2011 IEEE International Conference on Cluster Computing

Process Distance-aware Adaptive MPI Collective Communications

Teng Ma*, Thomas Herault*, George Bosilca*, Jack J. Dongarra†
* *EECS, University of Tennessee, Knoxville*
Email: {tma, herault, bosilca}@eecs.utk.edu
† *EECS, University of Tennessee, Knoxville*
Oak Ridge National Laboratory, Oak Ridge, TN, USA
University of Manchester, Manchester, UK
Email: dongarra@eecs.utk.edu

***Abstract*—Message Passing Interface (MPI) implementations provide a great flexibility to allow users to arbitrarily bind processes to computing cores to fully exploit clusters of multi-core/many-core nodes. An intelligent process placement can optimize application performance according to underlying hardware architecture and the application's communication pattern. However, such static process placement optimization can't help MPI collective communication, whose topology is dynamic with members in each communicator. Conversely, a mismatch between the collective communication topology, the underlying hardware architecture and the process placement often happens due to the MPI's limited capabilities of dealing with complex environments.**

This paper proposes an adaptive collective communication framework by combining process distance, underlying hardware topologies, and runtime communicator together. Based on this information, an optimal communication topology will be generated to guarantee maximum bandwidth for each MPI collective operation regardless of process placement. Based on this framework, two distance-aware adaptive intra-node collective operations (Broadcast and Allgather) are implemented as examples inside Open MPI's KNEM collective component. The awareness of process distance helps these two operations construct optimal runtime topologies and balance memory accesses across memory nodes. The experiments show these two distance-aware collective operations provide better and more stable performance than current collectives in Open MPI regardless of process placement.

***Keywords*-MPI, Collective Communication, Process Distance, Hierarchical Algorithm, Ring Algorithm**

I. Introduction

Clusters of multi- and many-core nodes are currently the most popular platform in high performance computing.

With the increasing number of computing resources and memory hierarchies integrated into a single compute node, the distribution of MPI processes inside a node become critical in order to fully exploit the node's capabilities. Moreover, even if not yet standardized by the MPI Forum, most of the MPI libraries provide proprietary interfaces to bind MPI processes to specific cores.

A lot of research has been done to adjust the process layout based on an application's communication pattern and underlying hardware architecture. MPIPP [1] provided a profile-guided approach to automatically find the optimal mapping between MPI processes and resources to minimize the cost of point-to-point communications for arbitrary message passing applications. Emmanuel Jeannot *et al.* proposed a near-optimal process placement algorithm called "tree match" that maps processes to resources [2]. E.g. a profiling file shows point-to-point communication between pairs of processes: (0, 1), (2, 4), (3, 6) and (5, 7) occupies the most percentage in MPI communication time. The process placement module will arrange these pairs of processes as closely as possible with respect to physical distance as depicted on Figure 1: pairs of processes are bound to two cores on the same socket. As a bridge between MPI libraries and applications, these intelligent process placement libraries help users find an optimal process placement in order to reduce the communication cost of the whole application.

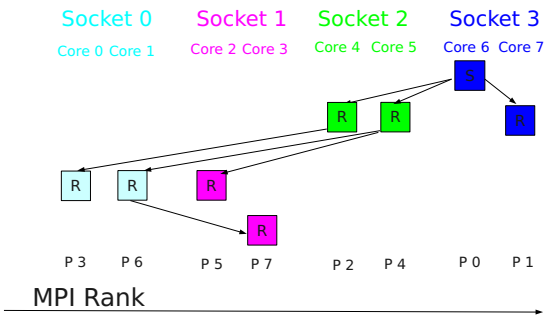

Figure 1. An example of an in-order binomial broadcast tree spanning over 8 processes on quad-socket dual-core nodes, which are placed following applications' communication pattern.

Although these intelligent process placements have the potential to significantly decrease the overall communication time, their methodology is based on a pure point-to-point communication pattern, ignoring the different communication topologies used by MPI collective communications. As most of the collective communications, exhibit specific communication patterns which are allowed to adapt to the underlying architectural features (split binary tree, binomial tree, chain and etc.), there is a mismatch between the MPI internal collective topology (which is usually created based on

978-0-7695-4516-5 2011

DOI 10.1109/CLUSTER.2011.30

the processes MPI ranks) and the external process placement decision. Under the same process placement in Figure 1, let's assume a broadcast operation with these 8 processes on 8 cores in the application. Based on MPI ranks, a binomial broadcast tree is constructed as in Figure 1. Every edge along the critical path($P0 \rightarrow P4 \rightarrow P6 \rightarrow P7$) in the broadcast tree crosses the longest physical distance, a bus connecting each socket. Obviously this binomial broadcast tree will lose its advantage and efficiency due to a mismatch between broadcast topology, underlying hardware architecture, and process placement.

