HierKNEM: An Adaptive Framework for Kernel-Assisted and Topology-Aware Collective Communications on Many-core Clusters

Teng Ma*, George Bosilca*, Aurelien Bouteiller*, Jack J. Dongarra†
* *EECS, University of Tennessee*
1122 Volunteer Blvd., Knoxville, TN 37996-3450, USA
Email: {tma, bosilca, bouteill}@eecs.utk.edu
† *University of Tennessee*
Oak Ridge National Laboratory, Oak Ridge, TN, USA
University of Manchester, Manchester, UK
Email: dongarra@eecs.utk.edu

***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 caches 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 core distance, 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 aptitude to support duplex communications or operate multiple concurrent copies simultaneously. This calls for a collaborative approach between multiple layers of collective algorithms, targeting 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 how this framework orchestrates the collaboration between multiple layers of collective algorithms. The resulting scheme enables perfect 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 node, a paramount requirement for scalability on future many-core hardware.

***Keywords*-MPI, multicore, cluster, HPC, collective communication, hierarchical**

I. Introduction

While the insatiable demand of computing power from the domain sciences motivates the deployment of ever powerful High Performance Computing (HPC) systems, thermal and power consumption concerns have curbed the growth of both 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 node. The top ranking machine in the latest Top500 list, the K computer, uses more than half a million cores[1]. Processors with 12 cores are available from major commodity vendors, and it is common to have these deployed in multiple socket boards featuring 8 to 48 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, which threatens the performance efficiency of the machines. Namely, within nodes, non uniform memory accesses (NUMA), memory and shared cache hierarchies, dismiss the assumptions of regular load balance and even 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; its implementations provide both highest performance and portability. With respect to portability, not only an MPI code compiles on different machines, but it also exhibits an excellent efficiency, because network topologies and features are accounted for by the MPI library rather than the application code. With the introduction of multicore nodes, both of these features have been threatened in MPI, most implementations treating multicore nodes as mere SMP units and ignoring their internal hierarchies. To alleviate this issues, some attempts have been made to use hybrid programming models, retaining MPI between nodes and a thread-centric approach (pthreads, OpenMP, TBB, ...) between cores. The suitability of this approach is questionable, research showing a similar number of applications that successfully benefited from the approach compared with failures to reach any performance improvement. It also has several productivity drawbacks: it imposes a significant complexity on programmers, renders explicit management of hierarchies which defeats performance portability, and imposes major rewrite

[1] http://www.top500.org

of legacy applications. We believe that the MPI standard is a competitive proposition for harnessing the power of multicore clusters, should the implementation 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 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 simultaneously. 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 node communications.

In this work, we present how a kernel-assisted topology-aware collective framework: HierKNEM, 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 even the memory copy load among available cores. The resulting scheme enables perfect overlap of intra-node communication time by external communications, thanks to innovative hierarchical algorithms. We demonstrate experimentally, by considering three 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 node, a paramount requirement for scalability on future many-core hardware.

The rest of this paper is organized as follows: Section II 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 III 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's 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 IV. Finally, Section V concludes the paper with a discussion of the results.

II. 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.) [3]. 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 [4]. The Tuned collective module, in Open MPI, is iconic of such an approach; MPICH2 and other MPI libraries feature an implementation of the similar idea. Unfortunately, because the *tuned* collective operations have been designed to fit single-processor clusters, none of these parameters could reflect a runtime processes' topology in a view of physical distances. To further exacerbate this issue, intelligent process placements, used as a bridge between applications and MPI libraries, e.g. MPIPP [5], often break regular process-core binding patterns and schedule continuous processes (in the way of MPI ranks) to cores between a long physical distance. This irregular mapping leads into a further mismatch between collective topologies and underneath hardwares [1].

The conventional effort toward adaptive collective communications to hierarchical hardware topologies is leader-based hierarchical algorithms [6], [7], [8], [9], [10], [11]. The early trying of the hierarchical approach focuses on collective communication on clusters of SMPs [6] or Grids [12] to reduce the message amount crossing high latency and low bandwidth links. Combining with the SMP-aware method, leader-based hierarchical algorithms were widely applied into collective communications on multicore clusters, e.g., MPI on Quadrics networks [7], Open MPI's Hierarch collectives, or MVAPICH2 on Infiniband networks [9], [10], [11]. In these SMP-aware methods, multicore nodes are often treated as shared memory nodes. 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 implies two memory copies to pass a single message, greatly wasting memory

bandwidth and CPU cycles. When applying this approach into leader-based hierarchical algorithms, leader processes are heavily involved into 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 [13], [14] 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 [15] in MVAPICH2 or KNEM [16] 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 [17]. 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 gain [2]. Furthermore, efforts have been made to take into account NUMA hierarchies in the process placement and to optimize intra-node collective algorithms to adapt architectural features [1]. However, these projects focus solely on improving communications within a single shared memory multicore 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 the hardware.

