Kernel-Assisted and Topology-Aware MPI Collective Communications on Multi-core/Many-core Platforms

Teng Ma[a], George Bosilca[a], Aurelien Bouteiller[a], Jack J. Dongarra[a,b,c],

[a]*University of Tennessee, Knoxville, TN, USA*
[b]*Oak Ridge National Laboratory, Oak Ridge, TN, USA*
[c]*University of Manchester, Manchester, UK*

Abstract

Multicore Clusters, which have become the most prominent form of High Performance Computing (HPC) systems, challenge the performance of MPI applications with non uniform memory accesses and shared cache hierarchies. Recent advances in MPI collective communications have alleviated the performance issue exposed by deep memory hierarchies by carefully considering the mapping between the collective topology and the hardware topologies, as well as the use of single-copy kernel assisted mechanisms. However, on distributed environments, a single level approach cannot encompass the extreme variations not only in bandwidth and latency capabilities, but also in the capability to support duplex communications or operate multiple concurrent copies. This calls for a collaborative approach between multiple layers of collective algorithms, dedicated to extracting the maximum degree of parallelism from the collective algorithm by consolidating the intra- and inter- node communications.

In this work, we present HierKNEM, a kernel-assisted topology-aware collective framework, and the mechanisms deployed by this framework to orchestrate the collaboration between multiple layers of collective algorithms. The resulting scheme maximize the overlap of intra- and inter- node communications. We demonstrated experimentally, by considering three of the most used collective operations (Broadcast, Allgather and Reduction), that 1) this approach is immune to modifications of the underlying process-core binding; 2) it outperforms state-of-art MPI libraries (Open MPI, MPICH2 and MVAPICH2) demonstrating up to a 30x speedup for synthetic benchmarks, and up to a 3x acceleration for a parallel graph application (ASP); 3) it furthermore demonstrates a linear speedup with the increase of the number of cores per compute node, a paramount requirement for scalability on future many-core hardware.

Keywords:

MPI, multicore, cluster, HPC, collective communication, hierarchical

1. Introduction

While the insatiable demand for increasing computing power from the domain sciences motivates the deployment of powerful High Performance Computing (HPC) systems, thermal and power consumption concerns have curbed the growth of both compute node count and processor frequency. As an alternate source of processing power, multicore clusters have become the most prominent form of HPC systems, and exhibit a rapid increase in the number of cores per compute node. The top ranking machine in the August 2012 Top500 list, the LLNL Sequoia computer, uses more than one and a half million cores[1]. Processors with 8 to 12 cores are the norm today, and it is not uncommon to have such processors deployed in multiple socket boards featuring 8 to 96 cores, with network-style interconnection between caches or to the memory banks (e.g. Intel QPI or AMD Hyper-transport). Unfortunately, this new hardware trend challenges the assumptions made by most current HPC programming models, directly threatening the performance efficiency of the machines. Namely, within compute nodes, non uniform memory accesses (NUMA), memory and shared cache hierarchies, weaken the assumptions of regular load balance and uniform link bandwidth and latency.

In the era of the single-core cluster, the Message Passing Interface (MPI) standard has enjoyed a wide adoption in the HPC community, thanks to two key features: implementations could provide the highest level of performance while maintaining the application's portability. With respect to portability, not only an MPI code compile on different machines, but it also exhibits an excellent efficiency, because network topologies and collective patterns of communications are accounted for by the MPI library rather than the application itself. With the introduction of multicore compute nodes, both of these features have been threatened in MPI, most implementations treating multicore compute nodes as mere SMP units and ignoring their internal hierarchies. To alleviate these issues, some attempts have been made to use hybrid programming models, retaining MPI between computer nodes and a thread-centric approach (pthreads, OpenMP, TBB, ...) between cores. The suitability of this approach is questionable, with research showing a similar number of applications that benefited from the approach compared with failures to reach any performance improvement. Moreover, from the productivity point of view several drawbacks are evident: it imposes a significant complexity on programmers, renders explicit the management of hierarchies which defeats performance portability,

[1]http://top500.org/lists/2012/06

and imposes major rewrite of legacy applications. We believe that the MPI standard is a competitive proposition for harnessing the power of multicore clusters, should the implementations of the standard use the proper techniques to account for core and memory link properties, especially in the area of collective communications.

Indeed, recent advances in MPI collective communications have already demonstrated that the performance issues incurred by multicore memory hierarchies can be solved on shared memory multicore computer nodes. The careful mapping between the collective topology and the core distance [1], and the use of single-copy kernel assisted mechanisms deep inside the collective algorithms [2] have been proven to greatly increase the shared memory communication efficiency. However, on distributed memory machines, like clusters of multicores, a single approach cannot encompass the extreme variations not only in the bandwidth and latency capabilities, but also in features such as the aptitude to operate multiple concurrent copies. Efficient multicore shared memory approaches are so specific, including kernel assisted copies, that they cannot apply to network communications; on the other hand, regular network approaches fail to extract performance off shared memory links. This calls for a collaborative approach between multiple layers of collective algorithms, dedicated to managing intra and inter computer node communications.

In this paper, which is an extension of our distinguished work [3], we present how HierKNEM, a kernel-assisted topology-aware collective framework, orchestrates the collaboration between multiple layers of collective algorithms. Leaders are selected among the core-centric collective algorithm, to participate in the inter-node collective topology. Intra-node communications are managed by offloading memory copies to non-leader processes, taking advantage of the kernel-assisted single-copy approach to balance the memory copy load among available cores. The resulting scheme enables perfect overlap of intra-node communication with inter-node communications, thanks to innovative hierarchical algorithms. We demonstrate experimentally, by considering three distinct collective patterns (one-to-many, many-to-many and many-to-one), that 1) this approach is immune to modifications of the underlying process-core binding; 2) it outperforms state-of-art MPI libraries (Open MPI, MPICH2 and MVAPICH2) demonstrating up to a 30x speedup for messages between 8KB and 256KB in synthetic benchmarks, and up to 3x speedup for a parallel graph application (ASP); 3) it demonstrates a linear speedup with the increase of the number of cores per computer node, a paramount requirement for scalability on future many-core hardware.

The rest of this paper is organized as follows: Section 2 introduces related work on current efforts to optimize MPI collective communication on multi-core clusters and the application of kernel-assisted approach into MPI libraries. Then, Section 3 describes the framework for kernel-assisted hierarchical collective communications on clusters of multicore and details three collective algorithms: one-to-all (Broadcast), all-to-one (Reduce), all-to-all (Allgather), and their corresponding implementations in a new Open MPI collective component: HierKNEM. These algorithms are experimentally compared with state-of-the-art MPI implementations to assess the benefits of the hierarchical approach in Section 4. Finally, Section 5 concludes the paper with a discussion of the results.

2. Related Work

The legacy approach to implement collective communication is to adopt one of many different communication topologies (linear, chain, split binary tree, binomial tree, etc.) [4]. These basic approaches can be refined by enabling parallel treatment through message pipelining, a technique in which large messages are split into smaller chunks to maximize steady-state bandwidth. Furthermore, a runtime decision module can be used to select the best algorithm and tuning parameters, according to message size, communicator size, and other input variables [5]. The Tuned collective module, in Open MPI, is iconic of such an approach; while MPICH2 and other MPI libraries feature a similar approach restricted at the compilation time. Unfortunately, while mapping communications to specific network topologies yield drastic performance improvement on single-core architectures, the increase in the number of cores renders this problem more complex, requiring additional parameters that could reflect the runtime processes' topology in view of physical distances. To further exacerbate this issue, intelligent process placements, used as a bridge between applications and MPI libraries, e.g. MPIPP [6], have the tendency to exacerbate the amount of point-to-point communications between specific processes as they consider the collective communication as a simple set of point-to-point communications. This triggers two issues: on one hand it imposes the preselected topology for all subsequent runs and computes the neighborhood relation-ship based on a communication pattern likely to change with the execution environment. This irregular mapping leads to a further mismatch between collective topologies and underneath hardware [1].