The topology of MPI collective communication varies during the life cycle of application, because the communicator in the application can change, and the order of the processes participating in each collective communication is dynamic. As the process placement is fixed for an application, even with the help of the previously described tools, without precise knowledge of the collective communication algorithm to be used for a specific collective in a communicator, globally optimizing the parallel applications become an very difficult task. However, if internally the MPI library takes advantage of the architectural features of the hardware environment, and adapts its own communication algorithms to maximize the hardware capabilities, solving the process placement problem becomes simpler.

In this paper, we propose a general framework for MPI implementations to detect, express, and take advantage of the runtime process distance. Based on runtime process distance information in the context of each node, the MPI library constructs an adaptive communication topology for each collective operation, and this topology reflects the underlying hardware architecture. This distance-aware collective communication always provides optimal performance regardless of process placement of the members participating in the communicator. This automatic approach at the MPI level provides further optimization and complements the intelligent process placement approach without user intervention.

The rest of this paper is organized as follows: Section II provides some background about collective logical topology and topology-aware MPI communication. Section III formulates and outlines the extent of the problem between process placement, underlying architecture and the runtime communicator. Then, Section IV describes our framework designed to combine process distance information with collective communication topologies, and two distance-aware adaptive collective operations (Broadcast and Allgather) are implemented in Open MPI's KNEM collective communication component as examples. A performance study is presented in the Section V, substantiating the benefits of our approach when compared to the Open MPI's default optimized collective component. Finally, Section VI concludes the paper with a discussion of the results and future directions.

II. Related Work

One of the most straightforward ways of improving the performance of collective communications is to adopt specialized communication topologies (linear, chain, split binary tree, binomial tree and etc.) [3], based on the network properties, such as latency and bandwidth, and collective communication requirements (amount of data, number of participating peers). Both MPICH2 [4] and Open MPI [5] implemented optimized collective algorithms and provide a runtime selection framework to determine the optimal algorithms based on message and communicator size, e.g. tuned collective in Open MPI. These algorithms actually use "fixed" topologies decided by pre-defined fan-out and communicator size. Additionally, most of these fixed topologies are built based on MPI's logical ranks, and are totally agnostic of process placement. It's impossible for these algorithms to provide stable and optimal performance under all circumstances: any process placement, any underlying hardware architecture, and dynamic communicators.

One of the first articles to depict the importance of process placement [6], describes the remapping to MPI libraries according to underneath hierarchical architecture such as clusters of SMP nodes. The proposed methodology, based on graph partitioning, generates a multi-level hierarchical view of the hardware environment. Similar approaches found their way into MPI implementations a few years later when the first platform-specific topology-aware vendor MPI libraries appeared on the market [6], [7].

Several research teams have investigated topology-aware collective algorithms on specific platforms: cluster of clusters, Grids and InfiniBand clusters. Thilo Kielmann *et al.* proposed MagPIe [8] which is a hierarchical MPI collective communication operations for cluster of clusters. A similar topology-aware multilevel approach is introduced in [9] to tackle collective operations in Grids. D.K.Panda *et al.* proposed SMP-aware [10] or topology-aware [11] collective operations over InfiniBand clusters. These topology-aware algorithms achieved a significant performance increase using knowledge about the underlying networks or hardware architecture. However, most of these algorithms only tackle special cases, and some of them are difficult to port in general MPI libraries due to a lack of a generalized MPI framework to express and detect distances between processes. With more hierarchies introduced into nodes or clusters, there is an urgent demand for a general framework for MPI to detect and express process layouts based on process distance.

Teng Ma *et al.* proposed a framework to select the optimal communication parameters according to run-time process distance between peers for MPI intra-node point-to-point communication in [12]. Compared with point-to-point communication, collective communication is more demanding about the physical topology information, and more sensitive about memory hierarchies, and distance between processes.

With the knowledge of such hardware information, efficient scheduling can be implemented in specific collective operations to balance memory accesses across memory controllers and maximize the overall collective communication bandwidth.

The Portable Hardware Locality (hwloc) [13] is a project providing a much needed portable abstraction of hierarchical topology of modern CPUs, including NUMA nodes, sockets, caches, cores, and simultaneous multithreading. Hwloc software package has been widely adopted in state-of-art MPI libraries: Open MPI, MPICH2, and MVAPICH2. Our run-time process distance detection framework is also based on the information collected by hwloc.

III. Mismatch Problem

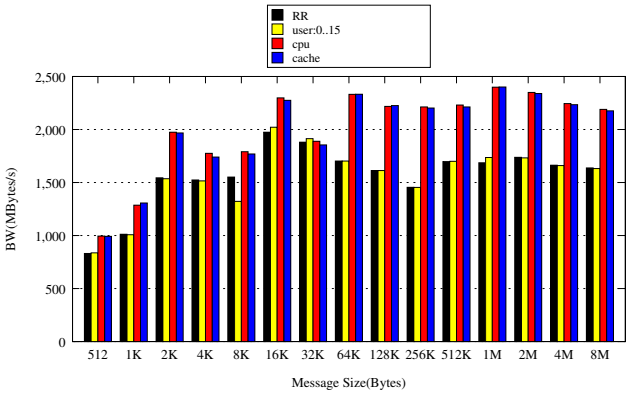

Figure 2. Bandwidth Comparison of MPICH2-1.4 Broadcast Operation on Zoot between 4 binding cases:round-robin, user:0..15, CPU and cache.

