Kernel-Based Offload of Collective Operations – Implementation, Evaluation and Lessons Learned

Timo Schneider[1], Sven Eckelmann[1], Torsten Hoefler[2], and Wolfgang Rehm[1]

[1] TU Chemnitz, Germany {timos,ecsv,rehm}@hrz.tu-chemnitz.de
[2] University of Illinois at Urbana-Champaign, IL, USA, htor@illinois.edu

Abstract. Optimized implementations of blocking and nonblocking collective operations are most important for scalable high-performance applications. Offloading such collective operations into the communication layer can improve performance and asynchronous progression of the operations. However, it is most important that such offloading schemes remain flexible in order to support user-defined (sparse neighbor) collective communications. In this work, we describe an operating system kernel-based architecture for implementing an interpreter for the flexible Group Operation Assembly Language (GOAL) framework to offload collective communications. We describe an optimized scheme to store the schedules that define the collective operations and show an extension to profile the performance of the kernel layer. Our microbenchmarks demonstrate the effectiveness of the approach and we show performance improvements over traditional progression in user-space. We also discuss complications with the design and offloading strategies in general.

1 Introduction

The Message Passing Interface (MPI) standard [12] is the de-facto standard for implementing today's large-scale high-performance applications. Part of MPI's success is it's high portability, not only from a correctness, but also from a performance perspective. This is achieved by defining several high-level collective communication operations that specify communication primitives on groups of processes instead of process pairs. The implementation of such collective operations can now be optimized to the particular machine architecture and network topology. Several non-trivial algorithms have been developed to optimize those group communications, e.g., [2, 16].

A recent addition to the MPI standard (in the upcoming MPI-3.0 standard) builds upon this success and introduces nonblocking versions of all MPI collective operations [7]. Nonblocking collective operations allow the application to perform computations while the communication (and synchronization) is performed "in the background".

Different software implementation options have been explored for different network architectures [6] but the major problem, how to progress the collective

algorithm efficiently, remains open. This problem exists because most advanced collective algorithms have multiple stages where a stage can only be started if some preconditions are satisfied. A simple example is a binary tree where an inner node can only send the message to its children after it has received it from its parents. Checking if a message was received and conditionally starting new transmissions requires to transfer the program control from the application to the collective implementation. Hoefler and Lumsdaine analyzed in [5] different schemes to progress the communication subsystem. Their study showed that, without loosing CPU power (cores), the application needs to progress the library by calling it regularly (e.g., with MPI_Test()). This solution is of course not feasible in the general case due to long calls to libraries that are not MPI-aware (e.g., Level 3 BLAS).

As systems grow larger, collective operations on the whole set of processes might not be feasible. Even though, many collective operations scale logarithmically with the number of processes for small input sizes, frequent communication can inhibit scalability. Thus, algorithm-design needs to address this issue and localized communications (e.g., nearest neighbor) become most important. Nevertheless, most algorithms are still written in a bulk synchronous model [17] with iterative communication and computation phases and the computation phases of many such applications communicate within a fixed neighborhood (e.g., each process has four neighbors in a two-dimensional five-point stencil computation). Such neighbor exchanges can be viewed as a localized (or *sparse*) collective group communication and optimized with similar principles as traditional collective operations [10]. The addition of a set of calls to support such a communications is considered for MPI-3.0. The communication topology of such sparse collective communications is expressed by the user at runtime and their nonblocking variants suffer from similar progression issues as traditional (we call them *dense*) collective operations.

1.1 Related Work

Several communication systems offer direct (offloaded) hardware support for some MPI collective operations [1,14]. However, such implementations often fail to support the full spectrum of collective operations and cannot express user-defined sparse collectives.

Several works propose to offload an abstract definition of a collective operation into the communication layer (e.g., a network interface card). Portals 4 [15] specifies triggered operations where new messages can be sent based on arriving messages. InfiniBand ConnectX-2 [4] specifies chained Queue Pair operations which can trigger new messages inside the HCA. GOAL allows to specify communication schedules as complete dependency graphs that can be downloaded into the communication layer [9]. All offload techniques allow nearly fully asynchronous execution of nonblocking collective operations with minimal impact on the running application. Akihiro and Ishikawa show a possible design for kernel-level asynchronous operations in [13].