The conventional effort toward adapting collective communications to hierarchical hardware topologies is leader-based hierarchical algorithms [7, 8, 9, 10, 12, 13]. Earlier attempts toward hierarchical approaches on collective communication on clusters of SMPs [7] or Grids [14] focused on reducing the amount of data or the number of messages crossing the low bandwidth or high latency links, respectively. Combining with the SMP-aware method, leader-based hierarchical algorithms were widely applied to all collective communications patterns, e.g., MPI on Quadrics networks [8], Open MPI's Hierarch collectives, or MVAPICH2 on Infiniband networks [10, 12, 13]. In these SMP-aware methods, multicore compute nodes are often treated as shared memory nodes without internal hierarchy. As a consequence, the layered collective components that handle inter and intra-node communications do not cooperate tightly, leading to suboptimal pipelining and sometimes contradictory tuning choices. This results in another difference with our proposed work: the intra-node communication is mainly implemented by a copy-in/copy-out approach using a shared memory segment.

The copy-in/copy-out approach requires two memory copies for each message, greatly wasting memory bandwidth and CPU cycles. When applying this approach into leader-based hierarchical algorithms, leader processes are heavily involved in intra-node data movement [2], resulting in serializing the inter- and intra-node communications. Most of the intra-node communication overhead accumulates and results in a significant overhead that cannot benefit from overlap by inter-node communications. Obviously, such overhead is bound to increase with the number of cores; the copy-in/copy-out at the leader process has to be sequentially executed once for each of the processes participating in the collective communication.

To reduce the overhead from double memory copies in the copy-in/copy-out approach, one-sided single-copy methods have been proposed. SMARTMAP [15, 16] is an effort to make use of a simple page table management in catamount systems to implement single-copy intra-node communication. Another direction is the kernel-assisted approach such as LiMIC [17] in MVAPICH2 or KNEM [18] in MPICH2 and Open MPI. This kernel-assisted approach has been widely used to speed up large messages' point-to-point communication on shared memory machines [18]. Furthermore, an intra-node collective communication component, KNEM collective [2], is implemented into Open MPI, based directly on the KNEM copy and not implemented over KNEM-enabled point-to-point communication. The KNEM collective harnesses KNEM's single-copy and direction control techniques to offload memory copies to non-root processes, providing a significant performance boost [2]. Also, efforts have been made to take into account NUMA hierarchies in the process placement and to optimize intra-node collective algorithms to include architectural features [1]. However, these projects focus solely on improving communications within a single shared memory multicore compute node. The aspects regarding cooperation of these complex algorithms with the inter-node layer of the collective communication have not been addressed so far. There is an obvious need to develop algorithms that encompass both layers and account for all particularities and varieties of hardware.

3. Collective Algorithm Composition

3.1. Framework

As hinted previously, most existing approaches to develop hierarchical collective communications are based on a multi-level approach where the top level represents the largest area network, and each subsequent level is for a smaller area network. While they provide interesting performance compared with single-level approaches, they do not benefit from the entire overlapping potential of collective algorithms, as the transition processes (i.e. processes that are leafs in one level and become root on the next), are step by step blocked in a collective for a particular level. What has been missing in these attempts at providing hierarchical collective operations on clusters of multicore system was the ability to express a multi-level algorithm with a very tight level of interoperability between the levels. In the present effort, we want to enable an unprecedented level of integration between different algorithms, by dissolving the boundaries between the levels, and allowing the transition processes to overlap collective between the inter and intra levels.

From a technical point of view, in most of the hierarchical approaches including ours, collective communication is divided between inter- and intra-node communication. Each process has an intra-node communicator encompassing all processes hosted on the same physical compute node. Among these local processes, a leader process is selected to represent the compute node in the inter-node layer. All non-leader processes only communicate with the local leader process and then messages are forwarded by the leader process to remote leader processes on remote compute nodes. The advantage is that the messages carried through expensive inter-node links are explicit, giving leverage for the algorithm composition to minimize cross-traffic volume. From a technical standpoint, what differentiates our approach compared to previous attempts is the level of integration between the layers of the hierarchy, allowing multiple algorithms to coordinate their pipelining strategies at a very low level.

One major challenge for multi-level algorithms is to coordinate around the usage of common resources. In this particular instance, one should pay attention to the load imposed on the memory bus. This load is two-folds: on one side sending/receiving data over the network translates into moving data across the PCI bus from the memory bus. On the other side, moving data inside the compute node generates memory bus traffic, and therefore collides with the network transfer (the data in Figure 4 highlight this fact). Therefore, special care has been taken to minimize the number of memory transfers at the inter-node level. The approach chosen in this framework is to base all intra-node memory transfers on the KNEM collective components, described in [2]. KNEM's offloading capability is naturally matched up with leader-based hierarchical collectives: workloads of memory copies can be offloaded onto non-leader processes. Non-leader processes can simultaneously read or write leader processes' memory through KNEM primitives; meanwhile, leader processes can dedicate themselves to inter-node forwarding, without sequentialization experienced by less integrated hierarchical approaches. For communication strictly within large NUMA compute nodes, different approaches yield varying performance. Our new hierarchical algorithms leverage the knowledge accumulated on a single compute node [2] to design sound algorithm compositions that can cope with a large number of cores within compute nodes. The experimental section demonstrates how well these approaches collaborate with another layer.

In this new context, we provide three improved versions of the most used collective communications: a one-to-many (Broadcast), a many-to-many (Allgather) and a many-to-one (Reduce).

3.2. Broadcast

Let's assume the intra-node communicator for each compute node is lcomm, the inter-node communicator for leader processes is llcomm, and process rank is P. Suppose a two-level broadcast algorithm, using a spanning tree-based approach for the inter-node level and a linear approach for the intra-node level. Our HierKNEM broadcast algorithm is adaptive enough to handle special cases, e.g. when all processes are allocated on a single compute node, our broadcast is transformed into a linear algorithm identical to the KNEM one; when each compute node has a single process in the communicator, our HierKNEM broadcast is automatically morphed into a spanning tree broadcast identical to the inter-node level.

Algorithm 1 presents the pseudo-code of the HierKNEM Broadcast. In order to save space, we trimmed the pseudo-code handling the special cases mentioned above and presented the algorithm processing a general case: each compute node has more than one process bound to different cores and all leader processes are organized into a spanning tree with more than two levels: a root node, intermediate nodes, and leaf nodes. At first, each leader process registers 'rbuf' into KNEM device and gets a 'cookie' back at step 2. This cookie is a unique identifier to point to an entry recording rbuf's physical memory address, and any other process in the compute node having this identifier can access (based on the granted right) this registered buffer via the KNEM module. This cookie will then be broadcasted to all non-leader processes on the same compute node (step 3 and 33). Afterward the message is divided into equal-sized fragments and forwarded in a pipelining fashion along the spanning tree composed of all leader processes (between step 4 and 29). In this particular context, father and children mentioned in Algorithm 1 refer to the process up and down the spanning tree from the current process P. For intermediate and leaf nodes in the spanning tree, once the leader processes receive

```
Input: MPI_Bcast(void *rubf, int count, MPI_Datatype dtype, int root, MPI_Comm comm)
if P is leader process then
    Register rbuf into KNEM device and get a cookie;
    Broadcast this cookie to all non-leader processes on the same compute node;
    if P is root process then
        for $i \leftarrow 1$ to seg_num do
            Isend segment i to its children in spanning tree;
            Wait for all Isend;
        end
    else if P is a leader process in an intermediate compute node then
        Post Irecv for 1st segment from its father;
        for $i \leftarrow 1$ to seg_num-1 do
            Post Irecv for next segment(segment i+1) from its father;
            Wait for previous Irecv(segment i);
            Isend received segment(segment i) to its children;
            Barrier in the lcomm communicator;
            Wait for all Isend;
        end
        if $i \equiv$ seg_num then
            Wait for previous Irecv(last segment);
            Isend last segment to its children;
            Barrier in the lcomm communicator;
            Wait for Isend;
        end
    else
        for $i \leftarrow 1$ to seg_num do
            Recv segment i from its father;
            Barrier in the lcomm communicator;
        end
    end
    Barrier in the lcomm communicator;
    Deregister buffer from KNEM device;
else
    Get KNEM cookie from the leader process;
    if P is on the same compute node with root process then
        Fetch the whole data from root process by KNEM;
    else
        for $i \leftarrow 1$ to seg_num do
            Barrier in the lcomm communicator;
            Fetch segment i from leader process by KNEM;
        end
        Barrier in the lcomm communicator;
    end
end
```

Algorithm 1: The HierKNEM Broadcast Algorithm.

a segment from its father node, they will notify all non-leader processes on the same compute node to fetch the segment (step 15 and 21). Upon receiving this notification at step 38, each non-leader process will fetch the segment by a KNEM *get* operation at step 39. This *get* operation is one-sided and will be offloaded to the non-leader processes. Therefore the overhead of intra-node data movement can be overlapped at the leader process with the forwarding between leader processes on the upper level (step 12, 14, or 20).

