Optimizing non-blocking Collective Operations for InfiniBand

Torsten Hoefler
Open Systems Lab
Indiana University
Bloomington IN 47405
Email: htor@cs.indiana.edu

Andrew Lumsdaine
Open Systems Lab
Indiana University
Bloomington IN 47405
Email: lums@cs.indiana.edu

***Abstract*—Non-blocking collective operations have recently been shown to be a promising complementary approach for overlapping communication and computation in parallel applications. However, in order to maximize the performance and usability of these operations it is important that they progress concurrently with the application without introducing CPU overhead and without requiring explicit user intervention. While studying non-blocking collective operations in the context of our portable library (libNBC), we found that most MPI implementations do not sufficienctly support overlap over the InfiniBand network. To address this issue, we developed a low-level communication layer for libNBC based on the Open Fabrics InfiniBand verbs API. With this layer we are able to achieve high degrees of overlap without the need to explicitly progress the communication operations. We show that the communication overhead of parallel application kernels can be reduced up to 92% while not requiring user intervention to make progress.**

I. Introduction

Non-blocking collective operations have recently received attention as a promising new class of communication operations [1]. Their benefits include better utilization of current architectures due to communication/computation overlap [2] and the mitigation of the negative effects caused by the pseudo-synchronizing behavior [3] of collective operations. Non-blocking collectives support the combination of common communication optimization mechanisms such as early binding [4] and higher-level expressiveness of communication operations [5].

Despite their many benefits, these operations are not available in the current Message Passing Interface (MPI) standards [6], [7]. Several publications describe mechanisms that are comparable to non-blocking collectives (NBC) [8], [9], [10], but LibNBC [11] is the only freely available and portable implementation of all MPI collective algorithms. An MPI-optimized version of LibNBC for fully connected networks has been presented in [12]. It was shown that LibNBC adds only a negligible overhead to the communication path and enables high degrees of overlap. The portable version of LibNBC is built on top of non-blocking MPI point-to-point operations. As a result, the overhead and the achievable overlap are directly dependent on the overhead and (independent) progress of MPI_Isend and MPI_Irecv in the underlying MPI library.

Several benchmarking studies about overlap [13], [14], [15] show controversial results for different MPI libraries. However, the studies do mainly agree that independent progress is limited in current open-source MPI libraries. As we will discuss below, our own experience was similar: the overlap potential of current open-source MPI libraries is limited.

A. *LibNBC Architecture*

The current version of LibNBC is a software package that offers non-blocking variants of all MPI collective operations. LibNBC's central element is a collective schedule that contains all the information to execute a collective operation. A schedule is usually different for every combination of rank and argument. The schedule is generated in the immediate function (e.g., NBC_IBcast) and attached to the returned NBC_Handle (cf. MPI_Request). A schedule may consist of multiple logical rounds, where the next round can only be started if all (potentially non-blocking) operations of the current round finished, i.e., all operations which are in the same round are executed simultaneously and must be independent. Further details about the schedule layout and possible elements are discussed in [12] and [16].

The execution of a schedule is performed by the scheduler in LibNBC. The scheduler is responsible to start new communication rounds and to keep track of all open requests. The scheduler is called after the creation of a new schedule (in the immediate function, e.g., NBC_IBcast) to start the first round and every time when NBC_Test is called. The scheduler tests all outstanding communication requests for completion and starts the next round if all are completed. It returns a special flag (NBC_OK) to the user if the last round (and therewith the operation) was finished.

A special blocking version of the scheduler is called by NBC_Wait. The main difference between blocking and non-blocking scheduler execution is that all calls to the underlying communication system are blocking (MPI_Wait instead of MPI_Test) and that the scheduler loops until the whole schedule is executed.

However, the overlap potential is limited by this design, mainly by two factors [12]. First, the user has to call NBC_Test periodically to advance the internal state of the scheduler (start new communication rounds) and second, the overlap is limited by the underlying MPI library.

The first problem could be solved if the scheduler was executed in a separate progress thread. However, this would require a thread-safe (MPI_THREAD_MULTIPLE) MPI library and thus limit portability significantly. The second problem, the underlying communication system, causes a higher performance loss. This could be overcome if another transport layer would be used to send and receive messages asynchronously. It would be beneficial if this transport layer ensures asynchronous progress. However, the MPI standard does not define a clear progress rule and the support for asynchronous progress is limited.

The first step in optimizing LibNBC for a particular network like InfiniBand would be to use the low-level network interface in a way that enables highest overlap, full asynchronous progress and is optimized for the needs of LibNBC. Not all features of MPI are needed by LibNBC, for example, the MPI library needs to implement a rather complicated protocol to support MPI_ANY_SOURCE which is not needed by LibNBC. The requirements of LibNBC are listed in the following.

- **non-blocking send** is used to start a send operation and should return immediately (low overhead)
- **non-blocking receive** is used to post a receive and should return immediately (low overhead)
- **request objects** are needed to identify outstanding communications. All request objects have to be relocatable!
- **communication contexts aka communicators** are used as a communication universe to represent MPI communicators passed by the user
- **message tags** are used to differentiate between multiple different outstanding collective operations on a single communicator
- **message ordering** message with the same tag must match in the receiver side in the order they were issued on the sender side
- **test for completion** this test should be non-blocking and specific to a request object. It might be used to progress the communication. However, fully asynchronous progress is preferable.
- **wait for completion** is optional (can be a busy test), but might be used for different optimizations (e.g., lower power consumption by using blocking OS calls)

Having defined the requirements of LibNBC, we will briefly describe the InfiniBand network in the next section.

