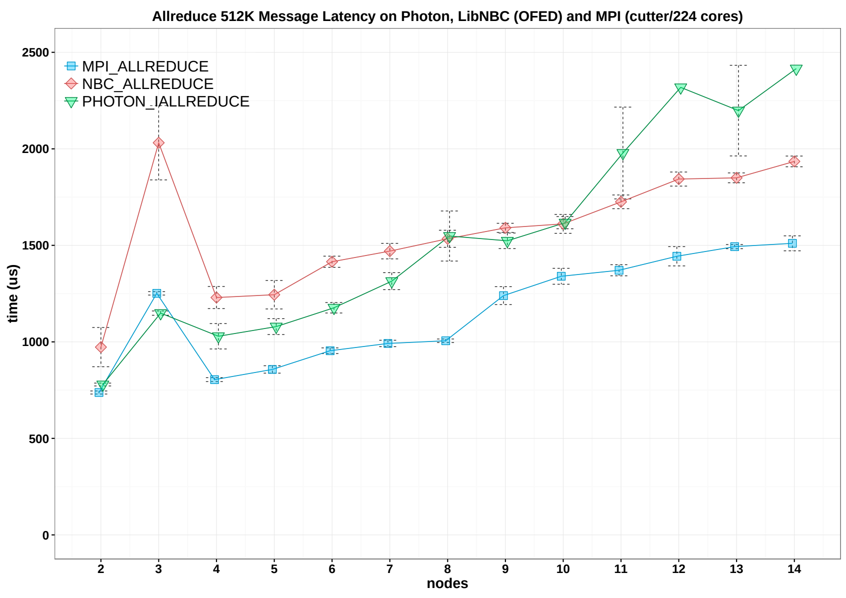

Figure 5: ‘**Allreduce**‘ scaling performance on cutter for Large messages (512K bytes) with Photon, MPI and LibNBC

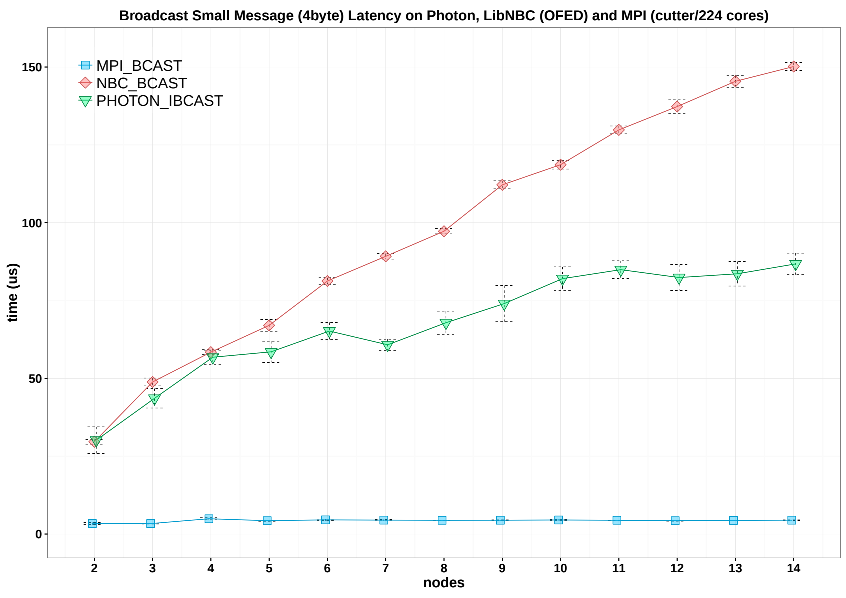

Figure 6: ‘**Broadcast**‘ scaling performance on cutter for small messages (4 bytes) with Photon, MPI and LibNBC

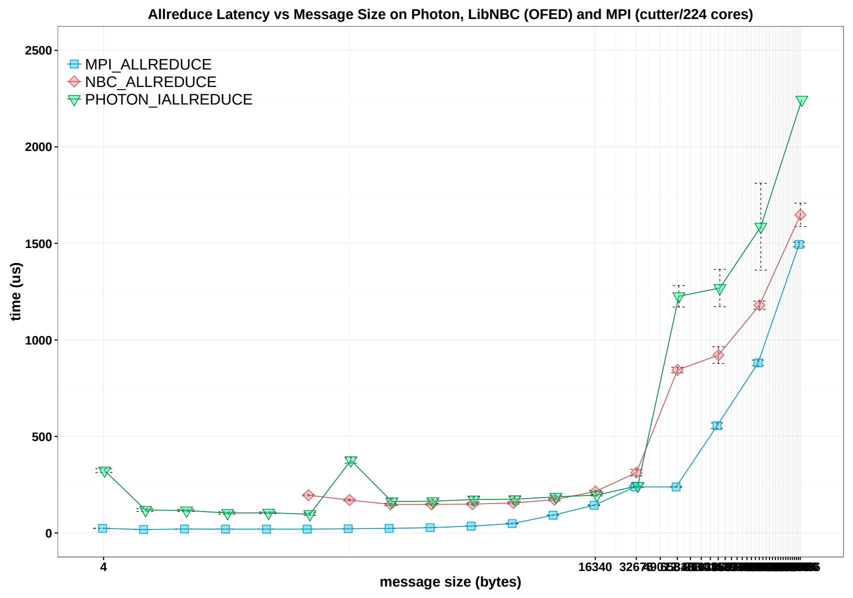

Figure 7: ‘**Allreduce**‘ performance by scaling message size from 4 bytes to 8 MBytes on cutter (14 nodes) with Photon, MPI and LibNBC

