Message Progression in Parallel Computing - To Thread or not to Thread?

Torsten Hoefler [1], Andrew Lumsdaine [2]

Open Systems Laboratory, Indiana University
150 S Woodlawn Ave, Bloomington, IN 47405, USA
[1] htor@cs.indiana.edu
[2] lums@cs.indiana.edu

***Abstract*—Message progression schemes that enable communication and computation to be overlapped have the potential to improve the performance of parallel applications. With currently available high-performance networks there are several options for making progress: manual progression, use of a progress thread, and communication offload. In this paper we analyze threaded progression approaches, comparing the effects of using shared or dedicated CPU cores for progression. To perform these comparisons, we propose time-based and work-based benchmark schemes. As expected, threaded progression performs well when a spare core is available to be dedicated to communication progression, but a number of operating system effects prevent the same benefits from being obtained when communication progress must share a core with computation. We show that some limited performance improvement can be obtained in the shared-core case by real-time scheduling of the progress thread.**

I. Introduction

Asynchronous progression of communications is an important and controversial topic in high-performance computing. It is relatively clear that one can leverage the hardware parallelism of the network and the CPU by using both entities at the same time. However, achieving this task is usually tedious and often involves restructuring of the parallel algorithm or at least manual transformations of the parallel code. Researchers have often been disappointed after they invested a huge effort to achieve overlapping of communication and computation, because the underlying middleware did not support asynchronous progress efficiently. The most widely used communication library standard, the message passing interface (MPI) standard, offers non-blocking routines that enable overlapping of communication and computation. However, the standard does not define a clear progression rule and leaves it up to a "high quality" implementation to offer true asynchronous progress. Thus, many early MPI libraries did not offer progression and just performed all communication in the respective test or wait calls.

However, some researchers have been able to provide significant speedups using overlap of communication and computation (up to 1.9 in [1]). Brightwell et al. [2] classifies the source of performance advantage for overlap and Dimitrov [3] uses overlapping as fundamental approach to optimize parallel applications for cluster systems. Thus, we conclude that overlapping of communication and computation is an important optimization technique. Figure 1 shows the effects of asynchronous or wait-based communication to parallel codes performing overlapping communications.

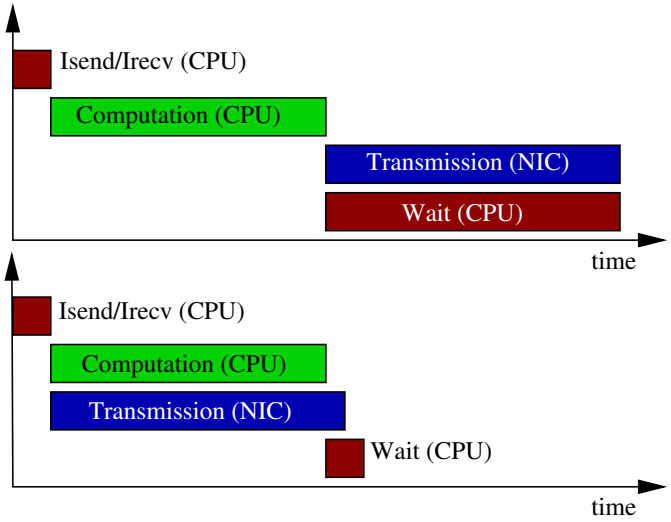

Fig. 1. Fully asynchronous versus wait-based message transmission for overlapped communication

Several analyses, such as [4] or [5] measured the overlap of different MPI implementations and lead to unsatisfying results. Especially the analysis of applying overlapping techniques to parallel Fast Fourier Transformations led to controversial results [6], [7], [8], [9], which is most likely due to the different abilities of the underlying communication system to perform asynchronous message progression. We showed in [10] that high-level communication operations, non-blocking collective operations, can improve application performance if the communication progresses in the background. However, those operations add another layer of complexity to the progression discussion.

A common assumption is that the progress just happens in the background without user intervention. While this might be true for some communication libraries or systems, there are cases where the user needs to progress the messaging subsystem manually. An easy way to do this for MPI is to call MPI_Test on the outstanding communication requests periodically because the MPI standard guarantees that this will eventually finish the transmission (assuming a correct

MPI program). But this technique might not be practicable in all cases, for example when the programmer overlaps a single library call to some serial computation library (e.g., BLAS [11]). Also, managing the requests and the test calls can be a huge burden for the user. Thus, a high quality implementation should ensure fully asynchronous progress whenever possible. In this paper, we analyze and evaluate different options for message progression at the point-to-point and collective level with InfiniBand as an example network. The next section discusses well-known strategies to implement point-to-point messages and to interact with the communication hardware.

II. Messaging Strategies in Communication Middleware

To discuss message progression schemes efficiently, we introduce common messaging protocols and implementation options. Most MPI libraries implement two different protocols to transmit messages. Dependent on the message-size, either an eager or a rendezvous protocol is selected to implement the message transmission. Eager transmissions send the message without synchronization to the receiver where it is buffered until the application process receives it. The rendezvous protocol delays the message transmission until the receive process has posted the receive operation to the library. Another option is to use pipelined message transmission in the rendezvous protocol [12].

Those protocols build on single message sends. The eager protocol sends the message directly from the sender to the receiver. However, the rendezvous protocol requires at least two synchronization messages and the actual data transmission, which might be pipelined and thus consist of many send operations. Different strategies to send those messages are based on the two simple operating system (OS) concepts, polling and interrupt. Most middleware systems only implement polling mode (i.e., the program spins on the main CPU while querying the hardware) for user-level messaging to enable OS bypass. The other option, interrupts, requires interaction with the OS which might suspend the process while waiting. It is assumed that the necessary syscall (privilege change) to enter the operating system code is rather expensive and thus, many modern messaging systems focus on OS-bypass schemes where all communication is performed in userspace. Thus, the polling based approach delivers, due to OS bypass, a slightly lower point-to-point latency and is therefore used in common high-performance MPI libraries for InfiniBand such as Open MPI and MVAPICH.