B. The InfiniBand Network

InfiniBand^{TM} [17] is a widely used cluster interconnect that supports many different options to transmit messages. We analyzed the performance of different transport functions and types in [18]. This analysis limits our choice to RDMA-Write over Reliable Connection and Send/Receive of Unreliable Datagrams. Different works have shown that the Unreliable Datagram transport can be beneficial at large scale [19], [20], [21], however the complexity of the implementation that has to ensure reliability in software seemed not feasible for our first implementation and we chose RDMA-Write.

To use the Reliable Connection transport type, the communicating nodes need to be connected via Queue Pairs (QP), consisting of a Send Queue (SQ) and a Receive Queue (RQ) that form a communication channel. The user can post Send Requests (SR) or Receive Requests (RR) to the queues which then operate fully asynchronously. SRs and RRs, are processed asynchronously and in order by the Host Channel Adapter (HCA) and when a SR finishes or fails, a completion entry is generated in the associated Completion Queue. All sent or received memory has to be registered with a call to the InfiniBand^{TM} library. This call usually performs operating system tasks (i.e., pinning of memory and adjusting translation tables) [22] and is thus expected to be slow. A usual way to reduce the impact of memory registration is lazy deregistration combined with a registration cache [23], [24] to re-use existing registrations if possible.

An MPI optimized version of LibNBC exists and different MPI implementations could be used to match it to InfiniBand. The overlap potential of an exemplary MPI implementation with LibNBC will be analyzed in the following Section.

II. MPI IMPLEMENTATIONS FOR IB

Two popular MPI implementations for InfiniBand, Open MPI and MVAPICH, exhibit similar performance characteristics. Neither implementation offers asynchronous progress (a progress thread) of outstanding messages. The two-sided semantics of MPI force the implementer to implement a protocol where the sender has to wait until the receiver posted the receive request because the message size is not limited (this prevents the usage of pre-posted receive buffers). This protocol is commonly called “rendezvous protocol”. Another MPI feature, the MPI_ANY_SOURCE semantics, force the implementation to perform at least three message exchanges for every large message [25].

All benchmarks are conducted on the “odin” cluster at Indiana University. Odin consists of 128 nodes with dual cpu dual core 2.0Ghz Opteron 270 HE processors connected with Mellanox Technologies MT23108 InfiniHost adapters to a 288 port InfiniBand switch. Due to space restrictions and implementation difficulties with MVAPICH[1], we decided to analyze Open MPI in detail. Since the investigated MPI libraries don’t have (fully) asynchronous progress for large messages, the user-program has to progress the requests manually. The only way to do this in a fully portable way is to test every outstanding request for completion because the MPI standard mandates that repeated calls MPI_Test must complete a request eventually. However, calling MPI_Test during the computation is not only a software-technological nightmare (passing the requests down to the computation kernels) but is also a source of two kinds of significant overhead. The first source of overhead is simply the time spent in MPI_Test itself. The second overhead source is less obvious but more influential. Calls to libraries (e.g., BLAS [26]) must be split

[1]’MPIDI_CH3I_RMDA_init(95): Error initializing MVAPICH2 malloc library’

up into smaller portions which, first, destroys code structure, and, second, might lower efficiency (i.e., cache efficiency, the cache is also polluted by the calls to MPI_Test). Thus, calling test is not a feasible option. However, not calling any test with the MPI implementations results simply in no overlap at all (see analyses in the following section).

Thus, the user of the current LibNBC is forced to perform test calls in his code. Accepting this, the user faces another problem because the decision when and how often test should be called is non-trivial. Too many calls cause unnecessary overhead and not enough calls will not progress the library and causes unnecessary waiting. It is easy to show that the optimal "test-patterns" depend on the protocol used by the MPI library and therewith on the library itself. Given this complexity that an application programmer faces today, he usually just applies a simple heuristic of calling test when it is convenient or not at all. However, overlap performance in this case is clearly suboptimal.

We will analyze different test strategies in the following section with the goal of deriving better heuristics.

A. Open MPI Message Progression for LibNBC

We use LibNBC to analyze the progression strategies. LibNBC's scheduler calls MPI_Testall on all outstanding requests related to the NBC_Handle that NBC_Test is called with. Thus, the test behavior is transparent. We extended our benchmark NBCBench which was first introduced in [12] to support different test strategies. NBCBench follows the principles for collective benchmarking described in [27], [28], [29] to ensure highly accurate results. The benchmark is run twice for every combination of message size and communicator size. The first run determines the time that the (blocking) execution takes and the second run runs a computation of the length of the first run between init and wait of the collective communication. The implemented test strategy issues N tests in equidistant times during the simulated computation. N is a function of the message size and therewith indirectly of the transfer time, it is computed as

$$N = \left\lfloor \frac{size}{interval} \right\rfloor + 1$$

For example, if the datasize is 4096 bytes and the interval is 2048 bytes, the benchmark issues one test at the beginning, one after 50% of the computation and one at the end. The test-interval is chosen by the user.

We chose two collectives that are not influenced by the missing asynchronous progress of LibNBC itself, but represent a common subset. The first operation, NBC_Igather, represents a many-to-one operation while the second operation, NBC_Ialltoall represents the group of many-to-many collectives.

We benchmarked different test-intervals (0 for no tests, 1024, 2048, 4096 and 8192) for NBC_Igather and NBC_Ialltoall for 16 and 64 nodes. We analyzed two different memory registration modes of Open MPI, leave pinned (where the memory is cached in a registration cache) and no leave pinned (the memory is registered in a pipelined way to overlap registration and communication) [25].