This is the fundamental reason why our HierKNEM collective can outperform other collective components: intra-node communication is offloaded to non-leader processes and leader processes can dedicate themselves into inter-node message forwarding. In an ideal situation, the intra-node communication overhead can be completely hidden from the overall execution time and the entire collective communication execution time made close to the inter-node collective execution time (the collective on the leader processes communicator). In the event of a perfect overlap, a multi-core broadcast operation can be made *number-of-compute-nodes* dependent instead of *number-of-cores* dependent.

3.3. Reduce

```
Input: MPI_Reduce(void *sbuf, void *rbuf, int count,MPI_Datatype dtype,
                  MPI_Op op, int root, MPI_Comm comm)
1  if P is the $1^{st}$ leader process then
2      for i ← 1 to seg_num do
3          Wait notification from $2^{nd}$ leader;
4          Reduction in the llcomm for segment i;
5      end
6  else if P is the $2^{nd}$ leader process then
7      for i ← 1 to seg_num do
8          Fetch segment i from $1^{st}$ leader;
9          Reduction between two leaders' segment i;
10         Reduction in the new_comm for segment i;
11         Push reduction result of segment i to $1^{st}$ leader's tmpbuf;
12         Notify $1^{st}$ leader that pushing operation is done;
13     end
14 else if non-leader processes exist inside the compute node then
15     for i ← 1 to seg_num do
16         Reduction in the new_comm for segment i;
17     end
18 end
```

Algorithm 2: The HierKNEM Reduce Algorithm.

Similarly to the Broadcast algorithm, the HierKNEM Reduce uses an inter-node communicator (llcomm) and intra-node communicator (lcomm). In addition to these two communicators, the HierKNEM Reduce creates another local communicator, a subset of the lcomm, to organize all non-leader processes on the same compute node (new_comm). This new_comm is used to isolate leader processes from the intra-node reduction. The HierKNEM Reduce is actually a double-leader algorithm: the 1^{st} leader process participate in the upper level (inter-node) reduction while the 2^{nd} leader process will be the root for an intra-node reduction on each of the new_comm communicators and responsible for updating the 1^{st} leader with the local contribution to the upper level reduction. Algorithm 2 describes the HierKNEM Reduce for a general case: each compute node has more than two processes participating in a reduction operation: one leader for the inter-node reduction and another leader for the intra-node reduction. In order to save space, we trimmed the algorithm of the handling of special cases, the internal management and distribution of KNEM registrations.

The 2^{nd} leader fetches segment i from the 1^{st} leader's sbuf by a KNEM get operation (step 8), and applies the reduction operation between their sbuf's segment i (step 9). As a root process, the 2^{nd} leader calls an intra-node reduction for segment i in the new_comm with the result from step 9. After finishing this reduction, the 2^{nd} leader will push the reduction result of segment i to the 1^{st} leader by a KNEM writing. After getting the notification from the 2^{nd} leader (step 3), the 1^{st} leader will trigger an inter-node reduction between leader processes with pushed results for segment i. The intra-node reduction for segment i+1 can be overlapped with inter-node reduction for segment i thanks to KNEM's one-sided operation and the pipelining reduction algorithm between hierarchical communicators.

3.4. Allgather

The HierKNEM provides two algorithms for Allgather: a leader-based algorithm for clusters of small compute nodes (2-6 cores per compute node) and a ring algorithm for large compute nodes. The leader-based algorithm has three steps: 1) gathering messages into leader processes; 2) exchanging data between leader processes; and 3) broadcasting data from leader processes to non-leader processes. Step 1 and 3 happen inside a compute node while step 2 exchanges data using inter-node communications. At the inter-node level (step 2), the leader processes are organized into a logical ring and each leader process communicates only with the left and right neighbors in this ring. Once leader processes get a message from step 1 or step 2, they will notify non-leader processes to fetch data by KNEM copy. Because KNEM copy in step 1 or 3 is one-sided and always offloaded onto non-leader processes, leader processes only synchronize with non-leader processes before or after non-leader processes write or read data into or from leaders. This synchronization overhead is minimal compared with the cost of intra-node data movement. As a result, the leader processes can dedicate themselves to inter-node data exchanging and steps 1-3 can be totally overlapped. The critical path of our algorithm depends on the overhead of inter-node exchanging or intra-node gather (step 1) and broadcast (step 3). When intra-node communication cost exceeds inter-node exchanging time (more cores per compute node or faster network), the leaders' memory bandwidth is overloaded by this ad-hoc memory access pattern. Thus, the overall throughput is seriously restricted by such a simple combination of Gather and Broadcast operations. So in clusters of large NUMA compute nodes, the HierKNEM Allgather adopts a ring-based algorithm to distribute data both at the inter-node and intra-node levels in order to avoid such hot-spots on leader processes. The HierKNEM Allgather ring algorithm is similar to the MPICH Allgather ring algorithm [19]: all processes are organized into a logical ring and each process receives messages only from its left neighbor and sends messages only to its right neighbor. This send and receive will be executed number of comm_size-1 times and a local memory copy will be executed at the beginning. A notable improvement over the ordinary ring algorithm, the construction of the HierKNEM's logical ring is not based on the order of MPI ranks but adheres to the physical process distance in terms of sockets and NUMA compute nodes. Thus, processes physically close are clustered together into a set. Only processes on edges between sets communicate through slow links: inter-node links or inter-socket links.

4. Experimental Evaluation

We used two clusters of the Grid5000 experimental platform: Stremi and Parapluie. The Stremi cluster features 32 compute nodes, each with two AMD Opteron 6164 HE twelve-core CPUs (24 cores per compute node). Each socket has 10 MB L3 caches and two NUMA memory nodes, and 6 cores in each socket share one NUMA memory node with 12 GB of memory (48 GB of memory per compute node). These 32 compute nodes are interconnected by Gigabit Ethernet. The Parapluie cluster is identical to Stremi, except that the 32 compute nodes are interconnected through a 20G Infiniband network.

Our HierKNEM collective is based on Open MPI version 1.5.3. We compared HierKNEM collective with the Open MPI's (1.5.3) Tuned, Hierarch collective, MPICH2 version 1.4.1 on the Ethernet cluster (Stremi) and MVAPICH2 version 1.7 on the Infiniband cluster (Parapluie). All implementations that support kernel assisted memory operations use KNEM version 0.9.6 [18] except MVAPICH2. MVAPICH2 uses LIMIC2 0.5.5 [17] as the kernel assisted memory copy module.

For intra-node communications, HierKNEM, Tuned, and Hierarch collective modules are configured to use the SM/KNEM BTL (byte transfer layer) as underneath point-to-point communication helper. SM/KNEM BTL uses KNEM copy to speed up point-to-point communication; for performance reasons, the copy-in/copy-out approach is still used for messages smaller than 4KB. The same configuration is applied to MPICH2 or MVAPICH2: KNEM/LIMIC copy is enabled for large message transfer (LMT). For inter-node communications, the appropriate low level point-to-point transport module is used, depending on the underlying hardware (Open IB, TCP). For all MPI libraries, the process/core binding is the default uniform "by-core" strategy, except when explicitly mentioned. In this default strategy, sequential MPI ranks are bound into adjacent processor cores until all slots of a compute node have been used, then the same process is applied for the next compute node in the list. To summarize, the underlying technology used by our HierKNEM algorithm and all other collective components to perform point-to-point operations is similar and uses KNEM copies, similarly process placement is comparable; therefore any performance difference roots solely in the proposed collective operation innovations.

The Intel MPI benchmark suite IMB-3.2 [20] is used to assess the difference between the collective components on a variety of collective operations. The ASP [21] problem is a typical example of a parallel graph shortest path search algorithm. It is used to illustrate how performance differences in micro-benchmarks translate into application improvement.