A more complex issue is the development of non-blocking high-level communication routines that involve interactions between multiple processes. Similar overlapping principles than in the point-to-point case can be used with those operations. However, the optimization for overlap is much more complicated because the communication protocols and algorithms are becoming significantly more complex as in the point-to-point case. Thus, we focus on the more complicated case to analyze the overlap of non-blocking collective operations in our work which of course also covers point-to-point progression.

III. Message Progression Strategies

Three fundamentally different messaging strategies can be found in parallel systems. A common strategy is to enforce manual progression by the user. This is frequently perceived as no progression because the programmers often do not progress or can not progress the library manually. A second strategy is the hardware-based approach where the message handling is done in the network interface card. The third approach, using threads for progression, is often discussed as the "silver bullet" but it has not found widespread adoption yet.

A. Manual progression

This scheme is the simplest to implement from the MPI implementer's perspective because there is no asynchronous progress. Every time, the user calls MPI_Test with a request, the library checks if it can make any progress on this request. Thus, the complete control and responsibility is given to the user in this case. There are several problems with this approach. The biggest problem is the opaqueness of the MPI library, i.e., the user does not know about the protocol and the status of a specific operation. Thus, for portable programs, he has to assume the worst case for asynchronous progress, the pipeline protocol, where he has to call MPI_Test to progress every fragment in order to achieve good overlap. However, the missing status information forces the user to adopt a black box strategy.

We proposed a black box testing scheme in [13] that issues N tests during a message transmission. N is defined as a function of the message size in order to reflect the needs of the pipelined protocol:

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

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. The scheme is shown in Figure 2.

B. Hardware-based progression

A possible solution to ensure full asynchronous progress is to do the protocol processing in the communication hardware. The Myrinet interconnection network offers a programmable network interface card (NIC) and several schemes have been proposed to offload protocol processing and message matching on this external CPU [14]. A similar offload scheme was proposed for Ethernet in [15]. Some proposals, such as [16], also implement NIC-based message broadcast schemes. The relatively simple barrier operation has also been implemented with hardware support [17]. Other schemes [18], [19] support collective operation offload for some operations but impose some limitations. However, none of those implementations allow overlap they only offer a blocking interface.

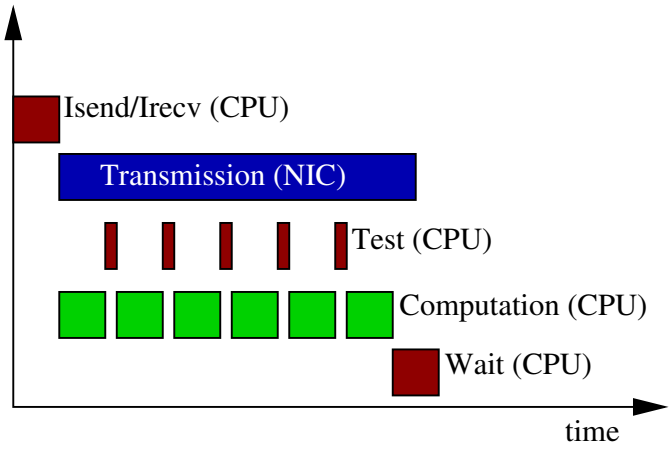

Fig. 2. Black-box test based manual progression strategy

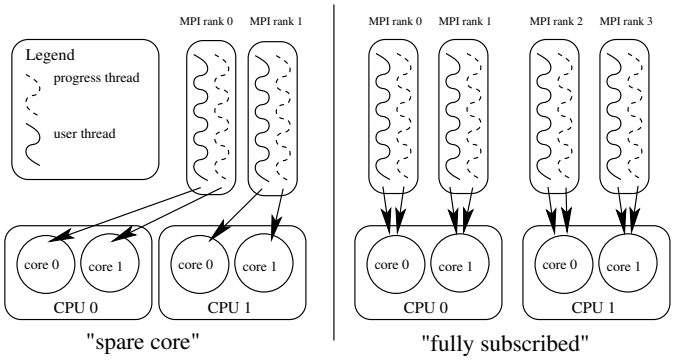

Fig. 3. Different subscription schemes for a dual-CPU dual-core configuration

C. Threads for Message Progression

Asynchronous progression threads have often been stated as a silver bullet in future work, but they are not widely used. This might be because the threaded programming model puts a huge burden on the system software implementer because the whole driver infrastructure and all libraries must be implemented reentrant (thread safe). However, some libraries, for example Open MPI, begin to explore the possibility of threaded progress. Other libraries like MPI/pro or HP MPI offer asynchronous progression threads but have not been analyzed in detail.

Threads have usually been used in high performance computing to implement thread-level parallelism (OpenMP, also in combination with MPI as a hybrid programming model [20]) or for other tasks that are not critical for communication, such as checkpoint/restart functionality.

Threads are a very promising model for asynchronous progression. One of the biggest problems with manual progression strategies is that it is very unlikely that MPI_Test hits the ideal time. It comes either too early and there is nothing to progress or too late and overlap potential is wasted. A threaded implementation would be either polling and thus get all messages immediately or the thread could be woken up and also progresses the communication layer at exactly the right times. A progress thread also enables fully asynchronous progression, i.e., without any user interaction. We will focus on a threaded progression of non-blocking collective operations in the following sections. However, before introducing our implementation, we discuss several operating system effects that influence the execution of messagingmessaging threads.