Figure 1 and 2 show the results of the overhead benchmark with Open MPI 1.2.4/openib on 16 and 64 nodes respectively. The overhead is defined as the time that is spent for

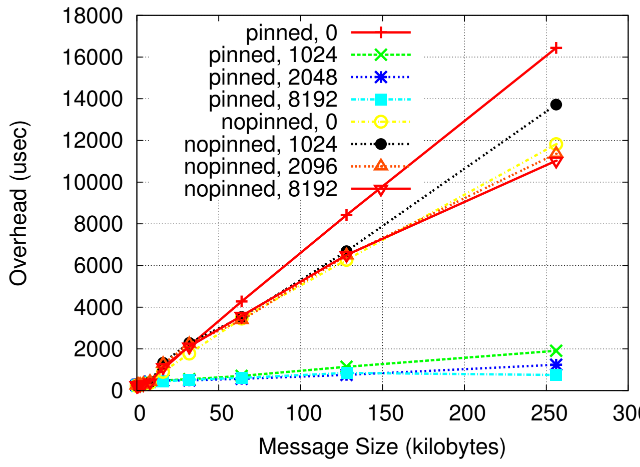

Fig. 1. Ialltoall 16 Nodes

communication. This is **not** the latency, but the sum of the time spent in initialization (e.g., the call to NBC_Igather call), testing (NBC_Test) and the waiting time at the end (NBC_Wait). Thus, the benchmark models the ideal overlap if all communication can be overlapped. The cases where no

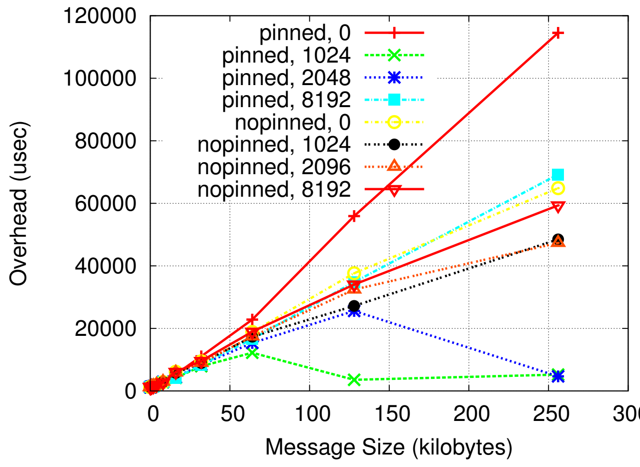

Fig. 2. Ialltoall 64 Nodes

tests were performed were equal to the blocking execution of MPI_Alltoall in this scenario (no overlap at all). The optimal test intervals differ between 16 and 64 nodes. While, on 64 nodes, testing every 1024 bytes seems most beneficial, a test-interval of 8192 bytes performs better on 16 nodes.

The results for NBC_Igather on 64 nodes, shown in Figure 3, show also different optimal test-intervals. It is even more complicated because a test-interval of 2048 bytes seems better for several message sizes. Thus, we can conclude that the optimal test strategy does not only depend on the MPI implementation, but also on the message size and communi-

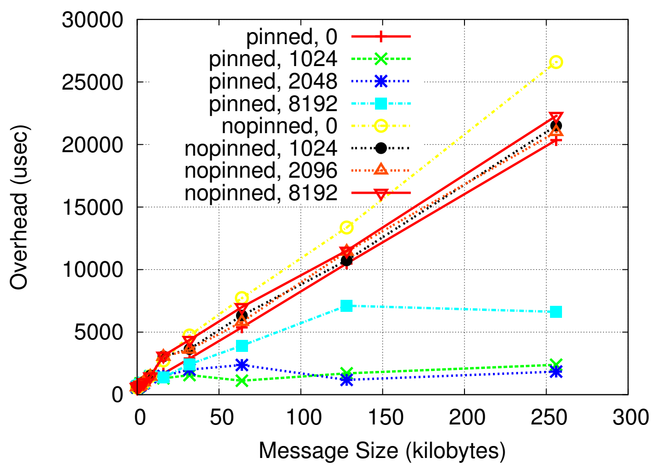

Fig. 3. Igather on 64 Nodes

cator size. A simple heuristic does not seem feasible. However, testing itself is suboptimal for several reasons, thus we should focus on different strategies.

This section has shown that even if the application programmer is willing to progress the MPI manually, the selection of an optimal strategy is highly non-trivial (and not very likely to be performed by a developer). Thus, we decided to implement a library that offers all necessary features to LibNBC directly on top of InfiniBand$^{\text{TM}}$ (the OFED verbs interface). The design and implementation of this library is described in the next section.

III. Implementation

Many MPI implementations are tuned for microbenchmark performance of the blocking Send/Recv operations. Thus, things like minimal CPU overhead are often neglected to achieve higher blocking performance. Only some implementations, like [30], optimize for overlap and often need to be enabled explicitely. In the design of LibOF, we did not focus on microbenchmark performance but we tried to minimize the CPU overhead (and thus maximal overlap). Thus, our blocking microbenchmark results are expected to be slightly worse than Open MPI's. Measurements with blocking communication showed that our microbenchmark latency and bandwidth lie within a 5% margin of Open MPI's. Plots of those results can just be omitted.

Additionally to the low overhead of the calls themselves, we also have to ensure that there is no need to call them often. To achieve this, we have to design protocols that enable the maximum asynchronism between the calling program and the InfiniBand$^{\text{TM}}$ network which offers asynchronous progress.

The library use MPI communicators as communication context. It attaches it's data as an attribute to every communicator. This communicator-specific structure stores all peer-specific information for this communicator and is initialized at the first use of this communicator. Due to the InfiniBand$^{\text{TM}}$ connection establishment mechanisms, the first call with every communicator is blocking and needs to be performed by all ranks in the communicator to avoid deadlocks.

Orienting on current MPI implementations, we decided to implement an Eager and Rendezvous protocol in order to achieve the blocking performance. Those protocols will be described in the following.