4.1. Leader Selection

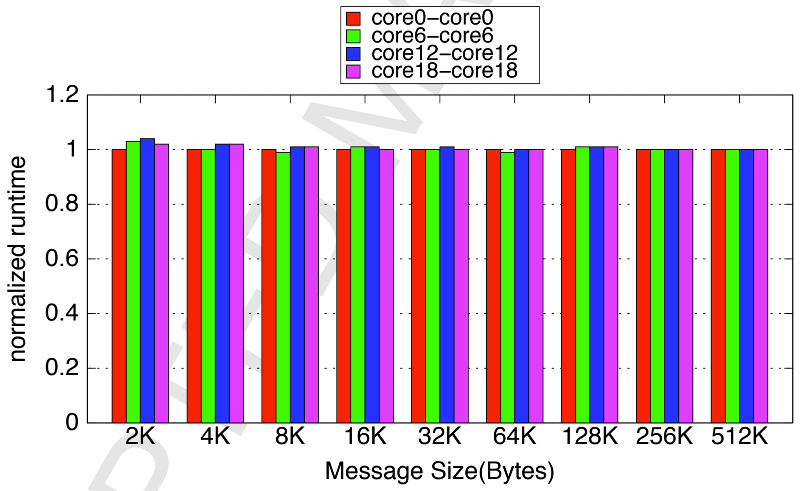


Figure 1: Non Uniform I/O Effect on point-to-point communication (Pingpong Test) Execution Time on Parapluie Cluster; Runtime is normalized to the result for Pingpong test between two compute nodes' core 0. (the smaller, the better).

Similar with Non-Uniform memory access (NUMA), Non-Uniform Input/Output access (NUIO) phenomenon was found in some platforms because some I/O devices are closer to some processors and memory banks than to the others [22]. The latency or bandwidth of inter-node communication between leader processes may be affected by the selection of leader processes due to this Non-Uniform Input/Output access. We did four pingpong tests between the Parapluie's two compute nodes to inspect the NUIO impact on our experiment platforms. To simplify the results in the graph, core 0, 6, 12, and 18 are selected in the experiments to represent other cores on four NUMA memory nodes. And Pingpong tests are only executed between two cores with the same id on two compute nodes. Figure 1 shows the execution time of pingpong tests between pairs of cores normalized to the runtime between core 0s. In most cases, the selection of processes on core 0 or core 6 as leaders will give out the best inter-node communication performance. But the difference is very trivial, about 1% to 2% for message sizes selected as pipeline sizes, so even a wrong leader selection here will not lead to a performance disaster of HierKNEM collectives. In the following experiments, HierKNEM keeps selecting processes with the minimum core id as leader processes in collective communications. In the future release, we will add a module to help HierKNEM select leaders according to the hardware locality [23].

Another issue related with the leader selection is how many leader processes were selected in the collectives. HierKNEM can be configured to select one leader process from each compute node, each board, each NUMA memory node, or each socket. The run-time configuration is decided by the ratio between inter-node and intra-node communication. For example in a scenario of large NUMA compute nodes connected by fast interconnections, e.g. a 20G or 40G Infiniband network, the runtime of intra-node collective communication (broadcast) possibly greatly exceed the runtime of inter-node forwarding (sending/receiving). HierKNEM will adjust the whole hierarchical topology by selecting leaders from each NUMA memory node instead of from each compute node. The HierKNEM Broadcast on the Parapluie Cluster follows this selection. This flexible selection will lead to a better overlapping between intra- and inter-node communication, and meanwhile intra-node memory accesses will be spread evenly over NUMA memory nodes. Except this special case, other leader selections still follow one leader process per compute node in HierKNEM collectives.

4.2. Pipeline Size

In the HierKNEM collective component, both the Broadcast and the Reduce operations are pipelining algorithms, in which messages are split into several smaller chunks. Tuning an optimal size of a chunk is a key criterion of every pipeline algorithm. Figure 2 presents the effect of the pipeline size on the HierKNEM Broadcast execution time. In this Broadcast test, 768 processes are spawned on the Parapluie cluster. To ease figure clarity, the execution time for all pipeline sizes is normalized to the runtime obtained with a pipeline size of 64KB (t_z/t_{64}). One can see that the pipeline size is indeed critical to the HierKNEM collective performance, and a wrong selection of pipeline sizes leads to significant penalty. On one hand, a too small pipeline size results in inefficient inter-node communication, as the small message latency comes to dominate, preventing the full point-to-point bandwidth from being leveraged; as an example, the Broadcast with a pipeline size of 4KB is more than 3 times slower than with 64KB. On the other hand, a too large pipeline size results in long pipeline fan-in and fan-out phases, where the pipeline algorithm is not at steady-state efficiency. Experimentally, 8KB or 64KB is the ideal pipeline size for the Broadcast operation on the Parapluie cluster for messages smaller or larger than 64KB. We did similar experiments for HierKNEM's Broadcast and Reduce on both the Parapluie and Stremi clusters. Table 1 shows the best pipeline size for each operation on each type of cluster. Both HierKNEM's Broadcast and Reduce algorithms use the pipeline size in Table 1 in the following tests.

Table 1: Best pipeline size for Broadcast and Reduce for Differing Network Capacities

Operation	Parapluie (IB20G)		Stremi (Ethernet)	
	message size	pipeline size	message size	pipeline size
Broadcast	[8KB,64KB)	8KB	[8KB,512KB)	16KB
	[64KB,∞)	64KB	[512KB,∞)	32KB
Reduce	[2K, 16MB]	64KB	[2K, 16MB)	64KB
	(16MB,∞)	1MB	[16MB,∞)	1MB

4.3. *Impact of Kernel-assisted approaches*

Intra-node point-to-point operations can be implemented using shared memory or kernel-assisted approaches. Kernel-assisted approaches have the benefit of decreasing the number of memory copies, a factor critical in memory intensive code sections such as the collective communications. Figure 3 shows a comparison of the broadcast collective bandwidth on the Stremi cluster between collective modules over the kernel-assisted or shared memory point-to-point communication. Kernel-assisted approach shows a huge performance boost in Open MPI's collective modules: Tuned and Hierarch collectives, up to 3$\times$ for some message sizes. On the opposite side, the performance difference regarding MPICH2 is less flagrant. Other collective operations, such as Reduce and Allgather, exhibit similar speedup when kernel-assisted approach are replacing shared memory point-to-point communication. Moreover, this performance improvement on collective communications is directly inherited by the applications using them, as highlighted in Section 4.8. Thus, in the remaining of this paper we focus on kernel-assisted approaches, discarding the sub-optimal approaches built on top of shared memory point-to-point communications.

4.4. Allgather Algorithm Selection

Although the two levels of algorithms are tightly integrated, there are still a variety of combinations that are possible, whose performance greatly varies depending on hardware features and properties. In the case of the Allgather algorithm, we identified two combinations of interest: both use the pipelined Tuned collective module between compute nodes, but the internal operation differs depending on the number of cores between compute nodes. Between cores, the algorithm can rely on the leader originating all messages simultaneously (referred to as “leader-based” algorithm), but for large core counts, this approach has the potential to result in heavy traffic contention on the memory bus of the core hosting the leader. For a larger number of cores per compute node, the ring algorithm has more potential to even out the load on all cores. Figure 4 shows the aggregate bandwidth for the two algorithms combinations, for a 512KB message’s Allgather operation on Parapluie’s 32 compute nodes, when increasing the number of processes per compute node from 2 to 24. The leader-based algorithm has a slight performance advantage in dual-core or quad-core compute nodes, as the parallel KNEM accesses overlap one another. For larger setups, the bandwidth contention on the leader core prevents aggregate bandwidth to scale, while the ring algorithm, which proves more scalable thanks to evenly distributing data access load across all memory links, dominates. Results (not presented here) are similar for other message sizes, and when using different inter-node networks on Stremi and Parapluie clusters. In the following tests, we use the ring algorithm as we mainly target large multicore compute nodes.

4.5. Collective communication performance

In this Section we will analyze the performance of three of the most used collective communication operations, namely Broadcast, Reduce and Allgather. They cover all of the regular collective patterns available in MPI, one-to-many, many-to-one and many-to-many, providing a quite extensive view of all the potential performance improvement in collective communications.