1) Operating System Effects: There was great effort to circumvent the operating system in the past with so called OS-bypass methods. User-level networking without operating system support was implemented for most modern network interconnects to enable lower latencies and avoid system calls. However, the operating system plays an important role in the administration of user and progress threads. We have to distinguish the different methods to access the network hardware (polling, interrupt) and the subscription factor of the cores (all cores run user threads vs. spare cores can run progression threads). The subscription scheme is shows for a rather common dual-CPU dual-core combination in Figure 3. Combining both schemes leads us to the following 4 combinations:

access method	core subscription
polling	fully subscribed
polling	spare cores available
interrupt	fully subscribed
interrupt	spare cores available

a) Operating System Scheduling: The operating system scheduler usually arranges the threads (or processes) in two or more more queues, a runnable queue and a waiting queue. Threads (or processes) in the runnable queue are waiting for the CPU and share this resource among each other, threads on the waiting queue wait for some other hardware event (i.e., a packet from the wire). To ensure fairness of the CPU sharing, each process has a time-slice to run on the CPU. If this time-slice is over, the scheduler schedules other runnable threads. The scheduler bases its decision on the thread priorities (which depends on the particular operating system). Typical length of time-slices are between 4 and 10 milliseconds. The Linux 2.6 default scheduler (called O(1) scheduler) implements such a time-slice based mechanism.

The analysis is easy if a spare core is available to each MPI process to run the progression thread. The difference between the polling (common implementation) and the interrupt based approach is that the polling might reduce transmission latencies slightly (due to OS bypass). The interrupt based approach might be more power efficient (since the hardware is idle during the message transmission) but might also have some effect to the performance of the operating system (and thus to other threads). However, this highly depends on the implementation of the OS (i.e., how is the locking implemented? does the scheduling overhead depend on the number of threads? ...).

The analysis is much more complicated if there are no idle cores available. We would argue that this is the common case in today's systems, i.e., if a user has 4 cores per machine, he usually launches 4 (MPI) processes on each

machine to achieve highest performance. In this scenario, the progression thread has to share the CPU with the computation thread. In the polling approach, the computation thread and the progression thread are both runnable all the time which leads to heavy contention in the fully subscribed case. This effectively halves the CPU availability for the computation thread and thus also halves the overall performance. Those effects can be limited by calling `sched_yield()` in the progression thread after some poll operations which leads to a re-scheduling. However, the progression thread is still runnable and will be re-scheduled depending on priority. The interrupt approach seems much more useful in this scenario because the progression thread goes to sleep (enters the wait queue) when no work is to be done and is woken up (enters the run queue) when specific network events (e.g., a packet is received) occur. This schedules the thread at exactly the right time (work is to be done) and is thus significantly different from the manual progression and the polling approach.

The interrupt-based mechanism raises two concerns. First, It seems unclear how big the interrupt latency and overheads are on modern systems. Second, the scheduler has to schedule the progression thread immediately after the interrupt arrives to achieve asynchronous progress. It is not sufficient if the thread is just put on the run queue and the computation thread is re-scheduled to finish its time-slice. Waiting until the time-slice of the active thread is finished increases the interrupt-to-run latencies by a time-slice/2 on average which effectively disables asynchronous progress because the time slices are one to two order of magnitudes higher than the transmission latency of modern networks. Unfortunately, this mechanism is common practice to ensure fairness, i.e., avoid processes that get many interrupts to preempt other compute-bound processes all the time. The Linux scheduler favors I/O bound processes but it might still not be sufficient to achieve highest overlap. To overcome this problem, one can increase the relative priority of the progression thread. We experiment with the highest possible priority, real time threads in Linux. If a real time (RT) thread is runnable, then it preempts every other thread by default. This might decrease the interrupt-to-run latency significantly. However, this might also increase the interrupt and context switching overhead significantly because the thread is scheduled every time an interrupt occurs and goes to sleep shortly after. All options are illustrated in Figure 4. The next section discusses our implementation of non-blocking high-level communication operations and our extensions for threaded asynchronous progression.

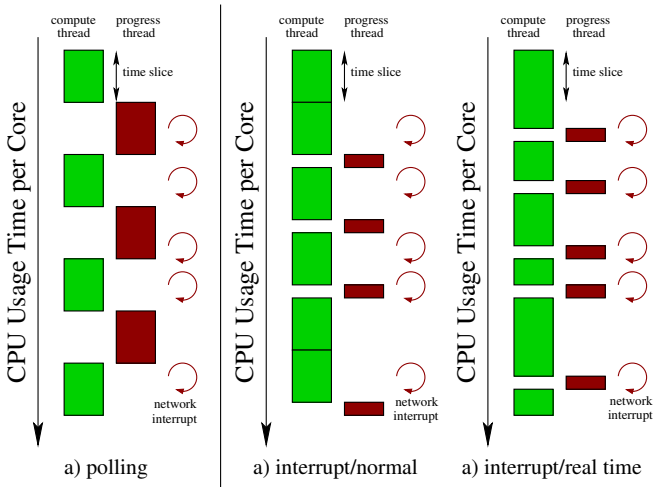

Fig. 4. CPU Scheduling Strategies for the fully subscribed case, comparing a polling progress thread with an interrupt based normal and real time progress thread.

IV. Implementation

We chose InfiniBand as an example network because it does not offer fully offloaded progression and is widely used in today's cluster systems. However, our results are generally applicable to all networks that base their progress on the CPU and consider threaded progression. Since there is no fully thread-safe MPI for InfiniBand available publicly, that allows to choose between polling and interrupts, we implemented our own Mini-MPI library that supports the subset of operations needed by our non-blocking collectives library LibNBC [10]. Our Mini-MPI, called LibOF [13], implements the standard non-blocking transmission functions (send/recv) in a way that enables highest asynchronous progress without user intervention. In this work, we extend and analyze LibOF to use a progression thread to ensure fully asynchronous progression.

A. Partially Asynchronous Collective Communication

The structure of LibNBC reflects the two levels of communication (point-to-point and collective). The user issues a non-blocking collective operation which returns a handle. This handle has a (potentially multi-round) schedule of the execution (cf. [10]) and a list of point-to-point operations of the current round attached. Progress is thus defined on the two levels, point-to-point progress and collective progress.