Open-MX [3] offers fast point-to-point communication for Ethernet networks. It is similar to ESP (cf. Section 3.2) in that it uses the Linux skb mechanism to transmit data. However, large parts of the protocol (for example reliability) are handled in user-space. Thus, it is not possible to use it for reliable communication from kernel-space yet.

In this work, we discuss a possible implementation of the flexible Group Operation Assembly Language (GOAL) framework in a general purpose operating system. GOAL allows to express arbitrary communication patterns and dependencies. The operating system acts as the resource broker on each end-node, it can immediately react to incoming messages (interrupts) from the hardware and progress the collective communication and thus solve the progression issue. In Section 2.1, we will discuss optimized design options for collective operation schedules, kernel-level execution limitations, and an extension for performance profiling of the kernel layer. In Section 3 we discuss our experimental design of the kernel-level in detail. Results are presented in Section 4 followed by a discussion of issues in our design and conclusions.

2 Expressing Collective Operations

GOAL allows to specify communication as a local dependency graph on each process [9]. The basic set of supported vertex types are sends, receives, and local operations. Dependencies (edges) can be added to enforce a certain execution order (i.e., an edge A→B means that operation A needs to complete before operation B is started). The matching send/receive statements across processes form a global communication graph that can be transformed during a compilation phase. GOAL allows the specification and transparent optimization of complex communication patterns.

Lower-level APIs, such as ConnectX-2 or Portals 4 would act as a concrete machine language, something that abstract GOAL graphs could be compiled into. However, both interfaces are only available on certain hardware platforms. In this work, we define a scheme which enables the execution of GOAL graphs within an operating system on standard hardware, such as Ethernet.

2.1 The GOAL interpreter

The task of the GOAL interpreter is to take an optimized representation of the dependency graph and execute the primitive operations which are defined by it. Each primitive operation should be executed as early as possible but without violating the specified dependencies. GOAL graphs are serialized in traversal order and are stored as a cache-friendly binary format called *schedule*. In our implementation[3], the binary schedule is stored in the format described in Figure 1.

This representation has the advantage that the whole dependency graph is stored in a contiguous memory block. This enables fast copying of the graph when

[3] http://www.tu-chemnitz.de/informatik/RA/dw/doku.php?id=en:espgoal:study

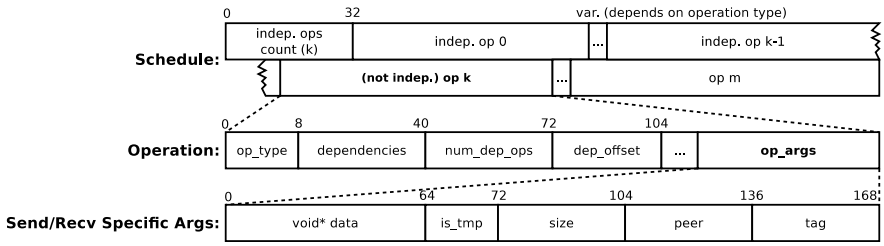


Fig. 1. Schedule Binary Format

transferring it from userspace to kernelland. If there is a dependency between the two operations u and v, which would be represented as edge $u \rightarrow v$ in the dependency graph, the dependency counter (*dependencies*) of the operation v will be at least one. The offset of the operation v will be listed in the adjacency list (*num_dep_ops, dep_offset*$_i$) of u.

When the interpreter starts to execute a schedule the *indep_ops_count* operations that have no incoming dependencies are executed first. Because each operation has a different set of arguments, the type of operation is encoded in an eight bit value at the beginning of each operation, so that the interpreter knows how much data has to be read. The *dependencies* value specifies how many incoming edges this operation has. When this counter reaches zero the operation will be executed. The scheduler is notified by the underlying network protocol whenever an operation is finished. When the scheduler starts an operation it's offset in the schedule binary is passed to network protocol layer. This address is also present in the information the scheduler gets upon completion. The adjacency list *num_dep_ops, dep_offset*$_i$ of the finished operation will be traversed. It contains the offsets of all operations that depend on this (now finished) operation. For each operation in that list the scheduler will decrease the *dependencies* counter by one. If such a counter reaches zero, the corresponding operation is executed.