III. Collective Algorithm Composition

A. 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 a 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 cluster 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 are divided between inter- and intra-node communication. Each process has an intra-node communicator encompassing every processes hosted on the same physical 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 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 differentiate 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 in moving data across the PCI bus from the memory bus. On the other side, moving data inside the node generates memory bus traffic, and therefore collides with the network transfer (the data in Figure 2 highlight this fact). Therefore, special care have 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 into inter-node forwarding, without sequentialization experienced by less integrated hierarchical approaches. For communication strictly within large NUMA nodes, different approaches yield varying performance. Our new hierarchical algorithms leverage from the knowledge accumulated on a single node [2] to design sound algorithm compositions that can cope with a large number of cores within 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).

B. 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 node, our broadcast is transformed into a linear algorithm identical to the KNEM one; when each 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 an 'cookie' back at step 2. This cookie is an unique identifier to point to an entry recording rbuf's physical memory address and any other process in the 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 36). 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 31). 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 a segment from its father node, they will notify all non-leader processes on the same node to fetch the segment (step 16 and 22). Upon receiving this notification at step 42, each non-leader process will fetch the segment by a KNEM *get* operation at step 43. 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 13, 15, or 21).

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 messages 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

Input: MPI_Bcast(void *rubf, int count, MPI_Datatype dtype, int root, MPI_Comm comm)

```
1  if P is leader process then
2      Register rbuf into KNEM device and get a cookie;
3      Broadcast this cookie to all non-leader processes on the same node;
4      if P is root node then
5          for i ← 1 to seg_num do
6              Isend segment i to its children in spanning tree;
7              Wait for all Isend;
8          end
9      end
10     else if P is intermediate node then
11         Post Irecv for 1st segment from its father;
12         for i ← 1 to seg_num-1 do
13             Post Irecv for next segment(segment i+1) from its father;
14             Wait for previous Irecv(segment i);
15             Isend received segment(segment i) to its children;
16             Barrier in the lcomm communicator;
17             Wait for all Isend;
18         end
19         if i ≡ seg_num then
20             Wait for previous Irecv(last segment);
21             Isend last segment to its children;
22             Barrier in the lcomm communicator;
23             Wait for Isend;
24         end
25     end
26     else
27         for i ← 1 to seg_num do
28             Recv segment i from its father;
29             Barrier in the lcomm communicator;
30         end
31     end
32     Barrier in the lcomm communicator;
33     Deregister buffer from KNEM device;
34 end
35 else
36     Get KNEM cookie from the leader process;
37     if P is on the same node with root process then
38         Fetch the whole data from root process by KNEM;
39     end
40     else
41         for i ← 1 to seg_num do
42             Barrier in the lcomm communicator;
43             Fetch segment i from leader process by KNEM;
44         end
45         Barrier in the lcomm communicator;
46     end
47 end
```

Algorithm 1: The HierKNEM Broadcast Algorithm.

processes communicator). In the event of a perfect overlap, a multi-core broadcast operation can be made *number-of-nodes* dependent instead of *number-of-cores*.

C. 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  end
7  else if P is the 2^nd leader process then
8      for i ← 1 to seg_num do
9          Fetch segment i from 1^st leader;
10         Reduction between two leaders' segment i;
11         Reduction in the new_comm for segment i;
12         Push reduction result of segment i to 1^st
           leader's tmpbuf;
13         Notify 1^st leader that pushing operation is
           done;
14     end
15 end
16 else if non-leader processes exist inside the node then
17     for i ← 1 to seg_num do
18         Reduction in the new_comm for segment i;
19     end
20 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 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 to 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 node has more than two processes participating to 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 9), and applies the reduction operation between their sbuf's segment i (step 10). 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 10. 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.