The two levels of progression for a binomial tree broadcast on four processes are displayed in Figure 5. Even though process 1 received the RTR message from process 3 early, it can only send the data after it received it from process 0. Those data-dependencies incurred by the collective algorithms add a new complexity to the progression.

The InfiniBand optimized LibOF supported nearly asynchronous point-to-point progress at the messaging level for the rendezvous protocol. The implemented wait-on-send strategy (the sender polls for a while to receive the ready-to-receive (RTR) message from the receiver) progresses if the nodes post the send and receive operations at similar times. However, if the RTR message arrives late, there will be no progress (unless the user progresses manually). The eager point-to-point protocol is fully asynchronous if free slots are available at the receiver-side. The implementation uses unsignaled RDMA-Write and polls the memory to detect memory completion. Consult [13] for a detailed analysis of the different progression schemes in comparison to manual progression. However, this protocol does not support progress on the collective messaging

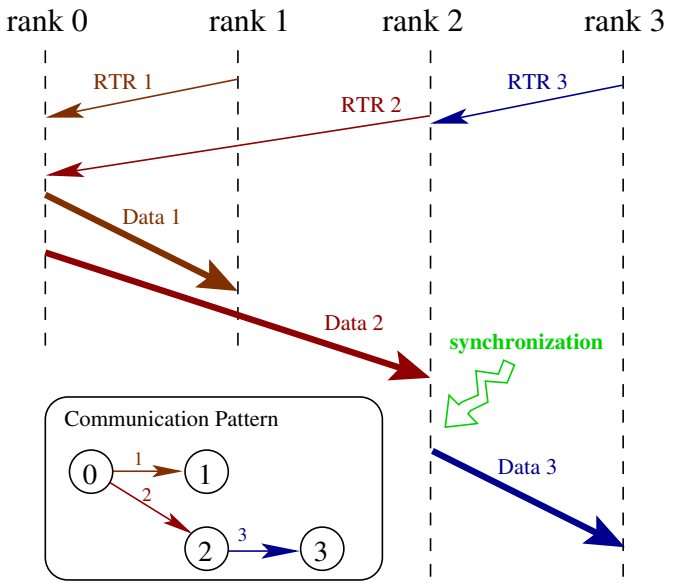

Fig. 5. Receiver initiated point-to-point messages for a tree broadcast pattern among four processes.

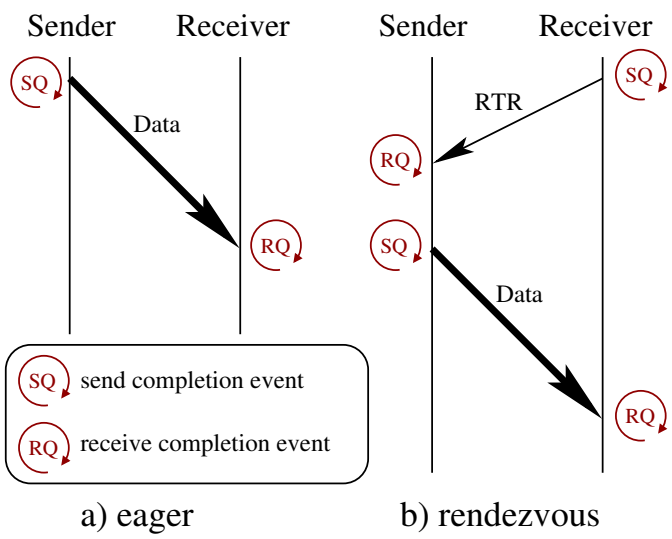

Fig. 6. Completion events generated by the hardware for both point-to-point protocols.

layer. If the collective algorithm consists of multiple communication rounds (e.g., broadcast is implemented as a tree or Allreduce in a pipeline way), only the first round is progressed automatically and the user has to progress the following rounds manually. This is a big limitation in the current implementation that our new threaded implementation seeks to overcome.

B. Fully Asynchronous Collective Communication

This section describes the design of the fully threaded version to deal with multiple communication contexts (MPI communicators) and multiple point-to-point InfiniBand connections (queue pairs (QP)). Our design focuses on an interrupt-based implementation because polling has been well analyzed in previous work (e.g., [21]). Each queue pair represents a channel between two hosts and is associated with a completion queue where events are posted. Those events could be the receipt of a new message or the notification of a message transmission. Each completion queue is associated with a so called completion channel to use interrupt driven message progression. Each completion channel offers a file descriptor that can be used in system calls (e.g., `select()` or `poll()`) to wait for events.

To get a notification for every packet, the implementation is changed to use RDMA-Write with immediate (signaled RDMA-Write) for every message transmission. This makes sure that the receiver receives an interrupt for incoming messages.

In our design, the progress thread handles all collective and point-to-point progressions. After the user thread posted a new collective operation, the new handle is added to the thread's worklist. This is one of two synchronization/locking points between the two threads. The second synchronization is when the user wants to wait on the communication where it waits until a (shared memory) semaphore attached to the handle becomes available (is activated by the progress thread). The test call just checks if the semaphore would block and returns true if not.

The progress thread itself generates a list of point-to-point requests from its collective worklist (every collective handle in the list has a list of point-to-point requests attached) and calls OF_Waitany() with the full list. When OF_Waitany() returns, a point-to-point request finished and the progress thread calls LibNBC's internal scheduler with the associated collective handle to see if any progress can be made on the collective layer. Then it compiles a new list of point-to-point requests (since a handle might have been changed) and enters OF_Waitany() again.

The implementation of OF_Waitany() uses the list of point-to-point requests to assemble a list of file pointers (by checking the associated completion channels). The function then blocks (with the `poll()` system call) until one of the file pointers gets available (i.e., a completion channel event occurred). Then it polls the associated completion queue, progresses the message for which the event occurred and returns if a message transmission finished. If no transmission finished, the function just enters the `poll()` call again. The number of completion events depends on the point-to-point protocol used. Figure 6 illustrates the protocol-dependent completions.