GOAL supports three types of primitive operations: send, receive and local operations. Each of those operations either operates on a single contiguous block of data or on scatter/gather lists. Send and receive operations are non-blocking. An operation completes if all specified buffers can be read and modified. That implies that a send operation can be finished as soon as the data has been copied into a temporary buffer in the case of eager send. The schedule execution is non-blocking, thus, all send and receive operations are implicitly nonblocking. Local operations are predefined arithmetic (add, sub, mult, div, max, min) and binary operations (and, or, xor) on all signed, unsigned and floating point datatypes from one to 64 bit width, a copy operation to copy data between local buffers, as well as a timing operation which records a timestamp at the time it is executed by the GOAL scheduler.

2.2 User vs. Kernel Level Design

The GOAL API allows the user to specify arbitrary dependency graphs. Each node in such a graph represents a single send, receive or local calculation operation. Therefore nodes can be created by the user by calling, for example, `GOAL_Send()` or `GOAL_Recv()`. For each input or output buffer that is given to these functions, there is a corresponding argument which can be either GOAL_USERSPACE or GOAL_SCRATCHPAD. The reason for this is that schedules can be defined in a different place than they are executed. If one would write a function that creates a tree based gather schedule with GOAL, this function would have to allocate a temporary buffer. But this function can not contain the corresponding call to free(), unless the schedule is also executed, waited on, and destroyed in that function — which would make it impossible to overlap the communication with computation. Therefore GOAL has a primitive memory management functionality. For each schedule, the user specifies how much temporary space is needed. GOAL will allocate such a *scratchpad buffer* before it starts to execute the schedule. If the user wants to reference data in the scratchpad buffer, he can do this by passing the byte offset (relative to the start of the scratchpad) and set the memory type argument to GOAL_SCRATCHPAD.

Edges $t \rightarrow h$ in the dependency graph can be created with the function `GOAL_Requires()`. After the graph is complete it can be compiled into the binary representation using `GOAL_Compile()` and run with `GOAL_Run`. The functions `GOAL_Test()` and `GOAL_Wait()` can be used to test and wait for completion of the handle returned by `GOAL_Run`. The GOAL API is implemented as a userspace library, while the actual interpreter is a kernel module. The GOAL_Run() function will hand over the binary schedule to the kernel module by doing an ioctl(). The complete control flow is depicted in Figure 2.

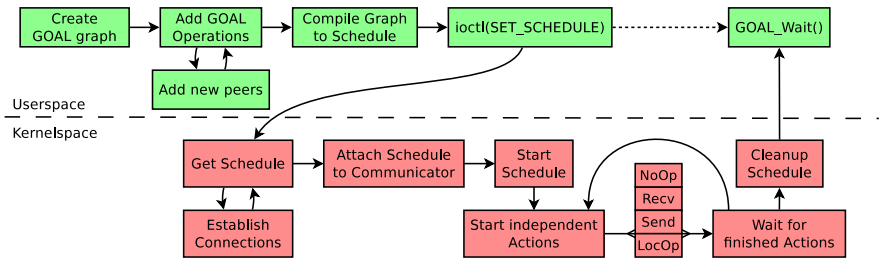


Fig. 2. ESPGOAL Control Flow

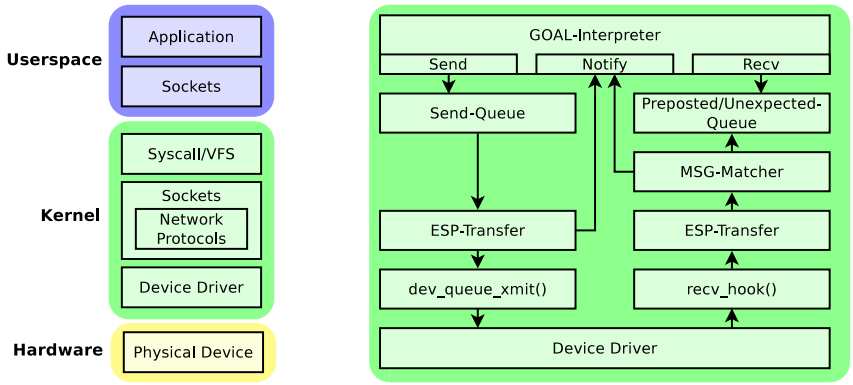
In our particular example implementation, we use the kernel-based Ethernet Streaming Protocol (ESP) [8], but we remark that any kernel-level communication mechanism will suffice. The ESP network protocol uses MAC addresses and device ids to identify peers. We collect this information during the definition

phase of the schedule: If the user adds a send operation to rank 7 to a dependency graph on rank 8 we use `MPI_Isend/Irecv` to exchange the MAC addresses between both peers. To keep the amount of out of band communication low, we cache that information in userspace. During `GOAL_Run` we pass the schedule as well as the list of all MAC addresses and device ids of the peers we will communicate with in that schedule to the GOAL interpreter. The interpreter will update the peer list for the active communicator by opening new connections (if they don't exist yet) before the schedule is started. Upon completion, the GOAL interpreter will change a memory location in the process address space, so the `GOAL_Wait()/-Test()` functions can poll that value to gather information about the status of a schedule in progress.