D. Allgather

The HierKNEM provides two algorithms for Allgather: a leader-based algorithm for clusters of small nodes(2-6 cores per node) and a ring algorithm for large nodes. Leader-based algorithm has three steps:1) gathering messages into leader processes; 2) exchanging data between leader processes; 3) broadcasting data from leader processes to non-leader processes. Step 1 and 3 happen inside a 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 communicate 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 node or faster network), 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 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 MPICH Allgather ring algorithm [18]: 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 adhere to the physical process distance in terms of sockets and NUMA 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.

IV. Experimental Evaluation

A. Experimental Conditions

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 node). Each socket is a NUMA node, with 12 MB L3 caches and 12 GB of memory. 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 Infiniband 20G.

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 1.7RC1 on the Infiniband cluster (Parapluie). All implementation that supports kernel assisted memory operations use KNEM version 0.9.6 [17].

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: KNEM 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 node have been used, then the same process is applied for the next 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 [19] is used to assess the difference between the collective components on a variety of collective operations. The ASP [20] 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 applications improvement.

B. Pipeline Size

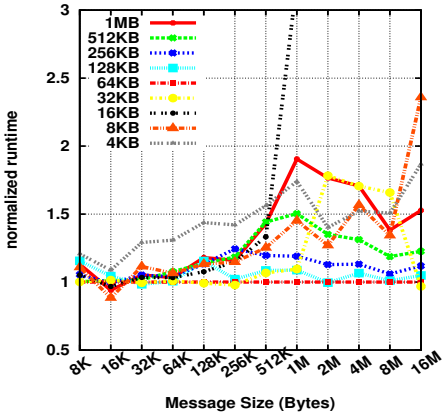


Figure 1. Effect of Pipeline Size on HierKNEM Broadcast Execution Time (Parapluie cluster: 768 Processes, 32 nodes, Infiniband 20G); Runtime is normalized to the result for 64KB pipeline size (the smaller, the better).

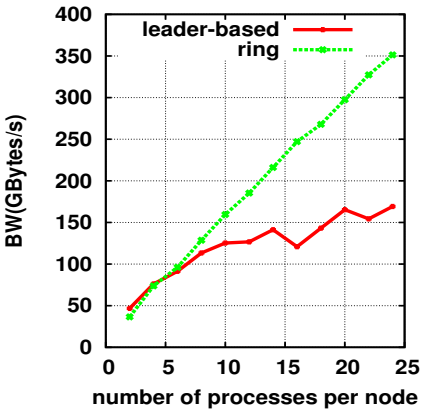
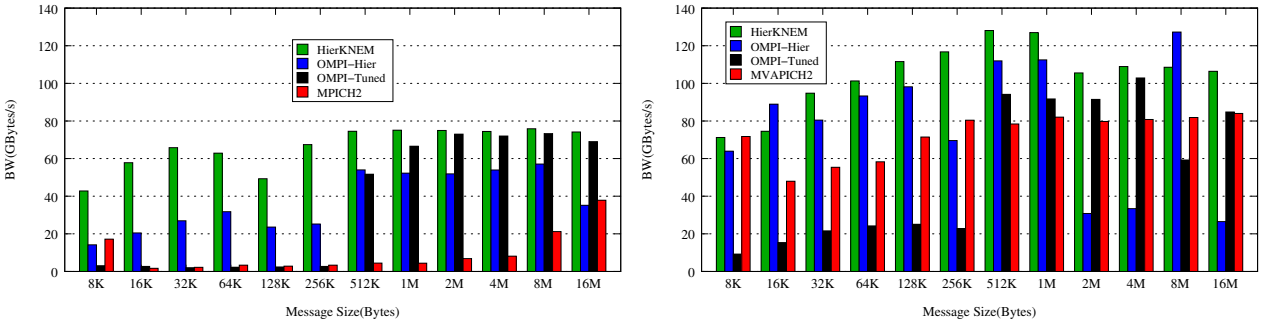


Figure 2. Bandwidth Comparison between Leader-based and Ring Allgather Algorithms, when increasing the number of processes per node (from 2 to 24), on Parapluie's 32 nodes.

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 1 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



(a) Stremi (Ethernet)

(b) Parapluie (IB20G)

Figure 3. Aggregate Broadcast bandwidth of collective modules on multicore clusters (768 processes, 24 cores/node).

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 16KB 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, 64KB is the ideal pipeline size for the Broadcast operation on the Parapluie cluster. We did similar experiments for HierKNEM's Broadcast and Reduce on both the Parapluie and Stremi clusters. Table I shows the best pipeline size for each operation on each type of clusters. Both HierKNEM's Broadcast and Reduce algorithms use the pipeline size in Table I in the following tests.