4.5.1. Broadcast Performance

Figure 5 presents the aggregate Broadcast bandwidth for HierKNEM, Open MPI’s Hierarch and Tuned modules, and MPICH2 or MVAPICH2 on, respectively, the Ethernet Stremi cluster or the Infiniband Parapluie cluster. On Stremi (Figure 5(a)), for message size between 8KB and 256KB, HierKNEM Broadcast provides a significant speedup, sometimes up to 30x, when compared with MPICH2 and

Open MPI. Compared with Open MPI's Hierarch, our HierKNEM version provides more than twice the aggregate bandwidth in this message size range.

For larger message sizes (superior to 512KB), the most important tuning factors are process mapping and proper pipelining to evenly spread the workload across cores and links. In the Tuned module, the "by core" binding luckily happens, in this experiment, to match the hardware topology; and the pipeline size selected by the tuned module to optimize the network communications is suitable for core communications. The Hierarchical module of Open MPI is not as successful for large messages, because the intra-node and inter-node layers do not cooperate to evenly spread the load of the pipelining algorithm. The leader processes are unavailable for long periods of time when they take part in the shared memory local operation, resulting in effectively sequentializing the local and remote collective operations without opportunity for overlap. With such a large core count, the large intra-node overhead offsets the benefits of the standard hierarchical algorithm. In contrast, the HierKNEM algorithm obtains better performance in all cases, thanks to explicitly taking into account process mapping and using directional KNEM control to offload parts of the operations onto the leaf processes, hence enabling intra and inter-communications to overlap.

Similarly, on the Infiniband cluster (Figure 5(b)), in most cases, the HierKNEM Broadcast still outperforms other collective components. One major difference in the results, when compared with the Ethernet case, is that the performance of the classical hierarchical algorithm is much better for small message sizes. On the Infiniband network, the tuning parameters selected by the two non-cooperating algorithms forming the hierarchical collective are matching better. However, as one can see, the tuning parameters for larger message size are not as lucky; the performance for large messages drops, with the notable exception of 512KB messages, for which the pipeline length matches the balance for 32 processes and 24 cores. This discrepancy illustrates the difficulty of tuning the behavior of separate collective algorithms cooperating in a hierarchical manner. Even with expert knowledge, it's unrealistic to tune Open MPI's Tuned collectives on such a complex system with so many hierarchies and diverse networks, when a small variation in message size results in unexpected and dramatic performance consequences. Although the HierKNEM module is not immune to the challenges of unpredictable and unstable performance on varying hardware, the fact that both algorithms select compatible tuning parameters, that the outer collective operation can overlap imperfection on the inner operation and that the collective topology is constructed to match core hierarchies greatly alleviates this difficulty, as illustrated by more stable results across the message size range.

4.5.2. Reduction Performance

Figure 6 presents the aggregate Reduce bandwidth on the Ethernet cluster (Figure 6(a)). For message sizes between 2KB and 32 KB, the HierKNEM Reduce competes closely with Open MPI's Hierarch Reduce. After 64KB, the HierKNEM Reduce dominates other collective components, thanks to a good overlapping between inter-node Reduce and intra-node Reduce. Similarly with the Broadcast, the Hierarch Reduce worsens for large messages due to the increased intra-node Reduce overhead which can't be dodged by overlap. Again, the performance of the Tuned Reduce improves for messages larger that 4MB, but is still 19%-28% slower than the HierKNEM Reduce.

On the Infiniband cluster (Figure 6(b)), the HierKNEM Reduce clearly dominates for message size in the range of [2KB, 32KB] and (1MB, 16MB]. When message size is between 64KB and 1MB, although HierKNEM Reduce still achieves significant speedup when compared with Open MPI's Hierarch and Tuned Reduce, it's a little behind MVAPICH2 performance, about 20% in the worst case. By profiling a 64KB message's Reduction operation with 32 processes on Parapluie's 32 compute nodes (no multicore or hierarchies), we discovered that the Open MPI Tuned Reduction suffers from a serious performance limitation on the Infiniband network; meanwhile MVAPICH2 enjoys very good performance (366μs for Open MPI compared to 281μs for MVAPICH2). As our HierKNEM composite algorithm reuses the original Tuned module for inter-node communications, it suffers from the same defect and cannot compete with MVAPICH2, until the Open MPI community addresses this issue.

4.5.3. Allgather Performance

Figure 7 presents the aggregate Allgather bandwidth. The HierKNEM Allgather is enabled only when the message size is larger than 8KB. The biggest message size is 1MB, because of the large amount of memory required for this all-to-all operation between 768 processes exhausting available system memory for larger sizes. On both clusters, the HierKNEM Allgather adopts a ring algorithm, as described in section 3.4. The Open MPI Hierarch module is not presented for this collective operation, as it has not been implemented.

On the Infiniband cluster (Figure 7(b)), both MVAPICH2 and Tuned Allgather operations slightly outperform HierKNEM's Allgather. In this message range, Open MPI's Tuned Allgather adopts a similar but more aggressive algorithm: neighbor exchange. Different than the ordinary ring algorithm which exchanges rcount data in each round, 2$\times$ rcount data are exchanged in each round in the neighbor exchange algorithm (except the first round). Neighbor exchange algorithm arranges processes continuous in MPI ranks together in a ring and the "by core" binding strategy used in this test coincidentally maps the logical ring of the Tuned Allgather correctly to the underlying hardware topology. As a consequence, Tuned and HierKNEM are actually running on the same underlying hardware topology, but Tuned does not have to pay for the extra cost of detecting the physical distance between processes, and fast networks like 20 G Infiniband show better performance in a more aggressive policy due to more bandwidth capacity available. On the Ethernet cluster (Figure 7(a)), the HierKNEM Allgather outperforms all other collective components for all message sizes. While adopting a similar ring topology for large messages, the Tuned Allgather on this Ethernet cluster suffers up to 50% in performance loss because the aggressive policy in the neighbor exchange algorithm overloads slow networks decreasing the overall network throughputs. Clearly, adopting an aggressive strategy, like neighbor exchange algorithm, should depends on the capacity of the networks.

4.6. Impact of Process Placement

It is well known that process placement can have a major impact on collective operations performance. Approaches such as MPIPP [6] have been designed to detect communication patterns during a “tuning run”, whose result is used to hint process placement to decrease long distance communication volume during subsequent production runs. However, this approach is not practical in many cases, as it requires being able to run smaller problems that exhibits similar communication patterns; and the collective algorithm underlying communication topology might depend on the message and communicator size. Another difficulty is that oftentimes, one might want to optimize for the pattern of point-to-point operations explicitly realized at the application level (such as the typical hypercube topology found in many CG implementations), which means that the process placement may or may not fit the expectations of the collective modules. As a consequence, the default deployment approach is usually less elaborate and simply allocates ranks sequentially on the available resources.

Figure 8 shows the impact of two typical process placements on the performance of the Broadcast and Allgather operations. The goal of this experiment set is to investigate the sensitivity of the hierarchical approaches to variations in the process placement. As such, more than raw performance, it is the difference between the same algorithm on different mappings that is of interest here. Hierarch has been trimmed from the figure, because it does not feature an Allgather operation, and uses a similar topology as HierKNEM for the Broadcast (hence similar performance trends). Considering the Broadcast (Figure 8(a)), one can witness that hierarchical approaches (HierKNEM and MVAPICH2 both feature a hierarchical algorithm) reach more stable performance. The Tuned algorithm exhibit very unstable performance trends, for some message sizes the bynode binding reaches better performance, while it is the contrary for larger messages.

Figure 8(b) further displays the importance of considering hierarchical features to enable portability of performance across varied process mappings. In this algorithm, the HierKNEM algorithm demonstrates very stable performance when changing from bycore to bynode process mappings. The performance variation between two bindings is less than 10%, which is very small when compared to the tremendous performance penalty suffered by non hierarchical algorithms, commonly more than 6× and sometimes up to 14× increased communication time. In the "by node" binding, the Tuned Allgather uses a ring algorithm for large messages; every edge of the logical ring (768 edges in this case) passes through inter-node links (Infiniband), causing a serious traffic congestion on the Infiniband network. This clearly illustrates the penalty suffered by topology-unaware algorithms when considering irregular process-core bindings. Although our HierKNEM collective pays an overhead due to constructing the internal topology, it provides stable performance independently of process placement. Such a flexible process placement is a desirable feature to enable deeper optimization of the hard-coded point-to-point communication patterns and ensure maximum performance with default settings on complex architectures.