The program must also ensure that if the user issues a new collective operation, this is picked up by the thread (to avoid deadlocks and achieve best progress). In order to do so, a pipe read file descriptor is added to the list of file pointers to `poll()`. Whenever a new request is added to the threads worklist, it is woken up from the wait queue by writing to this pipe.

This scheme enables fully asynchronous progress and the collective operations are finished in the background without

any user interaction. The following sections discuss benchmarks and performance results for several different configurations.

V. Experimental Analysis

We benchmark our implementation on the Odin cluster at Indiana University. Odin consists of 128 AMD 270 HE dual-core dual-socket nodes. It runs Red Hat Enterprise Linux with a 2.6.9 kernel. The network interface cards are Mellanox MT23108 InfiniBand HCAs and the Open Fabrics Enterprise Edition version 1.2 is used as a communication library.

A. Point-to-point Overhead

We implemented a new communication pattern benchmark for Netgauge [22] that measures the overlap potential and communication overhead for point-to-point messages and different progression strategies. The benchmark works as follows:

1) benchmark the time t_b for a blocking communication
   (a) start timer t_b
   (b) start communication
   (c) wait for communication
   (d) stop timer t_b
2) start communication
3) compute for time t_b
   (a) endtime = current time + t_b
   (b) while(current time $<$ endtime) do computation
4) wait for communication to finish

The measurement is done as ping pong with pre-posted receives on the client side, i.e., "start communication" posts a non-blocking receive and a non-blocking send and "wait for communication" waits until both operations finished. The server side simply returns the packets to the sender. The overhead t_o is the sum of the times spent to start the communication, progress the communication (test) and wait for the communication to complete.

We compare the Open MPI implementation which needs manual message progression (cf. Section III) with our Mini-MPI implementation. The first experiment is a parameter study which aims to find the best parameters for the manual progression of Open MPI. We conducted benchmarks for tests every 2^n bytes $\forall n = 10..18$. The test every 65536 bytes performed best for most message-sizes.

The results between two Odin nodes are shown in Figure 7. It compares Open MPI (using the best test configuration for every 65536 bytes) with our threaded and non-threaded Mini-MPI implementation (LibOF). The results show that the overlap optimized Mini-MPI has a generally lower CPU overhead than the Open MPI implementation. Adding a progression thread to this implementation decreases the overhead due to offloading the communication processing to a spare core.

We conclude from those experiments that the threaded progression strategy can be very beneficial for point-to-point messaging if the progression thread runs on a separate CPU core. The next section analyzes the collective progression behavior where all cores might be busy with computation.

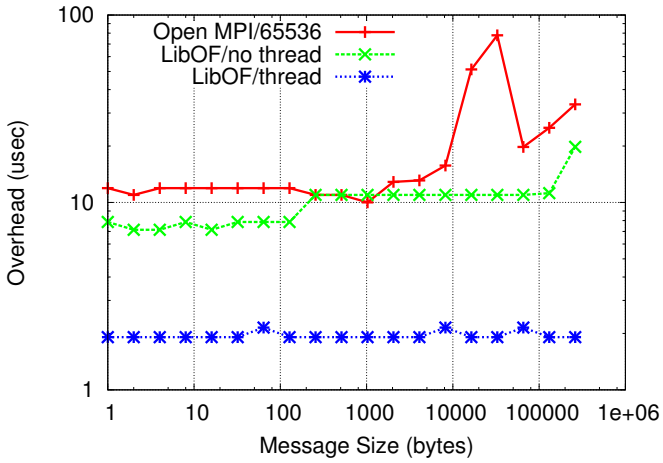

Fig. 7. Point-to-point overhead of the different implementation options.

B. Collective Overhead

In order to analyze CPU overhead for non-blocking collective operations, we implemented NBCBench [10], a benchmark following the same concept as our point-to-point measurements. The main difference is that there is no client/server concept since all processes are part of the collective call. This also requires a new synchronization scheme to ensure that all processes start at the same global time. We use the scheme outlined in our previous work on measuring collective operations [23] where every rank measures the difference between its local time and the time on rank 0. After that, rank 0 broadcasts a global starting time for the collective operation to all nodes. We apply a similar concept to generate computation time as in the point-to-point benchmark.

We benchmarked all collective operations and present Allreduce, the most important multi-round operation (cf. [24]). Other important single-round operations, like Alltoall have been evaluated in [13] and showed a high potential for overlap. The reduction operations are at the same time the most complicated operations to optimize for overhead. Those operations include, additionally to the communication, a computation step that uses a significant amount of CPU cycles in the reduction operation.

In our analysis we focus on the Allreduce operation which has been found to be he hardest to optimize for overlap [10]. All other operations perform (significantly) better (with lower overhead) in all benchmarked cases. The Allreduce implementation in LibNBC uses two different algorithms for small and large messages (cf. [25]). A simple binomial tree with a reduction to rank 0 followed by a broadcast on the same tree is used for messages smaller then $65kiB$. This algorithm has $2 \cdot log_2 P$ communication rounds on P processes. The large message algorithm chops the message into P chunks and performs $2 \cdot P$ communication rounds in a pipelined manner to reduce the data.