A. Eager Protocol

Our Eager protocol is designed to proceed completely asynchronously of the calling program. Every peer has a number of buffers that can store the eager message, its size the tag and some protocol information. Those peer-specific buffers are registered during communicator initialization and the necessary data (`r_key, address`) is exchanged. When a new send operation is initiated with `OF_Isend`, all necessary data is attached to the request, which is set to the status `EAGER_SEND_INIT`, and the function returns to the user. The first test call with this request copies the data into a pre-registered send-buffer (if available) and posts a signaled `RDMA_WRITE` send request to the peer's SQ. The send-buffer is a linear array in memory and a tag of -1 marks an entry as unused. To find an unused buffer, the array is scanned for a tag equal to -1 and the buffer-index is attached to the request. The WR id is set to the address of the request so that the buffer can be freed (tag set to -1) when the WR completes on the send side. `OF_Irecv` attaches the arguments to the request and sets the request's status to `RECV_WAITING_EAGER`. Every test on a request with this status scans the eager array for the tag. It copies the data in the receive buffer if the tag is found and notifies the sender that the receive buffer can be re-used. The notifications (`EAGER_ACKs`) are piggybacked (in the protocol information) to other eager messages or sent explicitly if more than a certain number of eager buffers are used.

B. Rendezvous Protocol

Our rendezvous protocol differs from the protocol used in any MPI implementation because LibNBC does not require a receive from any source. Thus, we can drive a receiver-based protocol where the receiver initiates the communication and the sender is passive until it is triggered. The receiver attaches all necessary information to the request and sets it to `RNDV_RECV_INIT` during `OF_Irecv` function. The first test on this request registers the receive buffer, packs `tag`, `r_key` and `address` into a preregistered RTR message buffer. This buffer is then sent with `RDMA_WRITE` to a preregistered location at the sender and the request is set to `RECV_SENDING_RTR`. The RTR send buffer is freed with a similar mechanism as the eager send buffer. `OF_Isend` sets the request's status to `SEND_WAITING_RTR` after attaching the arguments to the request and registering the send memory. A test on the sender-side scans the RTR array for the request's tag. If the tag is found, it posts the `RDMA_WRITE_WITH_IMM` send request to its local SQ and sets the request status to `SEND_SENDING_DATA`. A receive request is finished when the receiver received the data.

C. Optimizing for Overlap

Optimizing for overlap means minimizing the CPU overhead and maximizing the asynchronous InfiniBand$^{\text{TM}}$

progress. We minimize the CPU overhead by using our optimized protocols that only use a minimal number of operations to send and receive messages. For example, we use only a single CQ for send and receive requests because polling a CQ is relatively expensive [18].

Achieving the maximum asynchronism is easy in the case of the eager protocol and tricky in the case of rendezvous. A first simple optimization, called test-on-init in the following, is to call the first test in the send of the eager protocol and the receive in the rendezvous protocol during the `OF_Isend` and `OF_Irecv` functions respectively. This hands the (ready) message immediately to the InfiniBand^{TM} network and does not introduce unnecessary waiting until the first test is called by the user. However, it obviously increases the CPU overhead in those functions.

The test-on-init optimization makes the progress in the eager protocol completely asynchronous (no test is necessary to "push" messages). However, the rendezvous protocol does still need a test on the sender-side to send the message after the RTR arrived. Thus, no progress will happen before the test. The optimal time between the `OF_Isend` and the first test is also not trivially determinable (it would be a single latency if recv and send were started at the same global time). A simple approach would be to poll test until the RTR message has arrived, but this might introduce deadlocks because `OF_Isend` would depend on the receiver. We decided to implement a timeout-based mechanism that polls only a limited time to avoid deadlocks and will refer to it later as "wait-on-send". However, this mechanism increases the CPU overhead of the rendezvous send drastically. We will discuss techniques to mitigate this after we analyzed and compared the influence in the next section.

IV. Performance Results

We compare the overhead and overlap of our implementation to Open MPI's overhead with different techniques. We used two different microbenchmarking tools, Netgauge and NBCBench to assess raw performance. Real-world results are shown with two application kernel benchmarks, parallel compression and three-dimensions Fast Fourier Transform (3d-FFT).

A. Netgauge

Netgauge [31] is a tool to benchmark different network characteristics. Its key features are wide support for different networks (e.g., MPI) and communication patterns (e.g., LogGP [32]), an extremely easy interface to implement new patterns and/or network protocols and the portable high precision timing interface. We extended Netgauge with a module to use LibOF as communication channel and added a new communication pattern which assesses the overheads of non-blocking communication. The module's implementation is trivial and just maps Netgauge's (blocking and non-blocking) send/recv and test functions to `OF_Isend`, `OF_Irecv` and `OF_Test`. The communication pattern "nbov" does a simple ping-pong and measures the times to issue the non-blocking send or receive calls and loops on test until the operation succeeds. Additionally, it takes the time for all calls to test and divides them by the number of tests issued to get a rough estimation for the average time for the test operation. We use the Netgauge's high-precision timers (RDTSC [33]) to benchmark single messages and repeat the ping-pong procedure multiple times (1000) and average the results afterwards.