4 byte messages. We observe that, maximum slowdown of LibPhotonNBC collectives is about 15X to 20X w.r.t non blocking MPI Allreduce and Broadcast. However LibPhotonNBC ‘Allreduce‘ report better latency values for 512K messages (Figure 5) upto 14 nodes with average slowdown being around 1.5X w.r.t MPI. We also report performance results by scaling message size on ‘Allreduce‘ operation (Figure 7) on 14 nodes – LibPhotonNBC ‘Allreduce‘ operation running time follows closely with MPI and NBC, yet slower than both implementations.

We understand that LibPhotonNBC is currently not the most optimized version for small messages. However a better performance is visible for larger message sizes. This attributes largely LibPhotonNBC’s latency hiding optimizations, which better utilize the network bandwidth. Our profiling data shows that significant amount of time is spent in **NBC_Test()** routines and related message matching routines. It is therefore our belief that LibPhotonNBC can be improved further for better results.

4 Conclusion

This paper discussed our attempt to implement a non blocking collective library called LibPhotonNBC with Photon RDMA Library and our preliminary results on observing it in action. We used Photon PWC network operations coupled with completion events to assist communication schedules in LibPhotonNBC. We used LibNBC to integrate communication schedules and well known collective algorithms into Photon. Our initial benchmark results show some promise in collectives involving larger message transfers. We believe that there is significant potential in LibPhotonNBC to be a efficient and scalable collective Library for HPC applications and frameworks. Reducing message overheads in critical paths and exploiting Photon completion events for message tagging and efficient lookups may further enhance the performance of the LibPhotonNBC library.

References

[1] M. Barnett, R. Littlefield, D. G. Payne, and R. Van de Geijn. Global combine on mesh architectures with wormhole routing. In *Parallel Processing Symposium, 1993., Proceedings of Seventh International*, pages 156–162. IEEE, 1993.

[2] M. Barnett, L. Shuler, R. van De Geijn, S. Gupta, D. G. Payne, and J. Watts. Interprocessor collective communication library (intercom). In *Scalable High-Performance Computing Conference, 1994., Proceedings of the*, pages 357–364. IEEE, 1994.

[3] S. H. Bokhari and H. Berryman. Complete exchange on a circuit switched mesh. In *Scalable High Performance Computing Conference, 1992. SHPCC-92, Proceedings.*, pages 300–306. IEEE, 1992.

[4] E. W. Chan, M. F. Heimlich, A. Purkayastha, and R. A. Van de Geijn. On optimizing collective communication. In *Cluster Computing, 2004 IEEE International Conference on*, pages 145–155. IEEE, 2004.

[5] T. Hoefler, P. Kambadur, R. L. Graham, G. Shipman, and A. Lumsdaine. A case for standard non-blocking collective operations. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 125–134. Springer, 2007.

[6] T. Hoefler and A. Lumsdaine. Design, implementation, and usage of libnbc. *Open Systems Lab, Indiana University, Tech. Rep*, 8, 2006.

[7] T. Hoefler, A. Lumsdaine, and W. Rehm. Implementation and performance analysis of non-blocking collective operations for mpi. In *Supercomputing, 2007. SC'07. Proceedings of the 2007 ACM/IEEE Conference on*, pages 1–10. IEEE, 2007.

[8] N. T. Karonis, B. R. De Supinski, I. Foster, W. Gropp, E. Lusk, and J. Bresnahan. Exploiting hierarchy in parallel computer networks to optimize collective operation performance. In *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*, pages 377–384. IEEE, 2000.

[9] T. Kielmann, R. F. Hofman, H. E. Bal, A. Plaat, and R. A. Bhoedjang. Magpie: Mpi's collective communication operations for clustered wide area systems. *ACM Sigplan Notices*, 34(8):131–140, 1999.

[10] E. Kissel and M. Swany. Photon: Remote memory accessmiddleware for high-performance runtime systems. *First Annual Workshop on Emerging Parallel and Distributed Runtime Systems and Middleware, IPDRM*, 1, 2016.

[11] P. Mitra, D. Payne, L. Shuler, R. van de Geijn, and J. Watts. Fast collective communication libraries, please. In *Proceedings of the Intel Supercomputing Users' Group Meeting*, volume 1995, 1995.

[12] D. S. Scott. Efficient all-to-all communication patterns in hypercube and mesh topologies. In *Distributed Memory Computing Conference, 1991. Proceedings., The Sixth*, pages 398–403. IEEE, 1991.

[13] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of collective communication operations in mpich. *International Journal of High Performance Computing Applications*, 19(1):49–66, 2005.

[14] H. Zhu, D. Goodell, W. Gropp, and R. Thakur. *Hierarchical collectives in mpich2*. Springer, 2009.