3 Integration into the Operating System

3.1 Anatomy of the Linux Kernel Network Stack

The Linux kernel network stack consists of multiple layers, each tries to provide a different level of abstraction. The Linux network stack is shown in Figure 3(a).



(a) Linux Kernel Network Stack (b) GOAL Network Stack

Fig. 3. Comparison of the default Linux with the GOAL network stack

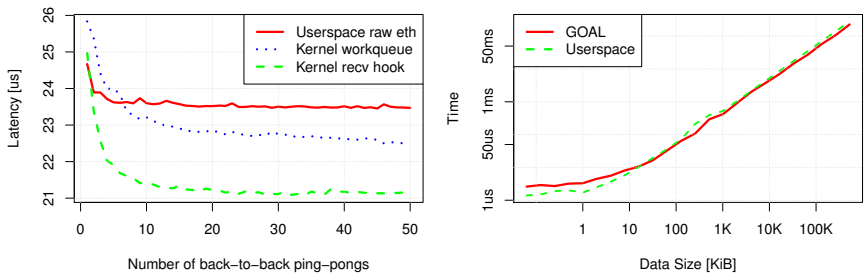
A network interface card typically has a hardware buffer to temporarily store incoming network packets. When a new packet arrives the Linux kernel is notified with an interrupt from the network card. The device driver retrieves the newly received packets and stores them in so called socket buffers (*skbs*).

Incoming packets (skbs) are handled by "receive hook" functions, registered with `dev_add_pack()` in Linux. Full skbs (including all Ethernet packet data, such as destination, ethertype, etc.) are sent with `dev_queue_xmit()`. This is the lowest layer of abstraction which is offered by the Linux kernel to send and receive data in a device independent manner. Our implementation uses this

interface to the driver layer inside the kernel. The benefit of this approach is that the functions mentioned above do not sleep and therefore they can be called in an irqhandler or tasklet.

Another possibility how to implement network communication in the Linux kernel is to use the kernel socket API. Utilizing kernel sockets is very similar to userspace socket programming, however, in the kernel one has to employ mutual exclusion strategies to prevent race conditions. Most network protocols supported by the Linux kernel, such as TCP are implemented with the kernel socket API. One disadvantage of the socket API is that certain functions, for example, sending data via `kernel_sendmsg()` can not be performed in an interrupt handler or tasklet. If such functionality is required it has to be implemented in a separate kernel thread or a workqueue element. Thus, we used workqueues to implement GOAL over ESP (ESPGOAL). Other network protocols such as TCP do not have to use workqueues or an extra kernel thread as the problematic socket API function which might sleep are usually called from userspace.

This raises the question if the scheduling overhead implied by using workqueues has a negative impact on ESPGOALs performance, compared to the other possible approaches to send and receive data in the kernel. If the overhead required to start a new work item in a workqueue is significant it is desirable to have an upper bound on its performance impact so that we can decide if it would be useful to exchange the ESP protocol with something that directly utilizes the functions offered by the device driver to send and receive data in future work.



(a) Comparison of different ways to communicate over Ethernet

(b) Comparison of floating point performance in user- and kernelspace

Fig. 4. Microbenchmark Results

We implemented a microbenchmark to assess the overheads affiliated with the different choices. Our benchmark consists of three different implementations of a pingpong scheme, one using the raw socket API from userspace and the other two run in kernelspace. The benchmark performs multiple round-trips for each measurement to amortize the startup overheads. Figure 4(a) shows the benchmark results for two CHiC nodes, see Section 4.1. We observe that the

overhead for inserting and scheduling a workqueue element adds about 1.6 μs of latency to each transmission.

3.2 The Ethernet Streaming Protocol

The Ethernet Streaming Protocol (*ESP*) [8] is a connection-oriented, port multiplexed and reliable protocol on top of Ethernet with optimized congestion control for static, switched networks. It can be used through standard sockets from kernel- and userspace. This makes it ideal to utilize it in a kernel based version of the GOAL interpreter. As mentioned before, Open-MX cannot be used from inside the kernel directly at this stage.