We ran our new pattern with Open MPI and LibOF to determine the overheads. The Isend overhead is shown in Figure 4. We set the eager protocol limit to 255 bytes for LibOF and

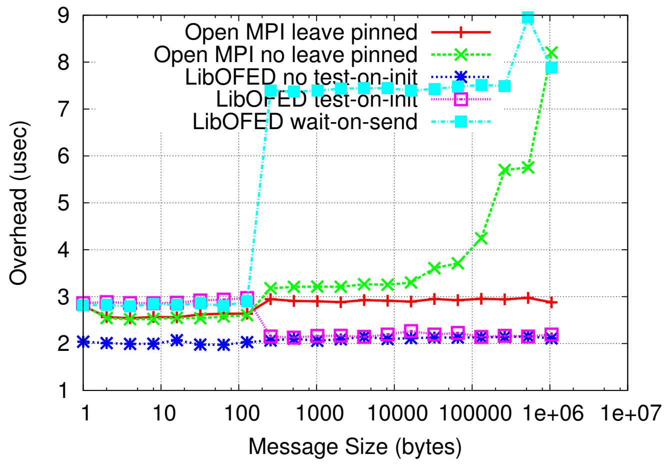

Fig. 4. Isend Overhead for Open MPI and LibOF

Open MPI. LibOF without test-on-init performs best because it does not start any operation during the Isend. The wait-on-send adds a huge overhead to every send operation as expected.

The Irecv overheads are shown in Figure 5. The wait-on-send and test-on-init show the same performance because wait-on-send implies test-on-init and the receiver side is not different. The overhead of registering the memory and sending the RTR message can be seen when the protocol is switched to rendezvous. It is still not clear if the minimization of the send/recv overhead is more important then the minimization of the test overhead.

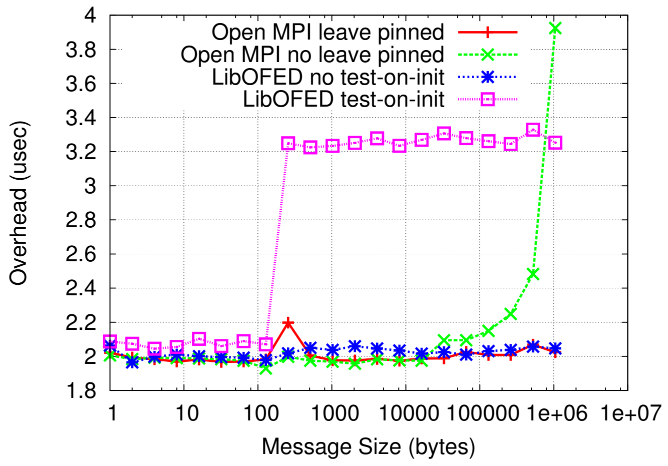

Fig. 5. Irecv Overhead

The test overheads are omitted due to space restrictions and lie in a range from 0.1 (rendezvous) to 2 (eager) microseconds.

B. Optimizing Wait-On-Send

We saw in the previous Section that wait-on-send adds a huge constant CPU overhead per message. LibNBC usually issues many messages at the same time (dependent on communicator size) so that this overhead adds up per message. To mitigate this effect and since we have transparent access to our implementation, we implemented a hook `OF_Startall` in LibOF that progresses multiple send requests until they leave the status `SEND_WAITING_RTR`. Thus, the overhead (which is basically the waiting time for the RTR to be transmitted) is only paid once for multiple messages. The `OF_Startall` is called by LibNBC directly after a new round is started and has also a timeout mechanism to prevent deadlocks.

C. NBCBench

We use NBCBench again to compare our implementation with the best results (with the "optimal" test interval) achieved with Open MPI (cf. Section II-A). We ran the same test-intervals as previously used with and without test-on-init. We ran the wait-on-send implementation only without tests because it is designed to run asynchronously and test would only add overhead.

The results for NBC_Ialltoall on 16 and 64 nodes are shown in Figure 6 and 7 respectively. The results indicate that our

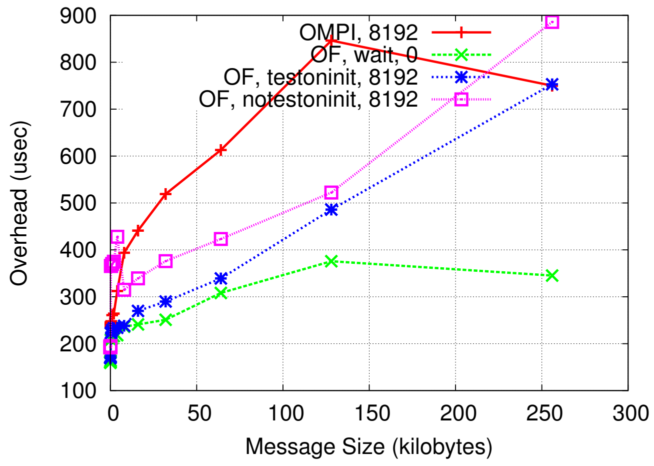

Fig. 6. Ialltoall overheads on 16 nodes

wait-on-send implementation (the optimized version) performs best in nearly all cases. The 64 node case where test-on-init with tests every 8192 bytes performs better lies in the small message range where blocking collective operations are faster (the overhead of generating the schedule is significant for small messages, cf. [12]).

Similar results can be found in Figures 8 and 9 which shown the comparison for NBC_Igather on 16 and 64 nodes respectively.