4.7. Core per Node Scalability

In the next experiment, we investigate the trend of aggregate bandwidth when varying the number of cores per compute node. The total number of compute nodes is left unchanged (32 compute nodes), but the number of processes per compute node is increasing for each experiment, reaching up to the maximum of 24 processes per compute node. The message size is kept constant at 2MB. Processes on each compute node are bound to cores sequentially.

On both clusters (Figures 9(a) and 9(b)), the aggregate bandwidth of HierKNEM Broadcast achieves a linear speedup when more cores per compute node are involved, because our HierKNEM Broadcast dodge the intra-node communication overhead by overlapping it with the inter-node message forwarding. Increasing processes (cores) per compute node does not increase the overall Broadcast completion time on these two platforms. This linear speedup can be maintained until the time necessary to perform the entire intra-node communication (a KNEM Broadcast) exceeds the inter-node forwarding time of the network.

4.8. Application Performance

We have asserted the maximum possible performance improvement by solely executing synthetic benchmarks over the modified operations. It is now needed to evaluate how much of this improvement results in improved performance for applications. To evaluate the impact of the HierKNEM collective algorithms on real application performance, we consider a typical parallel graph application: ASP [21].

This application unfolds the parallel Floyd-Warshall algorithm to solve the all pairs shortest path problem. At the beginning of each iteration the master process broadcasts a row of the square matrix representing edges weight to all peers in the communicator, in order to distribute the workload. The outer loop of the algorithm iterates on rows, until the entire matrix is treated. Overall, for a matrix of size N, the algorithm performs N broadcasts, with a message size of column_num $\times$ type_size. As a consequence, MPI_Bcast contributes to the majority of the runtime of the ASP's MPI usage.

Table 2: Kernel-Assisted Approach Comparison: ASP Application Execution Runtime Execution Breakdown on Stremi (Ethernet, 768 processes, 24 cores/ compute node). Using KNEM and SM.

Problem	HierKNEM		Tuned		Hierarch		MPICH2	
Size	Bcast	Total	Bcast	Total	Bcast	Total	Bcast	Total
16K	20.3s	97.4s	229s	308s	31.7s	109s	128s	204s
32K	79s	711s	929s	1560s	173s	806s	417s	1020s

			Tuned-SM		Hierarch-SM		MPICH2-SM	
			Bcast	Total	Bcast	Total	Bcast	Total
16K			229s	309s	66s	145s	124s	201s
32K			929s	1574s	245s	899s	429s	1040s

Table 2 compares the overall execution time and communication time (mostly MPI_Bcast) of the ASP application on the Stremi cluster when using different collective modules. By subtracting the communication time from the overall execution time, one can assert that ASP's computational part remains generally constant for a given problem size, independently of the communication setup. The major performance difference between these four setups comes from the communication overhead (MPI_Bcast). The cost of communications occupies 21% of the overall application runtime for the HierKNEM collective, while it rises to 74% when using Open MPI's default collective. Even considering the hierarchical broadcast, the HierKNEM's ability to overlap between inter and intra communications shows a significant improvement in this application.

The second part of the Table 2 shows the same application (ASP) on the same platform using shared memory point-to-point communication instead of kernel-assisted approaches. Similarly with the prior results in Section 4.3 most collective modules over shared memory point-to-point communication lose performance compared with kernel-assisted point-to-point communication due to double memory copies and more memory usage, sometimes up to 30%. The impact on collective performance is directly translated to waste time for the applications.

4.9. Experiments Accuracy

In this experimental evaluation, we preferred using widely accepted benchmarks, such as the Intel Message Passing (IMB) benchmark, for they are specifically designed to eliminate noise and artifacts: for small and intermediate messages, experiments are executed thousand of times with different roots in order to eliminate any hot caching and pipelining effect, while the experiments are executed hundreds of times for large messages. To further emphasize reproducibility, for each experiment, we reserved the whole clusters to avoid sharing network switches with other users. In addition to these precautions, every experiment was realized multiple times, and we took in account not only the minimum, maximum and average values, but the standard deviation as well. For the message size of 512KBytes, the IMB tests resulted in a bandwidth between 25.6 and 26.2 GBytes, with an average at 25.9GBytes, and a standard deviation of 0.17. Similarly, particular care has been taken to the standard deviation of the ASP application performance. Among 18 times of repeated experiments, the maximum ASP execution time was 146s, the minimum time 144s, the average execution time is 145.1, and the standard deviation is 0.6. These small standard deviation values are indicative that the design of the IMB benchmark, ASP application, and the precautions we have taken guarantee the accuracy and stability of our experimental results.

5. Conclusion

In this paper, we described a kernel-assisted topology-aware collective framework: HierKNEM, which enables efficient combinations of multiple layers of collective algorithms, to tackle collective communication on clusters of many-core compute nodes. The algorithms are built reusing modular combinations of existing collective algorithms (such as the Tuned and the KNEM components in Open MPI). The main contributions of this paper are: (1) propose an adaptive hierarchical collective framework to enable tight collaboration between the collective algorithms pertaining to different layers of the hierarchy, (2) combine offloading and pipelining techniques into the hierarchical framework to release leader processes from intra-node data movement, hence maximizing the overlap between inter- and intra-node communications, and (3) build internal collective topologies to form a mapping between the runtime process-core binding and the hardware features, which means stable collective performance independently of process placement.

We demonstrated the benefits of this approach by devising three hierarchical integrated collective algorithms, one of the most useful for each major type of collective communication (one-to-many: Broadcast, many-to-one: Reduce, and many-to-many: Allgather). Experimental results demonstrate that our approach outperforms not only non hierarchy aware state-of-art MPI implementations (MPICH2 and Tuned Open MPI), although these setups benefit from kernel assisted memory copies as well (KNEM), but also significantly outperforms approaches that account for the hierarchy (MVAPICH2 Broadcast, Hierarch component in Open MPI). A simple leader based algorithm that does not enable pipeline coordination, and intra-node copies offloading, under-performs compared to our HierKNEM approach that introduces these features. The performance improvement is visible not only in synthetic benchmarks, but also results in up to a ten-fold performance improvement when compared to the default non hierarchy aware strategy, and still features two-fold improvements when compared to other hierarchical strategies.

Acknowledgement

Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr).

References

[1] T. Ma, T. Herault, G. Bosilca, and J. J. Dongarra, "Process Distance-aware Adaptive MPI Collective Communications," in *Proc. of IEEE International Conference on Cluster Computing*, 2011.

[2] T. Ma, G. Bosilca, A. Bouteiller, B. Goglin, J. M. Squyres and J. J. Dongarra, "Kernel Assisted Collective Intra-node MPI Communication Among Multi-core and Many-core CPUs," in *Proc. of International Conference on Parallel Processing*, 2011.

[3] "HierKNEM: An Adaptive Framework for Kernel-Assisted and Topology-Aware Collective Communications on Many-core Clusters," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'12)*, 2012, IEEE.

[4] L. Huse, "Collective Communication on Dedicated Clusters of Workstations," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 1999.

[5] G. E. Fagg, G. Bosilca, J. Pješivac-Grbović, T. Angskun and J. J. Dongarra, "Tuned: A flexible high performance collective communication component developed for Open MPI," in *Proccedings of DAPSYS*, 2006.

[6] H. Chen, W. Chen, J. Huang, B. Robert, and H. Kuhn, “MPIPP: an automatic profile-guided parallel process placement toolset for SMP clusters and multi-clusters,” in *Proc. of International conference on Supercomputing*, 2006.

[7] N. T. Karonis, B. R. de Supinski, I. Foster, W. Gropp, E. Lusk, J. Bresnahan, “Exploiting Hierarchy in Parallel Computer Networks to Optimize Collective Operation Performance,” *The International Parallel and Distributed Processing Symposium*, 2000.

[8] Y. Qian and A. Afsahi, “RDMA-based and SMP-aware Multi-port All-Gather on Multi-rail QsNet II SMP Clusters,” *Proc. of International Conference on Parallel Processing*, 2007.

[9] H. Zhu, D. Goodell, W. Gropp, R. Thakur, “Hierarchical Collectives in MPICH2,” *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 2009.