The ESP protocol is transfer based. There are special flags to signal the start (TXS) and finish (TXF) of a transfer. After the initial TXS, the receiver requests more data, until he receives a TXF packet. This has the advantage that the receiver handles flow control and adapts it to the number of streams. The GOAL scheduler is only invoked for packets with the TXF flag set.

3.3 Asynchronous Progression

The GOAL interpreter is activated (run as a kernel workqueue element) in two conditions: Either ESP received a packet that had the TXF flag set or ESP could not receive more data into an skb because there is not enough memory available (memory pressure). The TXF flag indicates that a transfer is completed and the GOAL scheduler will try to match the received message against the preposted receive queue or put the message in the unexpected receive queue. If the message matched, the scheduler will mark the corresponding node in the schedule as completed and decrease the dependency counters of all nodes (operations) that depend on it. The scheduler will then immediately start all operations where the dependency counter reached zero.

If the scheduler was called because of memory pressure, it will also try to process messages that are not completely transferred yet. The header that is needed to perform message matching is transferred in the first 28 bytes of each transfer so the interpreter can perform message matching and partially copy the payload to the final destination for every socket that contains at least 28 bytes of data and holds a message that belongs to a preposted receive.

A workqueue item is implemented as a function pointer that will be executed in a special kernel thread. A modern Linux kernel (i.e., 2.6.36) will run one workqueue execution thread per core and decide which workqueue item to run based on a number of flags that can be set when allocating the workqueue structure. Currently the GOAL interpreter is run as a high-priority workload, which means that available workqueue items are to be scheduled by the kernel as soon as possible. Also our workqueue items are marked as unbound, which means they can be run on any core available, to maximize the chance that they are executed immediately.

3.4 Performing Reduction Operations in Kernel Space

Performing floating point operations inside the kernel space is supported by the macros `kernel_fpu_begin()/end()`, which save and recover the FPU state and disables preemption.

In order to assess the performance impact of a kernel-based reduction, we implemented a simple benchmark that computes the maxima of two 32 bit floating point vectors. We ran the benchmark in userspace and in the kernel with a GOAL local operation. We excluded the schedule startup overhead from all GOAL measurements. As shown in Figure 4(b) the kernel implementation is slightly slower than the userspace implementation for small datasizes, but outperforms the userspace implementation for larger vectors.

4 Benchmark Results

We implemented several collective operations with GOAL on top of ESP. It was shown that test-based schemes achieve reasonable overlap for large messages [7]. However, overlapping small-message communications remains hard due to the high ratio between control overhead and message-sending. Thus, we focus especially on small-message operations because they are most important at large-scale and are hardest to overlap. We implemented several optimized collective algorithms for small-message collectives. For all-to-all communication, we used the scheme proposed by Bruck [2] and for barrier and allreduce we implemented the well-known dissemination algorithm. Both schemes use $\log_2(P)$ stages in P processes and have a relatively complex dependency structure.

4.1 Experimental Setting

We conducted all experiments on the CHiC Cluster System at the University of Technology Chemnitz. CHiC consists of 530 compute nodes with two Opteron 2218 Dual-Core 2.6 GHz CPUs running Linux. Each node is equipped with two Tigon3 BCM95704A6 rev 2100 network cards which are connected to an 48 port Gigabit Ethernet Switch (SMC 8848M). We used an MTU of 1500. The NIC used supports interrupt coalescing. With the default coalescing parameters the latency was very high. If we disabled interrupt coalescing completely our systems became unstable. Therefore we optimized the interrupt coalescing settings with a genetic algorithm and the omx-pingpong tool included in the Open-MX distribution. The coalescing settings used were:

rx-usecs	rx-frames	rx-usecs-irq	rx-frames-irq	tx-usecs	tx-frames	tx-usecs-irq	tx-frames-irq
1	1	996	95	32	94	724	128

We used Open MPI 1.4.2 and Open-MX 1.3.4 in all experiments. We compare LibNBC over MPI with the different transports: TCP/IP (TCP), Open-MX (OMX), and ESP (used from user-level) with out kernel-based ESPGOAL implementation. The latency of a blocking execution (initiation call immediately followed by a wait) of the GOAL nonblocking collective operations and the LibNBC nonblocking collective operations is very similar. Figure 5 shows barrier as an example. It can be seen that the latency of the ESPGOAL Barrier is between the

Open MPI implementation with the TCP and MX BTL. This can be attributed to the well tuned Open-MX implementation, which uses a lot of optimizations that have not been done in ESP. Also note that minimizing latency for blocking collectives was not the goal for this work — we just want to ensure that our implementation is not substantially slower, which could invalidate our overlap results shown in Section 4.2. If an implementation spends a lot of time waiting for IO it would be easy to overlap the collective and the CPU overhead would be low.