Table I
BEST PIPELINE SIZE FOR BROADCAST AND REDUCE FOR DIFFERING NETWORK CAPACITIES

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

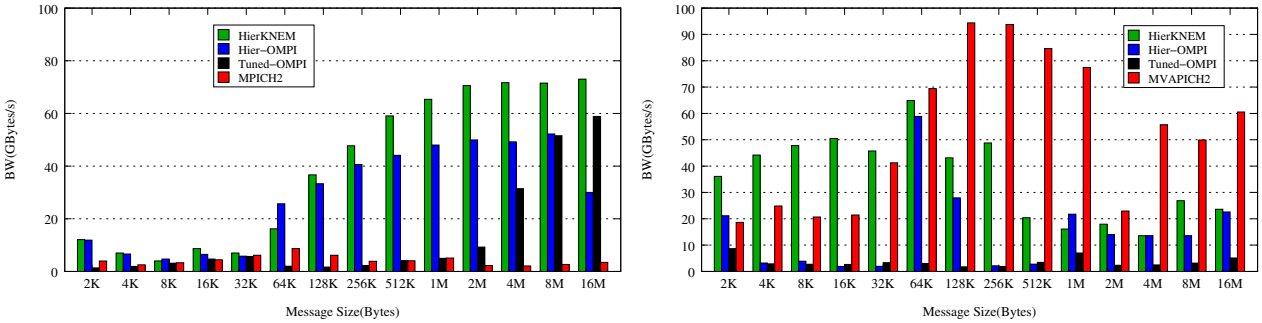
C. Allgather Algorithm Selection

Although the two level 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 nodes, but the internal operation differs depending on the number of cores between 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 into heavy traffic contention on the memory bus of the core hosting the leader. For larger number of nodes, the ring algorithm has more potential to even out the load on all cores. Figure 2 shows the aggregate bandwidth for the two algorithms combinations, for a 512KB message's Allgather operation on Parapluie's 32 nodes, when increasing the number of processes per node from 2 to 24. The leader-based algorithm has a slight performance advantage in dual-core or quad-core 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 size, 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 nodes.

D. Broadcast Performance

Figure 3 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 3(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 default modules. Even compared with Open MPI's Hierarch Broadcast, our HierKNEM Broadcast can provide 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 spreads evenly 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



(a) Stremi (Ethernet)

(b) Parapluie (IB20G)

Figure 4. Aggregate Reduce bandwidth of collective modules on multicore clusters (768 processes, 24 cores/node).

unavailable for long period of times 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. Contrasting, the HierKNEM algorithm obtains better performance in all cases, thanks to taking explicitly 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 3(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 8MB 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.

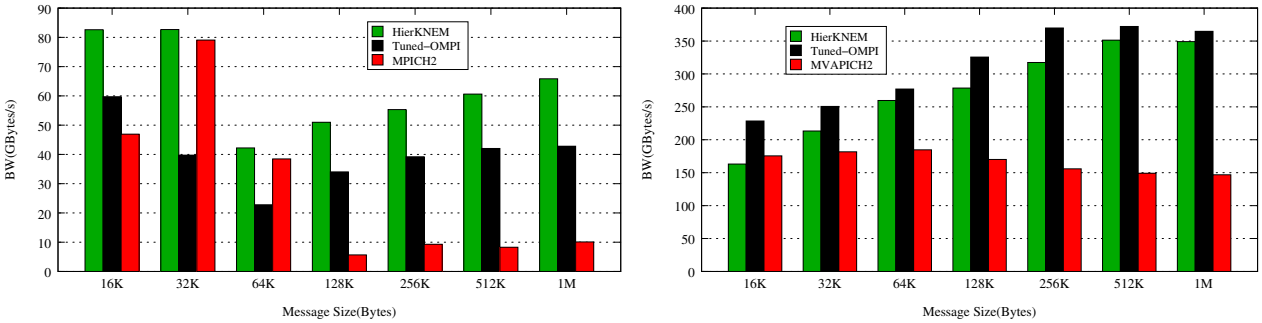
E. Reduction Performance

Figure 4 presents the aggregate Reduce bandwidth on the Ethernet cluster (Figure 4(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 4(b)), the HierKNEM Reduce clearly dominates for message size between 2KB and 32KB. When message size is bigger than 64KB, although HierKNEM Reduce still achieves significant speedup when compared with Open MPI's Hierarch and Tuned Reduce, it cannot match MVAPICH2 performance. By profiling a 64KB message's Reduction operation with 32 processes on Parapluie's 32 nodes (no multicore or hierarchies), we discovered that the Open MPI Tuned Reduction suffers from a serious performance limitation on the Infiniband network, even performing worse than on Gigabit Ethernet; meanwhile MVAPICH2 enjoys very good performance (515μs 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.