All the benchmarks and results in this paper have been presented for LibNBC's non-blocking collectives. Figure 10 compares the Performance of NBC_Ialltoall (A2A) and

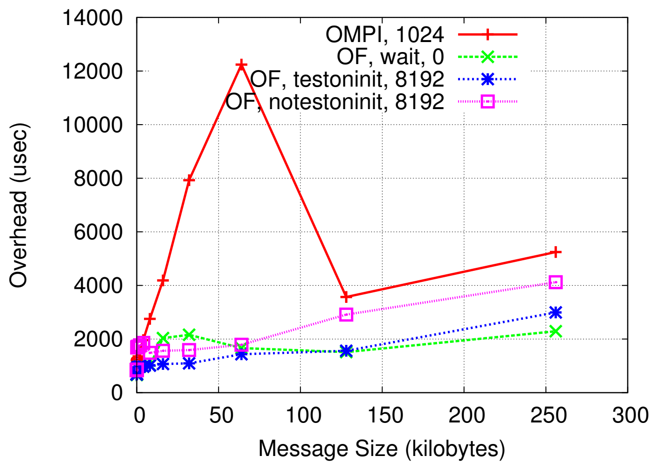

Fig. 7. Ialltoall overheads on 64 nodes

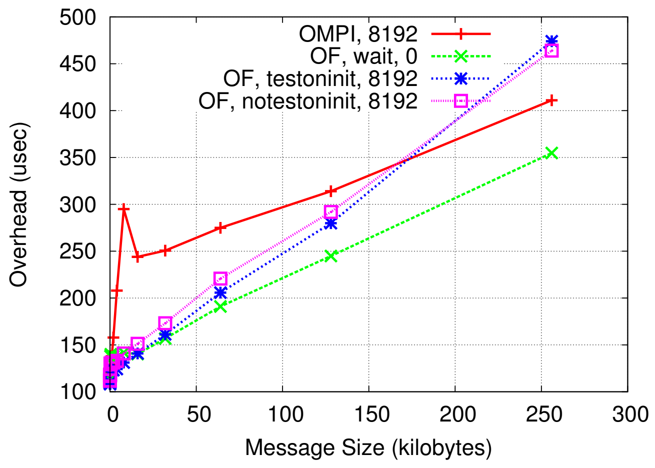

Fig. 8. Igather overheads on 16 nodes

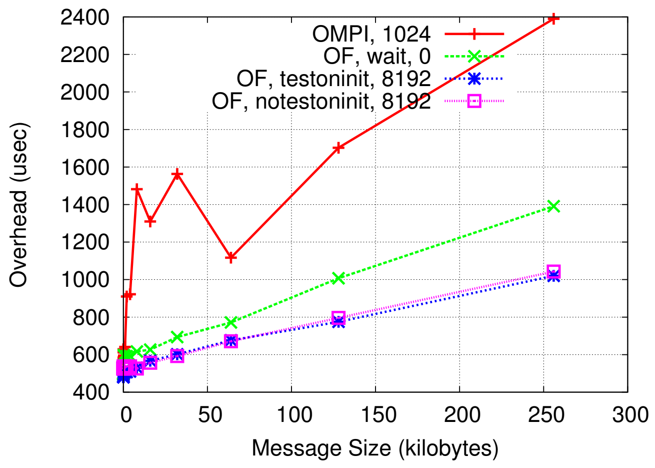

Fig. 9. Igather overheads on 64 nodes

NBC_Igather (GAT) to the highly optimized blocking MPI implementations [34].

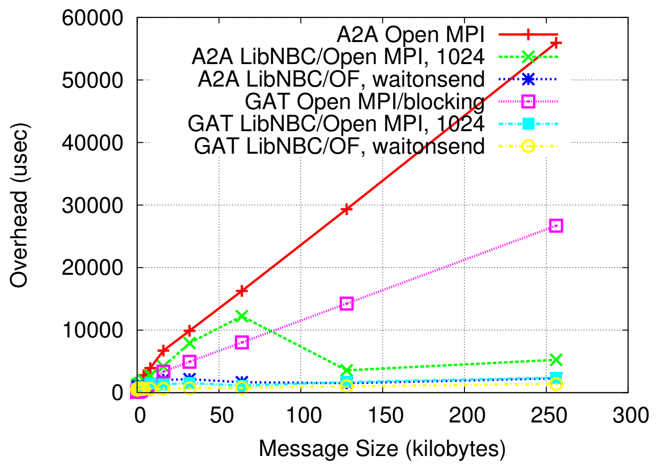

Fig. 10. Comparison of the Alltoall (A2A) and Gather (GAT) overheads between (non-blocking) LibNBC and (blocking) Open MPI on 64 nodes

D. Parallel Compression

Our parallel compression benchmark has first been introduced in [12] to analyze the influence of non-blocking collective operations to real application benchmarks. The compression benchmark represents any scientific application where data is processed (in our case compressed) in a distributed way and gathered to a single node at the end (e.g., to write it to mass storage). The final gathering is tricky because before the NBC_Igatherv or MPI_Gatherv can be issued, a MPI_Gather has to be preformed to know the datasize of every rank. The first Gather is non-overlappable. Figure 11 shows the communication overhead for the compression of

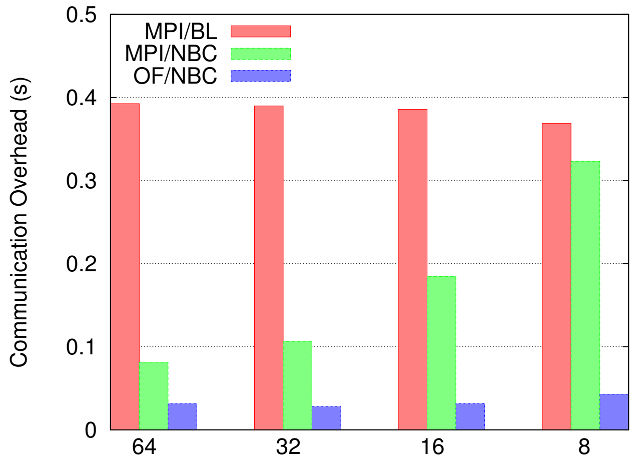

Fig. 11. Parallel compression communication overhead on 64 down to 4 nodes.

122MB on different node counts with blocking MPI calls, non-blocking calls with the standard LibNBC (based on MPI_Isend/Irecv calls performing NBC_Test to progress) and our optimized LibNBC/OF without tests using the wait-on-send implementation.