Figure 8 shows the overhead of a non-blocking Allreduce operation on 32 nodes with 1 process per node. The best test

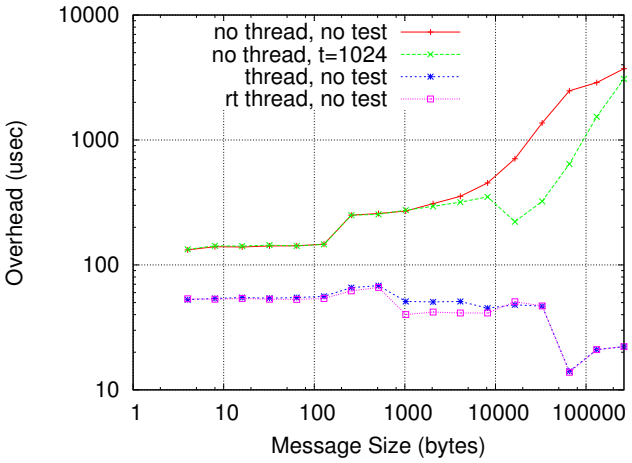

Fig. 8. NBC_Iallreduce overhead on 32 nodes, one process per node with different progression strategies

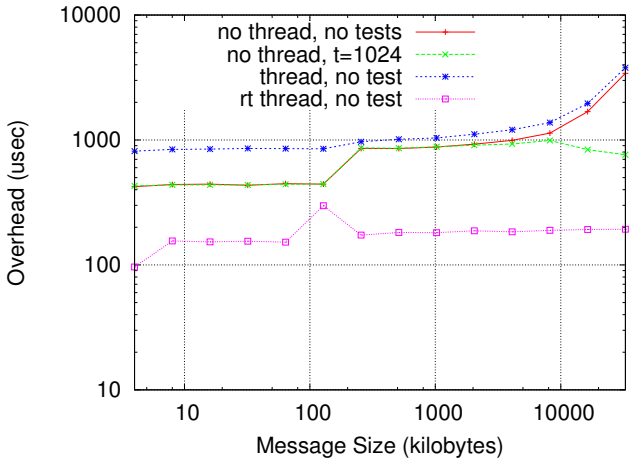

Fig. 9. NBC_Iallreduce overhead on 32 nodes, 4 processes per node with different progression strategies

strategy was to test every 1024 bytes, but this is significantly outperformed by the threaded implementation due to communication offload to spare cores. Real time threads do not improve performance in this scenario.

These results show that the threaded implementation is able to lower the CPU overhead by one order of magnitude if a spare CPU core is available to offload the computation. However, we showed in [21] that some applications can benefit from using all cores. That means that there might be no "free" cores on today's dual-, quad- or oct-core systems. Thus, we also have to evaluate our progression strategies in the case where all CPU cores are busy with user computation. However, in general, we assume that especially memory-bound and irregular algorithms like sparse solvers or graph problems can not use the growing number of available cores so that spare communication cores will be available for communication offload in the near future.

1) The Oversubscribed Case: Using MPI exclusively means running n MPI processes on each n-core node. This leads to a 2:1 oversubscription of threads vs. available cores. Other models, like hybrid MPI + OpenMP [20] might lead to only one progression thread per node but are more complicated to program and optimize.

To reflect the oversubscription case, we run 4 threads on each of the 32 test nodes of Odin. This leads to 128 processes performing the collective operations. Our results in Figure 9 show that our test strategy did not perform well when all cores are busy. The threaded implementation is also not able to decrease the communication overhead significantly due to scheduling problems in the operating system described in Section III-C1. However, those OS effects can be overcome with a real-time thread that preempts the user computation as soon as an InfiniBand completion event occurs.

The timing-based measurements only accounts for the CPU overhead of the actual communication calls (NBC_Iallreduce and NBC_Wait) but other overheads such as interrupt processing, time spent in the InfiniBand kernel driver and context switching overhead as well as cache pollution are not taken into account and could have a detrimental effect on real application performance. Thus, we analyze the number of context switches and the context switch and interrupt processing overhead in the following.

a) The number of context switches: The implementation allows for more then one event to be processed by the progress thread in a single interrupt. However, the real time scheduling might cause many context switches. The maximum number of context switches equals to the number of completion notifications which depends on the transport protocol and the collective algorithm. The eager protocol causes 1 completion on the sender and the receiver and the rendezvous protocol causes 2 completions on the sender the receiver as explained in Section IV-B. Thus, the tree-based small message algorithm causes up to $1 \cdot 2 \cdot log_2 P$ completion events in the eager case and $2 \cdot 2 \cdot log_2 P$ completion events in the rendezvous case (each intermediate node sends and receives $log_2 P$ messages). The number of interrupts is thus 14 or 28 in our example with 128 processes. The pipelined large message algorithm causes up to $1 \cdot 2 \cdot (2 \cdot P - 2)$ completions for eager messages and $2 \cdot 2 \cdot (2 \cdot P - 2)$ for rendezvous messages (each node sends and receives a single packet in $2 \cdot P - 2$ rounds. In our example with $P = 128$, this equals to 508 or 1016 interrupts.

2) Interrupt and Context Switch Overhead: It is not clear how much a threaded implementation suffers from context switch and system interrupt overhead. In our model, the latency incurred by those operations is less important because we assume that this will be overlapped with computation. The most important measure in our model is the CPU overhead, i.e., how many CPU cycles the interrupt processing and context switch "steals" from the user application. We describe a simple microbenchmark to assess the context switching and interrupt overhead in the following.

The benchmark measures the time to process a fixed work-

load on every of the c CPU cores. In order to do this, it spawns one computation thread on each core i that records the time t_1^i to compute the fixed problem in a loop. The benchmark has two stages, stage one measures the normal case where no extra interrupts are generated/received[1]. Then, the main thread that has been sleeping so far programs the real time clock interrupt timer to the highest possible frequency $f = 8192\,Hz$ for stage 2. In this stage, the main thread receives those interrupts and thus steals computation cycles from the worker threads that benchmark t_2^i on each core. The results from all cores for the two stages are averaged into $t_1 = \sum_i t_1^i/c$ and $t_2 = \sum_i t_2^i/c$. The difference $t = t_2 - t_1$ is the time that is added by the interrupts and subsequent context switches. The number of interrupts in stage 2 can be estimated with $i = t_2 \cdot 8192\,Hz$.