F. Allgather Performance

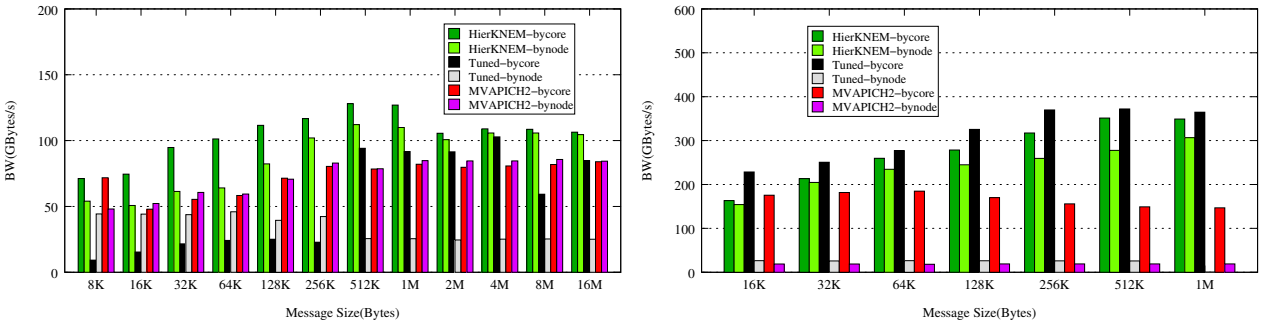
Figure 5 presents the aggregate Allgather bandwidth. The HierKNEM Allgather is enabled only when the message size is larger than 8KB. The biggest message size is 1MB, because of the large amount of memory required for this all-to-all operation between 768 processes exhausting available system memory for larger sizes. On both clusters, the HierKNEM Allgather adopts a ring algorithm, as described



(a) Stremi (Ethernet)

(b) Parapluie (IB20G)

Figure 5. Aggregate Allgather bandwidth of collective modules on multicore clusters (768 processes, 24 cores/node).



(a) Broadcast

(b) Allgather

Figure 6. Impact of process mapping: aggregate Broadcast and Allgather bandwidth of the collective modules for two different process-core bindings: by core and by node (Parapluie cluster, IB20G, 768 processes, 24 cores/node).

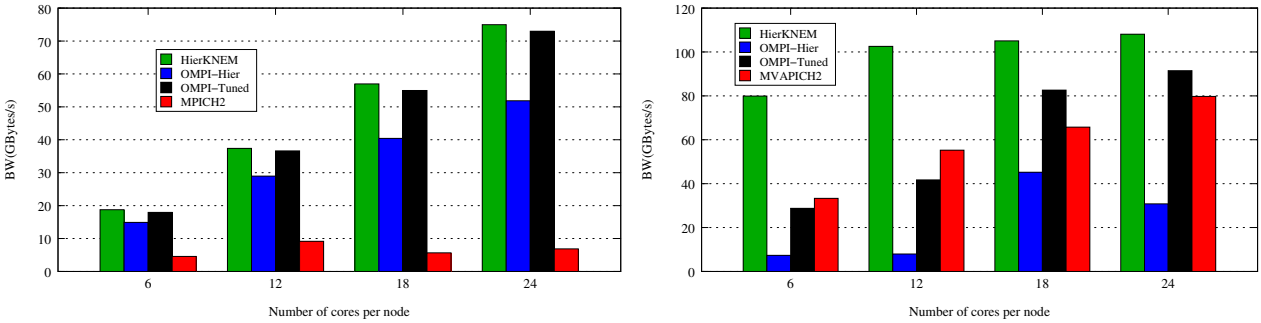
in section III-D. The Open MPI Hierach module is not presented for this collective operation, as it has not been implemented.