E. Parallel 3d-FFT

The parallel 3-dimensional FFT has also been introduced in [12] as an application-kernel benchmark to analyze non-blocking collective operations. We show results of a transformation of a 640^3 system on 64, 32 and 16 nodes and due to memory limitations, a 320^3 system on 8, 4 and 2 nodes running one process per node in Figure 12. The results show

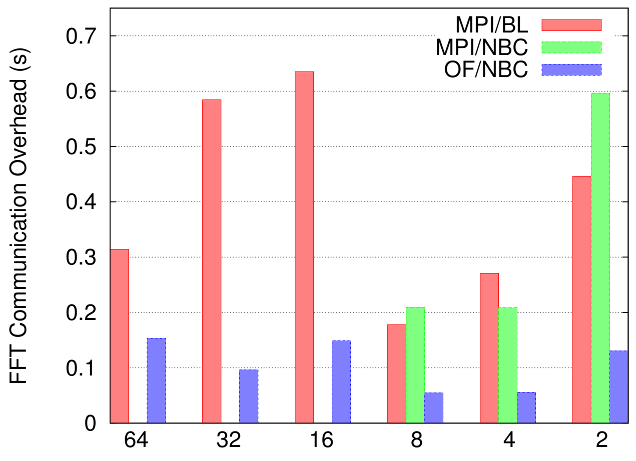

Fig. 12. 3d-FFT communication overhead on 64 down to 2 nodes.

clearly that the communication overhead and therewith the computation time is reduced significantly. The 640^3 system could not be transformed with LibNBC on top of Open MPI.[2]

V. Conclusions and Future Work

In this work, we analyzed the performance of LibNBC on InfiniBand^{TM} in detail and investigated several options to minimize the communication overhead (in order to maximize communication overlap). We showed that reasonable performance can be achieved at the user level using MPI and appropriate invocation patterns of MPI_Test (to guarantee progress of MPI). However, the invocation patterns depend not only on the MPI implementation but also on the communicator size and data size. We showed that, in general, the programmer would not able to derive simple and optimal heuristics for progressing MPI. Furthermore, having to manually progress MPI in this way is generally suboptimal (code structure and overheads). Based on the requirements of LibNBC, we implemented a low-level interface which uses the OFED verbs directly to communicate. We propose a new rendezvous protocol that does not require user-level intervention to make independent progress in the network. Furthermore, we show with several microbenchmarks and application kernels that our implementation performs significantly better than blocking communication and non-blocking communication based on Open MPI.

Future work includes the analysis of using threads for the asynchronous progress. We avoided threads in this work to

[2]it crashed with an InfiniBand^{TM} “RETRY EXCEEDED ERROR” in all runs (we suspect that the huge number of MPI_Isend/Irecv operations caused this problem)

retain portability to systems that do not support threads like the newest Cray XT or Blue Gene/L machines. Furthermore, we will implement an abstract interface in LibNBC to enable easy network-specific extensions. We will also analyze the implementation of InfiniBand-optimized collectives (e.g., multicast-based broadcast [35] or RDMA-based barrier [36]) in our library.

ACKNOWLEDGMENT

Thanks to Gleb Natapov (Voltaire), Christian Siebert (NEC) and the anonymous reviewers for helpful comments. This work was supported by a grant from the Lilly Endowment, National Science Foundation grant EIA-0202048 and a gift the Silicon Valley Community Foundation on behalf of the Cisco Collaborative Research Initiative and a grant by the Saxon Ministry of Science and the Fine Arts.

REFERENCES

[1] T. Hoefler, J. Squyres, G. Bosilca, G. Fagg, A. Lumsdaine, and W. Rehm, "Non-Blocking Collective Operations for MPI-2," Open Systems Lab, Indiana University, Tech. Rep., 08 2006.

[2] T. Hoefler, J. Squyres, W. Rehm, and A. Lumsdaine, "A Case for Non-Blocking Collective Operations," in *Frontiers of High Performance Computing and Networking - ISPA 2006 Workshops*, vol. 4331/2006. Springer Berlin / Heidelberg, 12 2006, pp. 155–164. [Online]. Available: ./img/hoefler-ispa06.pdf

[3] F. Petrini, D. J. Kerbyson, and S. Pakin, "The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8, 192 Processors of ASCI Q." in *Proceedings of the ACM/IEEE Supercomputing*. ACM, 2003, p. 55.

[4] R. Dimitrov, "Overlapping of communication and computation and early binding: Fundamental mechanisms for improving parallel performance on clusters of workstations," Ph.D. dissertation, Mississippi State University, 2001.

[5] S. Gorlatch, "Send-receive considered harmful: Myths and realities of message passing," *ACM Trans. Program. Lang. Syst.*, vol. 26, no. 1, pp. 47–56, 2004.

[6] Message Passing Interface Forum, "MPI: A Message Passing Interface Standard," 1995.

[7] ——, "MPI-2: Extensions to the Message-Passing Interface," Technical Report, University of Tennessee, Knoxville, 1997.

[8] ——, "MPI-2 Journal of Development," July 1997.

[9] A. Danalis, K.-Y. Kim, L. Pollock, and M. Swany, "Transformations to parallel codes for communication-computation overlap," in *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 2005, p. 58.

[10] L. V. Kale, S. Kumar, and K. Vardarajan, "A Framework for Collective Personalized Communication," in *Proceedings of IPDPS'03*, Nice, France, April 2003.

[11] T. Hoefler and A. Lumsdaine, "Design, Implementation, and Usage of LibNBC," Open Systems Lab, Indiana University, Tech. Rep., 08 2006.