4.2 Asynchronous Progress and Overlap

We now analyze the ability of ESPGOAL to asynchronously progress messages. For this, we use NBCBench [7] without any explicit progression. NBCBench uses a work-loop, which is calibrated at the beginning of the benchmark, to determine the overlap. This means that all interruptions by the kernel will "steal" time from the work loop and show up as overhead, see [5] for a detailed description.

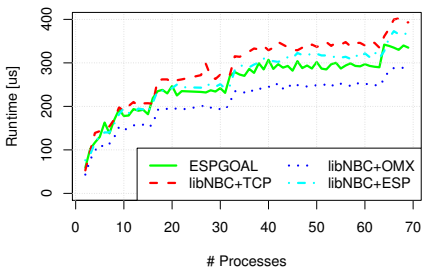
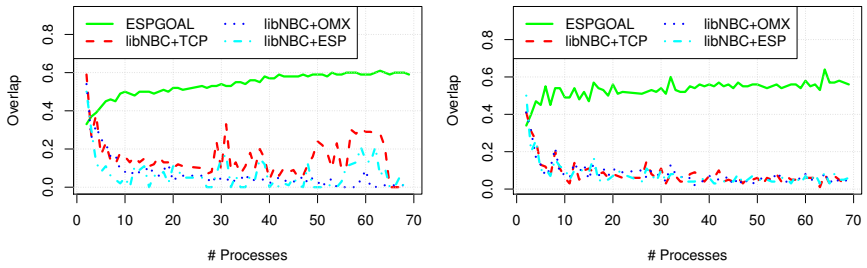


Fig. 5. Barrier Latency

NBCBench reports the share of the communication that can be overlapped with computation, a number between 0 and 1 (higher is better). Figure 4.2 shows the results for all-to-all of size 8 bytes per process and barrier. As expected,



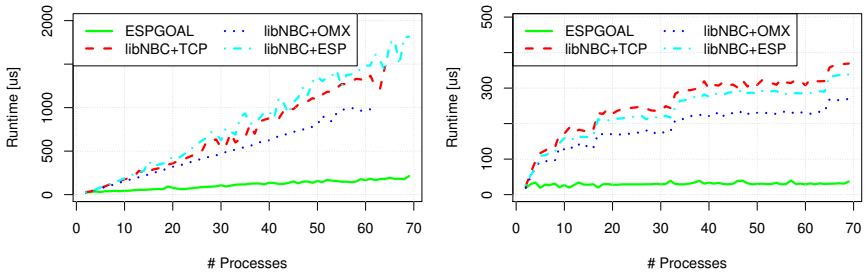
(a) All-to-all (b) Barrier

Fig. 6. NBCBench Overlap Benchmarks

we see high overlap with the ESPGOAL implementation while the unprogressed nonblocking collective operations exhibit very low overlap due to missing asynchronous progression (all but the first stage of the algorithm will be performed during the wait call).

4.3 CPU Overheads

In this section, we assess the absolute CPU overheads, i.e., the absolute non-overlappable time of the communication. We showed that asynchronous progression works well for the investigated operation. Reducing the absolute CPU overhead of the operations is most important to "free" the CPU for the user application. Figure 4.3 shows the absolute CPU overhead for each configuration. ESPGOAL causes a significantly lower CPU overhead than LibNBC in all configurations.



(a) All-to-all (b) Barrier

Fig. 7. NBCBench CPU Overhead

5 Conclusions and Future Work

We implemented a dependency driven communication framework that offers true asynchronous progress without an extra progression thread. We defined an API to use such a framework that supports simple sends and receives, vector sends and receives, and local operations. Our framework shows significant improvements in terms of host overhead over existing userland implementations of non-blocking collectives. Our work shows that it is possible to implement dependency driven communication schemes as a Linux kernel module without placing constraints on the user. For example our GOAL scheduler does not require the user to pin the memory used for communication buffers.