We ran this benchmark on the Odin cluster with 4 computation threads. The average fixed computation of 7 measurements in stage 1 was $t_1 = 19.14626\,s$ and in stage 2 $t_2 = 19.27969\,s$. The number of interrupts in stage 2 was thus $j = t_2 \cdot f \approx 157939$. The j interrupts delayed the work by $t = 133430\,\mu s$, thus yielding a CPU overhead per interrupt per core of $133430/1579394 \cdot 4 = 3.38\,\mu s$.

We show that interrupts and context switching between threads causes about $3.4\mu s$ overhead on our test system and is thus relatively expensive. Based on that, we conclude that frequent context switches increase the overhead significantly. Especially for the large-message Allreduce algorithm that might receive up to 1016 interrupts which would mean a constant overhead of $3.4ms$ per core. However, this overhead is not reflected in our current time-based computation analysis. Additional other overheads like the time spent in the InfiniBand driver stack and side effects like cache pollution are not modeled so far. To overcome this limitation, we propose a workload-based benchmark that simulates a real-world application with the computation of a constant workload in the different scenarios.

3) Workload-based overhead benchmark: We extend NBCBench with a workload-based computation scheme. The first step to obtain the blocking time t_b remains the same.

1) benchmark the time t_b for a blocking communication
2) find workload λ that needs t_b to be computed
   (a) $\lambda = 0$
   (b) while($t_\lambda < t_b$) { increase workload λ by δ;
   t_λ = time to compute workload λ }
3) start timer t_{ov}
4) start communication
5) compute fixed workload λ
6) wait for communication
7) stop timer t_{ov}

The overhead t_o in this case is the difference between the time for the overlapped case (computation and communication) and the computation time, thus $t_o = t_{ov} - t_c$.

We repeated the benchmarks with the new fixed workload scheme. The results for a single MPI process per node are as

[1]"no extra" means only the normal background "noise" in this case

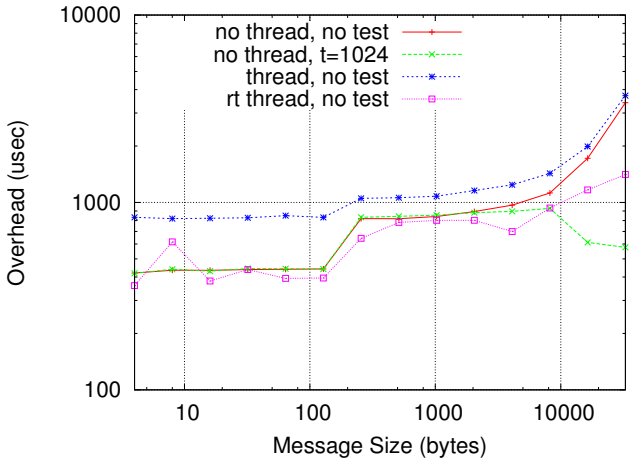

Fig. 10. NBC_Iallreduce overhead on 32 nodes, 4 processes per node with different progression strategies for the work-based benchmark

expected very similar to the results with the time-based benchmark and thus omitted due to space restrictions. However, the results in the overloaded case with 4 processes per node are rather different and shown in Figure 10. This benchmark reveals that even the real time thread is not able to decrease the communication overhead by an order of magnitude as shown with the time-based benchmark. However, the performance improvement is still significant with about a factor of two improvement (note the logarithmic scale of the graph) but mitigated by different sources of overhead, such as context switching, cache pollution and interrupt or driver processing. It is also interesting, that the right test strategy (in this case every 1024 bytes) is able to deliver higher performance for some message sizes due to the relatively low overhead of the test calls.

Large Allreduce messages and other collectives have also been studied but are not shown here due to space restrictions. It has been shown that large messages are easy to overlap with a test-based strategy [10]. Our results with the threaded approach support the previous results and the real-time thread performs an order magnitude better the normal threading in the time-based benchmark. This is reduced to a factor of two in the work-based benchmark that takes the different sources of overhead into account. Other collective operations, such as Reduce, Bcast, Alltoall, Allgather show significantly better results than the complex Allreduce operation because they do either not involve computation or deliver the result to a single host only. Due to space restrictions, we focused our analysis on the complex Allreduce operation which is also the most important collective operation.

C. Overcoming the Threading Issues

There are different ways to mitigate or even overcome the problems with threaded progression. The most obvious way would be to limit the number of interrupts by intelligent coalescing. This means that only the events that are important for progression (e.g., no local completions) generate an interrupt

and wake the progress thread up. This technique is already used in Myrinet/MX to progress point-to-point communication and is able to at least halve the number of interrupts and thus reduce the overhead significantly.

Onother easy change would be to replace the thread-based mechanism with a signal based concept, where the progression code is executed by a signal handler in the same thread. This would save the context switching and scheduler overhead time. However, the current implementation of signals is unreliable and needs to be made reliable to avoid deadlocks.

Another way would be to implement the whole progression engine inside the OS kernel. This would also eliminate context switches, scheduler overhead and also the expensive privilege changes between user- and kernel-space. Since the scheduler design is rather simple, this could be a viable solution to the progression problem. With this, we argue that operating system bypass might not be beneficial in all scenarios. Magoutis et al. also mention several other benefit of kernel-level I/O in [26].

The third and theoretically best but also most expensive way is to implement the whole high-level operation in the network hardware. Approaches to do full point-to-point message progression in the network interface card have been described in Section III-B. This would need to be extended with functionality to handle higher level communication patterns.

VI. Conclusions and Future Work

We analyzed different strategies for asynchronous progression for non-blocking collective communication operations in message passing libraries. We analyzed polling and interrupt based threaded implementations and can conclude that polling based implementations are only beneficial if separate computation cores are available for the progression threads. The interrupt-based implementation might also be helpful in the oversubscribed case (i.e., the progress and user thread share a computation core) but this depends on the collective operation as well as on operating system parameters. We found that the progression thread needs to be scheduled immediately after a network event to ensure asynchronous progress. A good way to implement this is the usage of real-time functionality in the current Linux kernel. Our analyses for the most complicated operation, Allreduce, show that it is hard to achieve high overlap. However, further analyses show that other simpler operations, like Reductions or Bcast perform very well in our model. All programs used in this article are available on the LibNBC webpage.