[12] T. Hoefler, A. Lumsdaine, and W. Rehm, "Implementation and performance analysis of non-blocking collective operations for mpi," in *Proceedings of the 2007 ACM/IEEE conference on Supercomputing (CDROM)*, 2007.

[13] C. Iancu, P. Husbands, and P. Hargrove, "Hunting the overlap," in *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 279–290.

[14] J. W. III and S. Bova, "Where's the Overlap? - An Analysis of Popular MPI Implementations," 1999. [Online]. Available: citeseer.ist.psu.edu/white99wheres.html

[15] W. Lawry, C. Wilson, A. B. Maccabe, and R. Brightwell, "Comb: A portable benchmark suite for assessing mpi overlap." in *CLUSTER*. IEEE Computer Society, 2002, pp. 472–475.

[16] T. Hoefler and A. Lumsdaine, "Design and implementation of the nbc library," Indiana University, Tech. Rep., 2006.

[17] The InfiniBand Trade Association, *Infiniband Architecture Specification Volume 1, Release 1.2*, InfiniBand Trade Association, 2003.

[18] T. Hoefler, C. Viertel, T. Mehlan, F. Mietke, and W. Rehm, "Assessing Single-Message and Multi-Node Communication Performance of InfiniBand," in *Proceedings of IEEE PARELEC 2006*. IEEE Computer Society, 9 2006, pp. 227–232.

[19] M. J. Koop, S. Sur, Q. Gao, and D. K. Panda, "High performance MPI design using unreliable datagram for ultra-scale InfiniBand clusters," in *Proceedings of the 21st annual international conference on Supercomputing*. New York, NY, USA: ACM Press, 2007, pp. 180–189.

[20] A. Friedley, T. Hoefler, M. L. Leininger, and A. Lumsdaine, "Scalable High Performance Message Passing over InfiniBand for Open MPI," in *Proceedings of 2007 KiCC Workshop, RWTH Aachen*, December 2007.

[21] M. J. Koop, S. Sur, Q. Gao, and D. K. Panda, "Zero-Copy Protocol for MPI using InfiniBand Unreliable Datagram," in *IEEE Cluster 2007: International Conference on Cluster Computing*, Austin, TX, USA, September 17-20, 2007.

[22] F. Mietke, R. Baumgartl, R. Rex, T. Mehlan, T. Hoefler, and W. Rehm, "Analysis of the Memory Registration Process in the Mellanox InfiniBand Software Stack," in *Euro-Par 2006 Parallel Processing*. Springer-Verlag Berlin, 8 2006, pp. 124–133.

[23] J. Liu, J. Wu, and D. K. Panda, "High Performance RDMA-Based MPI Implementation over InfiniBand," *Int'l Journal of Parallel Programming, 2004*, 2004.

[24] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation," in *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004.

[25] G. M. Shipman, T. S. Woodall, G. Bosilca, R. ch L. Graham, and A. B. Maccabe, "High performance RDMA protocols in HPC," in *Proceedings, 13th European PVM/MPI Users' Group Meeting*, ser. Lecture Notes in Computer Science. Bonn, Germany: Springer-Verlag, September 2006.

[26] C. L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krogh, "Basic Linear Algebra Subprograms for FORTRAN usage," in *In ACM Trans. Math. Soft., 5 (1979), pp. 308-323*, 1979.

[27] W. Gropp and E. L. Lusk, "Reproducible measurements of mpi performance characteristics," in *Proceedings of the 6th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. London, UK: Springer-Verlag, 1999.

[28] T. Worsch, R. Reussner, and W. Augustin, "On benchmarking collective mpi operations," in *Proceedings of the 9th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. London, UK: Springer-Verlag, 2002.

[29] T. Hoefler, T. Schneider, and A. Lumsdaine, "Accurately Measuring Collective Operations at Massive Scale," in *Accepted to the PMEO-PDS 08 workshop in conjunction with the 22nd International Parallel and Distributed Processing Symposium (IPDPS)*, April 2008.

[30] S. Sur, H.-W. Jin, L. Chai, and D. K. Panda, "Rdma read based rendezvous protocol for mpi over infiniband: design alternatives and benefits," in *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM, 2006, pp. 32–39.

[31] T. Hoefler, T. Mehlan, A. Lumsdaine, and W. Rehm, "Netgauge: A Network Performance Measurement Framework," in *Proceedings of Third International Conference, HPCC 2007*, vol. 4782. Springer, 9 2007, pp. 659–671. [Online]. Available: ./img/hoefler-netgauge.pdf

[32] T. Hoefler, A. Lichei, and W. Rehm, "Low-Overhead LogGP Parameter Assessment for Modern Interconnection Networks," 03 2007. [Online]. Available: ./img/hoefler-pmeo07.pdf

[33] Intel Corporation, "Intel Application Notes - Using the RDTSC Instruction for Performance Monitoring," Intel, Tech. Rep., 1997.

[34] J. M. Squyres and A. Lumsdaine, "The Component Architecture of Open MPI: Enabling Third-Party Collective Algorithms," in *18th ACM International Conference on Supercomputing, Workshop on Component Models and Systems for Grid Applications*, St. Malo, France, 2004.

[35] T. Hoefler, C. Siebert, and W. Rehm, "A practically constant-time MPI Broadcast Algorithm for large-scale InfiniBand Clusters with Multicast," in *Proceedings of the 21st IEEE International Parallel & Distributed Processing Symposium*. IEEE Computer Society, 03 2007, p. 232. [Online]. Available: ./img/hoefler-cac07.pdf

[36] T. Hoefler, T. Mehlan, F. Mietke, and W. Rehm, "Fast Barrier Synchronization for InfiniBand," in *Proceedings, 20th International Parallel and Distributed Processing Symposium IPDPS 2006 (CAC 06)*, April 2006.