On the Infiniband cluster (Figure 5(b)), both HierKNEM and Tuned Allgather operations outperform MVAPICH2's Allgather. In this message range, Open MPI's Tuned Allgather adopts a similar ring algorithm 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 the same algorithm, but Tuned does not have to pay for the extra cost of detecting the physical distance between processes. In future works, we intend to have HierKNEM build the topological map of the cores only once, at communicator creation, hence relieving that performance overhead.

On the Ethernet cluster (Figure 5(a)), the HierKNEM Allgather outperforms all other collective components for all message sizes. While adopting a similar ring algorithm for large messages, the Tuned Allgather on this Ethernet cluster suffers a up to 50% performance loss. We are still working on investigating possible reasons.

G. Impact of Process Placement

It is well known that process placement can have a major impact on collective operations performance. Approaches such as MPIPP [5] have been designed, to detect communication pattern 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 exhibit similar communication patterns; but the selected collective algorithm and its topology depend deeply 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. In the "by core" binding, processes are scheduled on a node until all of its available cores are used before proceeding to the next node. In the "by node" binding, a single process is bound onto each node in a round-robin fashion until all processes have been bound; nodes whose



(a) Stremi (Ethernet)

(b) Parapluie (IB20G)

Figure 7. Core per node scalability: aggregate bandwidth of Broadcast for 2MB messages on multicore clusters (32 nodes).

unbound cores have been exhausted in previous iterations are skipped.

Figure 6 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 6(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. For messages smaller than 256KB, the performance difference between HierKNEM Broadcast on different process mappings is larger than expected, this is because the IMB benchmark changes the root at every iteration of the benchmark sequentially. In the by-core binding, the next root process is located on the same node, hence the send buffer is fully loaded in cache. In the by-node binding, a new node is selected as the root at every iteration, disabling completely cache reuse.

Figure 6(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 18%, 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 (about 25% for 16KB messages and less than 10% for large messages), 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.

H. Core per Node Scalability

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

On the Ethernet cluster (Figure 7(a)), the aggregate bandwidth of HierKNEM Broadcast achieves a linear speedup when more cores per node are involved, because our HierKNEM Broadcast can dodge the intra-node communication overhead by overlapping it with the inter-node message forwarding. Increasing processes (cores) per node does not increase the overall Broadcast completion time on this platform. This linear speedup can be achieved until the time necessary to perform the entire intra-node communication (a KNEM Broadcast) exceeds the inter-node forwarding time of the network.

The scalability experiment on the Infiniband cluster (Figure 7(b)) exactly illustrates that behavior. Starting from 12 cores per node, the time to perform the entire intra-node Broadcast competes with the time to perform the inter-node forwarding, because the faster network dramatically reduces the later. While the linear algorithm adopted in our intra-node Broadcast avoids the pitfalls of overwhelming the

leader's process memory link, even with the help of one-sided operations (KNEM), the intra-node Broadcast ring has a runtime linear with the number of intra-node participants. Our HierKNEM Broadcast is nonetheless the best collective component to reach potential peak performance, and one has to consider that 24 cores per node and only 32 nodes are not typical expected figures for leadership class computing platforms in the future; the node count should be much larger, leading to a better balance of time taken for intra and inter communications. Overall, this experiment demonstrates that an efficient intra-node collective solution plays an important role in enabling future manycore clusters interconnected by faster networks to reach peak performance.

I. 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 [20]. 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 II
ASP APPLICATION EXECUTION RUNTIME EXECUTION BREAKDOWN ON STREMI (ETHERNET, 768 PROCESSES, 24 CORES/NODE).

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

Table II 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 ability to overlap between inter and intra communications shows a significant improvement in this application.

V. 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 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 are setup to 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 offload under-performs compared to our HierKNEM approach that introduces these features. The performance improvement is visible not only in synthetic benchmark, but translates into up to a tenfold performance improvement when compared to the default non hierarchy aware strategy, and still feature twofold 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 *Proceedings of 2011 IEEE International Conference on*

Cluster Computing. Austin, Texas: IEEE, 2011, pp. 196–204.

[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 *Proceedings of 2011 International Conference on Parallel Processing*, Taipei, Taiwan, Sep. 2011, pp. 532–541.

[3] L. Huse, "Collective communication on dedicated clusters of workstations," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, ser. Lecture Notes in Computer Science, J. Dongarra, E. Luque, and T. Margalef, Eds. Springer-Verlag, 1999, vol. 1697, pp. 469–476.