We also proposed several mechanisms to mitigate the different sources of overhead that we identified in this article. We will also investigate different collective algorithms that can further improve the CPU availability to the user thread.

Acknowledgments

The authors want to thank Douglas Gregor (Indiana University) and Tim Mattox (Indiana University) for helpful comments and suggestions and Frank Mietke (Technical University of Chemnitz) for his technical support during the evaluation phase. This work was partially 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.

References

[1] G. Liu and T. Abdelrahman, "Computation-communication overlap on network-of-workstation multiprocessors," in *Proc. of the Int'l Conference on Parallel and Distributed Processing Techniques and Applications*, July 1998, pp. 1635–1642.

[2] R. Brightwell and K. D. Underwood, "An analysis of the impact of MPI overlap and independent progress," in *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*. New York, NY, USA: ACM Press, 2004, pp. 298–305.

[3] 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.

[4] 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.

[5] 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

[6] A. Adelmann, W. P. P. A. Bonelli and, and C. W. Ueberhuber, "Communication efficiency of parallel 3d ffts." in *High Performance Computing for Computational Science - VECPAR 2004, 6th International Conference, Valencia, Spain, June 28-30, 2004, Revised Selected and Invited Papers*, ser. Lecture Notes in Computer Science, vol. 3402. Springer, 2004, pp. 901–907.

[7] C. Calvin and F. Desprez, "Minimizing communication overhead using pipelining for multidimensional fft on distributed memory machines," 1993.

[8] A. Dubey and D. Tessera, "Redistribution strategies for portable parallel FFT: a case study." *Concurrency and Computation: Practice and Experience*, vol. 13, no. 3, pp. 209–220, 2001.

[9] S. Goedecker, M. Boulet, and T. Deutsch, "An efficient 3-dim FFT for plane wave electronic structure calculations on massively parallel machines composed of multiprocessor nodes," *Computer Physics Communications*, vol. 154, pp. 105–110, Aug. 2003.

[10] T. Hoefler, A. Lumsdaine, and W. Rehm, "Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI," in *In proceedings of the 2007 International Conference on High Performance Computing, Networking, Storage and Analysis, SC07*. IEEE Computer Society/ACM, 11 2007. [Online]. Available: ./img/hoefler-sc07.pdf

[11] 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.

[12] 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.

[13] T. Hoefler and A. Lumsdaine, "Optimizing non-blocking Collective Operations for InfiniBand," in *Proceedings of the 22nd IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 04 2008. [Online]. Available: ./img//hoefler-cac08.pdf

[14] C. Keppitiyagama and A. S. Wagner, "Asynchronous mpi messaging on myrinet," in *IPDPS '01: Proceedings of the 15th International Parallel & Distributed Processing Symposium*. Washington, DC, USA: IEEE Computer Society, 2001, p. 50.

[15] P. Shivam, P. Wyckoff, and D. Panda, "Emp: zero-copy os-bypass nic-driven gigabit ethernet message passing," in *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*. New York, NY, USA: ACM Press, 2001, pp. 57–57.

[16] W. Yu, D. Buntinas, and D. K. Panda, "High Performance and Reliable NIC-Based Multicast over Myrinet/GM-2," pp. 197–204, 2003.

[17] W. Yu, D. Buntinas, R. L. Graham, and D. K. Panda, "Efficient and scalable barrier over quadrics and myrinet with a new nic-based collective message passing protocol." in *18th International Parallel and Distributed Processing Symposium (IPDPS 2004), CD-ROM / Abstracts Proceedings, 26-30 April 2004, Santa Fe, New Mexico, USA*, 2004.

[18] A. Wagner, D. Buntinas, D. K. Panda, and R. Brightwell, "Application-bypass reduction for large-scale clusters." in *2003 IEEE International Conference on Cluster Computing (CLUSTER 2003)*. IEEE Computer Society, December 2003, pp. 404–411.

[19] D. Buntinas, D. K. Panda, and R. Brightwell, "Application-bypass broadcast in mpich over gm," in *CCGRID '03: Proceedings of the 3st International Symposium on Cluster Computing and the Grid*. Washington, DC, USA: IEEE Computer Society, 2003, p. 2.

[20] R. Rabenseifner, "Hybrid parallel programming on hpc platforms," in *In proceedings of the Fifth European Workshop on OpenMP, EWOMP'03*, Aachen, Germany, 2003.

[21] 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, EuroPVM/MPI 2007*, vol. 4757. Springer, 10 2007, pp. 125–134. [Online]. Available: ./img/hoefler-nbc-standard.pdf

[22] T. Hoefler, T. Mehlan, A. Lumsdaine, and W. Rehm, "Netgauge: A Network Performance Measurement Framework," in *High Performance Computing and Communications, Third International Conference, HPCC 2007, Houston, USA, September 26-28, 2007, Proceedings*, vol. 4782. Springer, 9 2007, pp. 659–671.

[23] T. Hoefler, T. Schneider, and A. Lumsdaine, "Accurately Measuring Collective Operations at Massive Scale," in *Proceedings of the 22nd IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 04 2008. [Online]. Available: ./img/hoefler-pmeo08.pdf

[24] R. Rabenseifner, "Automatic MPI Counter Profiling," in *Proceedings of 42nd CUG Conference*, 2000.

[25] J. Pjesivac-Grbovic, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra, "Performance Analysis of MPI Collective Operations," in *Proceedings of the 19th International Parallel and Distributed Processing Symposium, 4th International Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems (PMEO-PDS 05)*, Denver, CO, April 2005.

[26] K. Magoutis, M. I. Seltzer, and E. Gabber, "The case against user-level networking," in *Proceedings of Workshop on Novel Uses of System-Area Networks (SAN-3)*, Madrid, Spain, 2004.