[10] A. R. Mamidala, L. Chai, H. W. Jin, D. K. Panda, “Efficient SMP-aware MPI-level broadcast over InfiniBand’s hardware multicast,” *Workshop on Communication Architecture for Clusters (CAC) held in conjunction with International Parallel and Distributed Processing Symposium*, 2006.

[11] K. Kandalla, H. Subramoni, G. Santhanaraman, M. Koop, D. K. Panda, “Designing multi-leader-based Allgather algorithms for multi-core clusters,” *Proc. of the IEEE International Symposium on Parallel&Distributed Processing*, 2009.

[12] K. Kandalla, H. Subramoni, A. Vishnu, D.K. Panda, “Designing topology-aware collective communication algorithms for large scale InfiniBand clusters: Case studies with Scatter and Gather,” *IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*, 2010.

[13] A. Mamidala, A. Vishnu, D.K. Panda, “Efficient Shared Memory and RDMA Based Design for MPI_Allgather over InfiniBand,” *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 2006.

[14] N. T. Karonis, B. R. De Supinski, I. T. Foster, W. Gropp, E. L. Lusk, “A Multilevel Approach to Topology-Aware Collective Operations in Computational Grids,” *Computing Research Repository*, 2002.

[15] R. Brightwell, K. Pedretti, T. Hudson, “SMARTMAP: operating system support for efficient data sharing among processes on a multi-core processor,” *Proc. of the ACM/IEEE conference on Supercomputing*, 2008.

[16] R. Brightwell, "Exploiting Direct Access Shared Memory for MPI On Multi-Core Processors," *International Journal of High Performance Computing Applications*, 2010.

[17] H. W. Jin, S. Sur, L. Chai, D. K. Panda, "LiMIC: support for high-performance MPI intra-node communication on Linux cluster," *International Conference on Parallel Processing*, 2005.

[18] D. Buntinas, B, Goglin, D. Goodell, G. Mercier, S. Moreaud, "Cache-Efficient, Intranode Large-Message MPI Communication with MPICH2-Nemesis," *International Conference on Parallel Processing*, 2009.

[19] R. Thakur, W. Gropp, "Improving the Performance of Collective Operations in MPICH," *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 2003.

[20] "Intel MPI benchmarks 3.2," *http://software.intel.com/en-us/articles/intel-mpi-benchmarks/*.

[21] A. Plaat, H. E. Bal and R. F. H. Hofman, T. Kielmann "Sensitivity of parallel applications to large differences in bandwidth and latency in two-layer interconnects," *Future Generation Computer Systems*, 2001.

[22] B. Goglin, S. Moreaud, "Dodging Non-Uniform I/O Access in Hierarchical Collective Operations for Multicore Clusters," *Workshop on Communication Architecture for Scalable Systems (CASS) held in conjunction with International Parallel and Distributed Processing Symposium*, 2011.

[23] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, R. Namyst, "HWLOC: a Generic Framework for Managing Hardware Affinities in HPC Applications," *The Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, 2010.

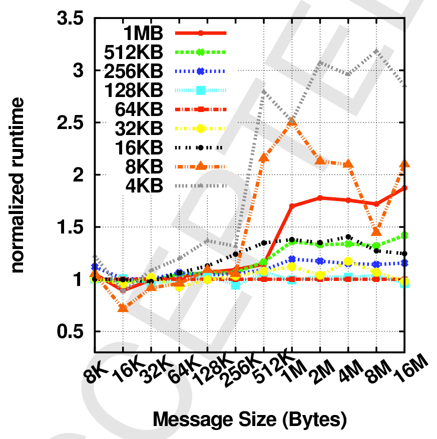


Figure 2: Effect of Pipeline Size on HierKNEM Broadcast Execution Time (Parapluie cluster: 768 Processes, 32 compute nodes, Infiniband 20G); Runtime is normalized to the result for 64KB pipeline size (the smaller, the better).

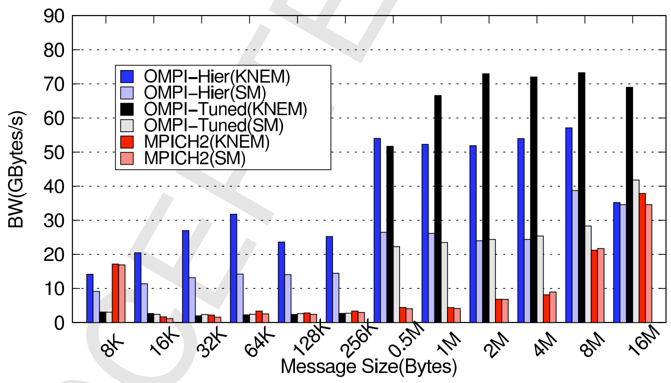


Figure 3: Comparison between Kernel-assisted and Shared Memory Approach: Broadcast Bandwidth on Stremi (Ethernet, 768 processes, 24 cores/ compute node)

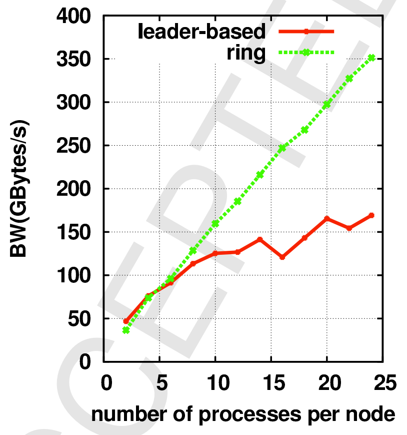
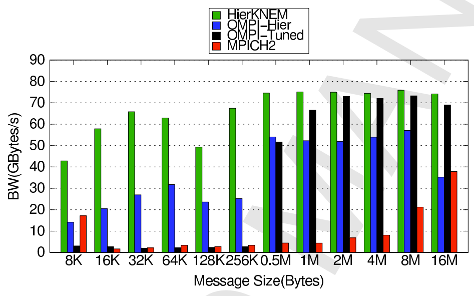
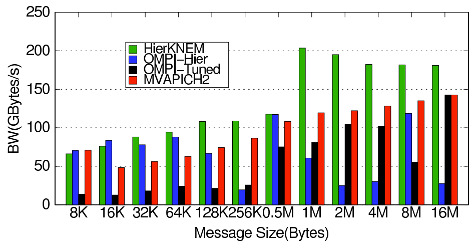


Figure 4: Bandwidth Comparison between Leader-based and Ring Allgather Algorithms, when increasing the number of processes per compute node (from 2 to 24), on Parapluie's 32 compute nodes.

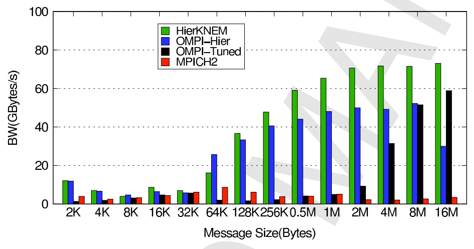


(a) Stremi (Ethernet)

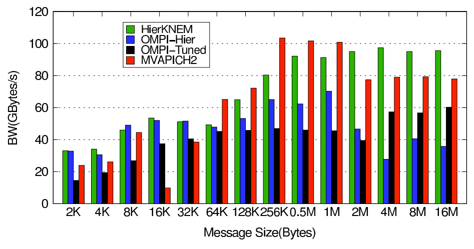


(b) Parapluie (IB20G)

Figure 5: Aggregate Broadcast bandwidth of collective modules on multicore clusters (768 processes,

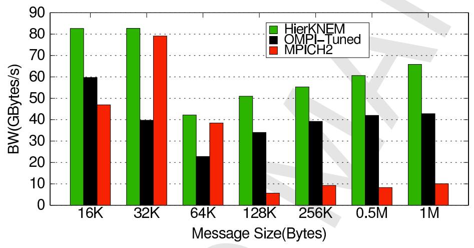


(a) Stremi (Ethernet)

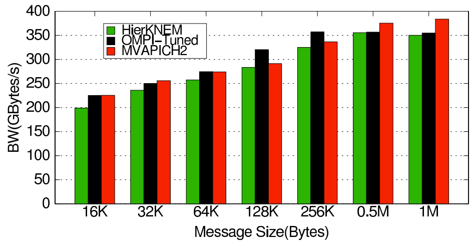


(b) Parapluie (IB20G)