In future work this implementation should be tuned further so that it can compete with state of the art low overhead Ethernet protocols such as Open-MX. One possible way to tune ESPGOAL even further would be to replace the ESP protocol with another low overhead Ethernet protocol that shows better performance in point to point latency benchmarks, for example it could be investigated if ESP can be replaced with the kernel part of Open-MX.

Another interesting optimization would be the use of the memory subsystem on multi-core nodes. An optimized GOAL implementation could directly push or pull the data into other processes memory similarly to kernel-level zero copy mechanisms such as KNEM [11].

Acknowledgments This work was supported in part by the DOE Office of Science, Advanced Scientific Computing Research X-Stack Program.

References

1. Almasi, G., et al.: Optimization of MPI collective communication on BlueGene/L systems. In: ICS '05. pp. 253–262. ACM Press, New York, NY, USA (2005)
2. Bruck, J., Ho, C.T., Upfal, E., Kipnis, S., Weathersby, D.: Efficient algorithms for all-to-all communications in multiport message-passing systems. IEEE Trans. Parallel Distrib. Syst. 8, 1143–1156 (November 1997)
3. Goglin, B.: High-Performance Message Passing over generic Ethernet Hardware with Open-MX. Elsevier Journal of Parallel Computing (PARCO) 37(2), 85–100 (Feb 2011)
4. Graham, R.L., et al.: ConnectX-2 InfiniBand management queues: First investigation of the new support for network offloaded collective operations. Cluster Computing and the Grid, IEEE International Symposium on pp. 53–62 (2010)
5. Hoefler, T., Lumsdaine, A.: Message Progression in Parallel Computing - To Thread or not to Thread? In: Proceedings of the 2008 IEEE International Conference on Cluster Computing. IEEE Computer Society (Oct 2008)
6. Hoefler, T., Lumsdaine, A.: Optimizing non-blocking Collective Operations for InfiniBand. In: Proceedings of the 22nd IEEE International Parallel & Distributed Processing Symposium (IPDPS) (04 2008)
7. Hoefler, T., Lumsdaine, A., Rehm, W.: Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI. In: Proc. of the SC 2007. IEEE Computer Society/ACM (Nov 2007)
8. Hoefler, T., Reinhardt, M., Mietke, F., Mehlan, T., Rehm, W.: Low Overhead Ethernet Communication for Open MPI on Linux Clusters CSR-06(06) (Jul 2006)
9. Hoefler, T., Siebert, C., Lumsdaine, A.: Group Operation Assembly Language - A Flexible Way to Express Collective Communication. In: ICPP-2009 - The 38th International Conference on Parallel Processing. IEEE (Sep 2009)
10. Hoefler, T., Traeff, J.L.: Sparse collective operations for MPI. In: Proceedings of the 23rd IEEE International Parallel & Distributed Processing Symposium (IPDPS), HIPS Workshop (May 2009)
11. Moreaud, S., Goglin, B., Goodell, D., Namyst, R.: Optimizing MPI Communication within large Multicore nodes with Kernel assistance. In: CAC 2010. IEEE Computer Society Press, Atlanta, GA (Apr 2010)
12. MPI Forum: MPI: A Message-Passing Interface Standard. Version 2.2 (June 23rd 2009), www.mpi-forum.org
13. Nomura, A., Ishikawa, Y.: Design of kernel-level asynchronous collective communication. In: Proceedings of the EuroMPI 2010. pp. 92–101. EuroMPI'10, Springer-Verlag, Berlin, Heidelberg (2010)
14. Petrini, F., Frachtenberg, E., Hoisie, A., Coll, S.: Performance Evaluation of the Quadrics Interconnection Network. Journal of Cluster Computing 6(2), 125–142 (April 2003)
15. Riesen, R.E., Pedretti, K.T., Brightwell, R., Barrett, B.W., Underwood, K.D., Hudson, T.B., Maccabe, A.B.: The Portals 4.0 message passing interface (April 2008), Sandia National Laboratories, Tech. Rep. SAND2008-2639, April 2008
16. Sanders, P., Speck, J., Träff, J.L.: Two-tree algorithms for full bandwidth broadcast, reduction and scan. Parallel Comput. 35, 581–594 (December 2009)
17. Valiant, L.G.: A bridging model for parallel computation. Commun. ACM 33(8), 103–111 (1990)