[4] G. E. Fagg, G. Bosilca, J. Pješivac-Grbović, T. Angskun, and J. Dongarra, "Tuned: A flexible high performance collective communication component developed for Open MPI," in *Proccedings of DAPSYS'06*. Innsbruck, Austria: Springer-Verlag, September 2006, pp. 65–72.

[5] H. Chen, W. Chen, J. Huang, B. Robert, and H. Kuhn, "MPIPP: an automatic profile-guided parallel process placement toolset for SMP clusters and multiclusters," in *Proceedings of the 20th annual international conference on Supercomputing*, ser. ICS '06. New York, NY, USA: ACM, 2006, pp. 353–360.

[6] N. T. Karonis, B. R. de Supinski, I. Foster, W. Gropp, E. Lusk, and J. Bresnahan, "Exploiting hierarchy in parallel computer networks to optimize collective operation performance," *The 14th International Parallel and Distributed Processing Symposium*, pp. 377–384, 2000.

[7] Y. Qian and A. Afsahi, "RDMA-based and SMP-aware Multi-port All-Gather on Multi-rail QsNet ii SMP Clusters," in *Parallel Processing, 2007. ICPP 2007. International Conference on*, Sep. 2007, p. 48.

[8] H. Zhu, D. Goodell, W. Gropp, and R. Thakur, "Hierarchical collectives in mpich2," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, ser. Lecture Notes in Computer Science, M. Ropo, J. Westerholm, and J. Dongarra, Eds. Springer Berlin / Heidelberg, 2009, vol. 5759, pp. 325–326.

[9] A. Mamidala, L. Chai, H.-W. Jin, and D. Panda, "Efficient SMP-aware MPI-level broadcast over InfiniBand's hardware multicast," in *6th Workshop on Communication Architecture for Clusters (CAC) held in conjunction with the 20th International Parallel and Distributed Processing Symposium*, April 2006.

[10] K. Kandalla, H. Subramoni, A. Vishnu, and D. Panda, "Designing topology-aware collective communication algorithms for large scale infiniband clusters: Case studies with scatter and gather," in *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, April 2010, pp. 1–8.

[11] A. Mamidala, A. Vishnu, and D. Panda, "Efficient shared memory and rdma based design for mpi_allgather over infiniband," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, ser. Lecture Notes in Computer Science, B. Mohr, J. Trff, J. Worringen, and J. Dongarra, Eds. Springer Berlin / Heidelberg, 2006, vol. 4192, pp. 66–75.

[12] N. T. Karonis, B. R. D. Supinski, I. T. Foster, W. Gropp, and E. L. Lusk, "A multilevel approach to topology-aware collective operations in computational grids," *Computing Research Repository*, 2002.

[13] R. Brightwell, K. Pedretti, and T. Hudson, "Smartmap: operating system support for efficient data sharing among processes on a multi-core processor," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, ser. SC '08. Piscataway, NJ, USA: IEEE Press, 2008, pp. 25:1–25:12.

[14] R. Brightwell, "Exploiting Direct Access Shared Memory for MPI On Multi-Core Processors," *International Journal of High Performance Computing Applications*, vol. 24, no. 1, pp. 69–77, Feb. 2010.

[15] H.-W. Jin, S. Sur, L. Chai, and D. Panda, "LiMIC: support for high-performance MPI intra-node communication on linux cluster," *Parallel Processing, 2005. ICPP 2005. International Conference on*, pp. 184–191, June 2005.

[16] "KNEM: High-Performance Intra-Node MPI Communication," http://runtime.bordeaux.inria.fr/knem/.

[17] D. Buntinas, B. Goglin, D. Goodell, G. Mercier, and S. Moreaud, "Cache-Efficient, Intranode Large-Message MPI Communication with MPICH2-Nemesis," in *Proceedings of the 38th International Conference on Parallel Processing (ICPP-2009)*. Vienna, Austria: IEEE Computer Society Press, SEP. 2009, pp. 462–469.

[18] R. Thakur and W. Gropp, "Improving the performance of collective operations in mpich," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2003, vol. 2840, pp. 257–267.

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

[20] A. Plaat, H. E. Bal, R. F. H. Hofman, and T. Kielmann, "Sensitivity of parallel applications to large differences in bandwidth and latency in two-layer interconnects," *Future Generation Computer Systems*, vol. 17, no. 6, pp. 769 – 782, 2001.