Figure 6: Aggregate Reduce bandwidth of collective modules on multicore clusters (768 processes,

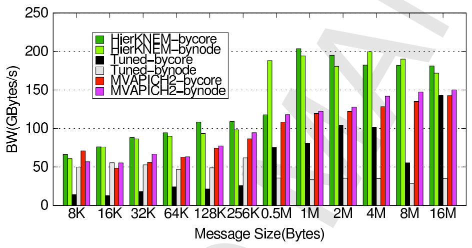


(a) Stremi (Ethernet)

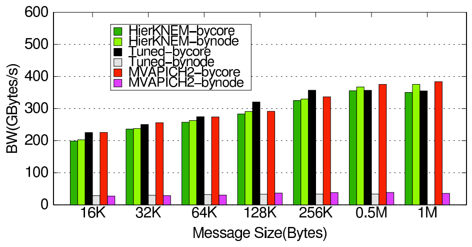


(b) Parapluie (IB20G)

Figure 7: Aggregate Allgather bandwidth of collective modules on multicore clusters (768 processes,

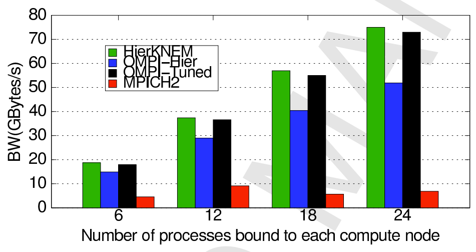


(a) Broadcast

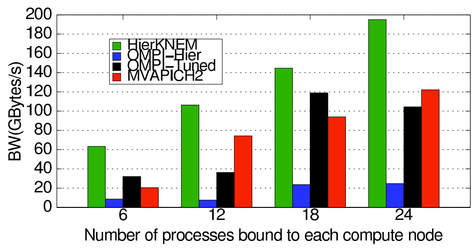


(b) Allgather

Figure 8: Impact of process mapping: aggregate Broadcast and Allgather bandwidth of the collective

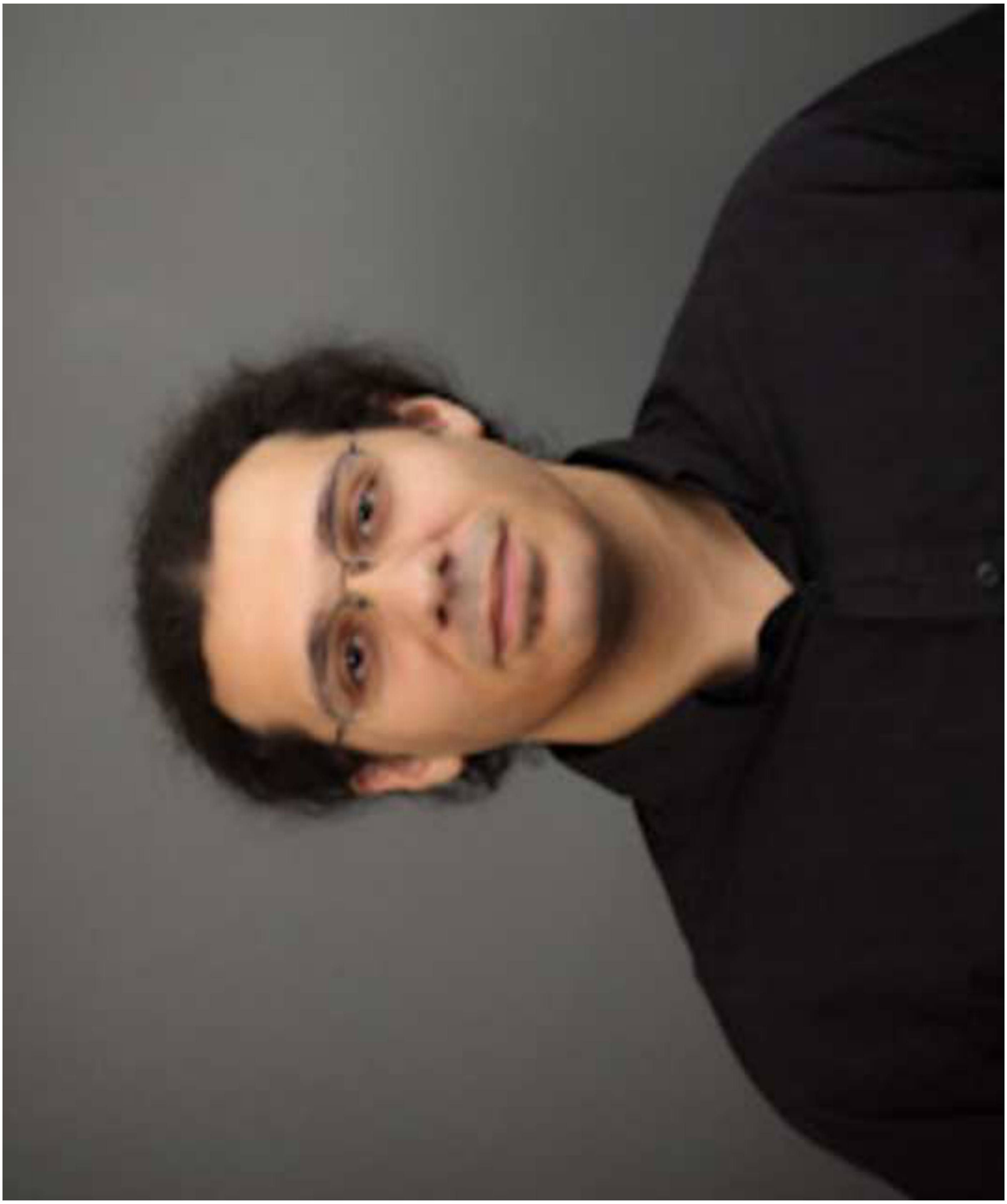


(a) Stremi (Ethernet)



(b) Parapluie (IB20G)

Figure 9: Core per compute node scalability: aggregate bandwidth of Broadcast for 2MB messages



Aurelien Bouteiller received is Ph.D. from University of Paris in 2006, under the direction of Franck Cappello. His research is focused on improving performance and reliability of distributed memory systems. Toward that goal, he investigated automatic (message logging based) checkpointing approaches in MPI, Algorithm Based fault tolerance approaches and their runtime support, mechanisms to improve communication speed and balance within nodes of many-core clusters, and employing emerging data flow programming models to negate the raise of jitter on large scale systems (DAGuE project). These works resulted in over twenty-five publications in international conferences and journals and three distinguished paper awards from IPDPS and EuroPar. He his also a contributor to Open MPI and participates to the MPI-3 Forum.

George Bosilca is a Research Associate Professor of Electrical Engineering and Computer Science, University of Tennessee, Knoxville. He obtained his PhD from the University of Paris XI with a background in computer architecture and parallel computing. Active member of several large scale projects (Open MPI, MPICH-V, CCI, DAGuE), his research encompass a large area in the distributed computing world. From low level network protocols to algorithmic based fault tolerance, Dr. Bosilca research target to reduce the gap between peak and sustained performance on large scale execution environments, effectively taking advantage of the heterogeneous capabilities of current and future computing platforms.

Jack Dongarra holds an appointment at the University of Tennessee, Oak Ridge National Laboratory, and the University of Manchester. He specializes in numerical algorithms in linear algebra, parallel computing, use of advanced-computer architectures, programming methodology, and tools for parallel computers. He was awarded the IEEE Sid Fernbach Award in 2004 for his contributions in the application of high performance computers using innovative approaches; in 2008 he was the recipient of the first IEEE Medal of Excellence in Scalable Computing; in 2010 he was the first recipient of the SIAM Special Interest Group on Supercomputing's award for Career Achievement; and in 2011 he was the recipient of the IEEE IPDPS 2011 Charles Babbage Award. He is a Fellow of the AAAS, ACM, IEEE, and SIAM and a member of the National Academy of Engineering.

Teng Ma is a PhD student in EECS Department of University of Tennessee, Knoxville. His research interest focused on design, implementation, and modeling of parallel communication libraries, parallel computer architectures; cluster and grid computing; modeling of parallel benchmarks; clusters with hardware accelerators; parallel programming models; parallel I/O and distributed file system.