Nowadays most of the MPI implementations recognize the process placement problem. In order to alleviate the issue, they propose proprietary interfaces to bind processes to resources. For example, Hydra, the process manager in MPICH2, provides users with options like 'user', 'rr', 'cpu', and 'cache'. "-binding user" is a user-defined binding. "-binding rr" is based on a round-robin mechanism using the operating system (OS) processor IDs. MPICH2 can also provide some cpu-aware or cache-aware allocation strategies such as "-binding cpu" that packs processes as closely as possible to each other with respect to CPU cores, and "-binding cache" that does it respectfully to the cache layout.

Figure 2 shows a bandwidth comparison of Broadcast operation (MPICH2 1.4), on an SMP node between four different binding strategies: round-robin(rr), user-defined binding(user), CPU and cache. The experimental setup consists of a quad-socket quad-core Intel Tigerton processor with a single memory controller on the front side bus (FSB), where logical consecutive core IDs belong to different sockets. Round-robin binding uses the logical order provided by the OS. MPI processes (ranks from 0 to 15) are bound to processor units(PU) in order (P#[0..15]), making neighbor processes always connected directly over the single memory FSB. 'user:0..15' binding strategy has the same binding map with round-robin binding on Zoot. Numbers behind 'user' are just PU's physical identities provided by the OS. Binding by 'cpu' and 'cache' gives out a different map from the round-robin strategy on Zoot. Neighbor processes (close in MPI ranks) are also close from each other, with respect to the number of buses to traverse.

In Figure 2, the same MPICH2's broadcast algorithm provides different performance under different process placement on Zoot. Compared with the 'cpu' and 'cache' cases, the bandwidth is reduced by up to 35% in the round-robin and user-defined cases. As MPICH's broadcast algorithm constructs the broadcast topology according to MPI's logical ranks, which does not include information about the physical topology, 'rr' and 'user:0..15' place the processes in a way that forces the broadcast algorithm to transfer data several times over the FSB, and therefore reducing the possibilities of cache reuse. On the opposite way, in the 'cpu' and 'cache' cases, the MPI library places the MPI ranks in a compact way: neighbor processes run on physically close cores. It turns out that the way the MPICH2 broadcast algorithm builds its internal tree matches this distribution scheme, which leads to more cache reuse and therefore a decreased communication time. While this worked for the MPI_COMM_WORLD communicator used in the tests, it is obvious that a communicator with the rank rearranged would have shown significantly different performance in the same process placement scenario. Moreover, the same algorithm building the internal communication tree differently, would not have reached the same performance level. Such issues are not particular to the MPICH2 library, and they can be found in other MPI implementations.

The fundamental problem in this mismatch phenomenon lies in MPI libraries' ignorance of inter-process distance; the collective communication topology is constructed according to logical MPI ranks not based on physical distances. In this context, even the most highly balanced and tuned algorithms will suffer from varied communication patterns, underlying hardware architecture, and process placement. Therefore, MPI libraries must be aware of runtime process placement, and directly reflect the physical topology on the communication pattern of the collective algorithms. Although MPI libraries can't modify runtime process placement, they can re-construct the internal communication topology according to the distance between cores on which processes run, instead of simply the MPI ranks, to match the underlying physical topologies.

The approach presented in this paper will automatically build the best fitting collective topology for each communicator, depending on the processes involved in the communicator, the collective algorithms in use, the hardware capabilities and the process placement.

IV. Framework

A. Distance Between Processes

The collective framework described in this paper bases its decisions on the distance between processes. Process distance reflects how many functional units, buses, caches, memory controllers and etc. messages travel through from source memory to destination memory. In fact, this distance is computed under the assumption that each process is bound to a particular computing unit (core), and therefore the distance is relative to the core placement at the hardware level. As a result, the distances can be measured using the memory and physical hierarchies. We use four factors to compute the distances between two processes (or cores as explained above): (1) sharing any caches; (2) on the same physical socket; (3) sharing any memory controllers; and (4) on the same physical board. To simplify the algorithm design, processes sharing any caches (L1, L2 or L3 caches) are considered as being at distance of '1', regardless of the shared cache hierarchy. For processes without common caches, we check whether or not they can satisfy conditions (2) and (3). If (2) and (3) are both satisfied, process are at distance '2'. If (2) is unsatisfied and (3) is satisfied, process are at distance '3'. If (2) is satisfied and (3) is unsatisfied, process are at distance '4'. If (2) and (3) are both unsatisfied, the distance is solely determined based on condition (4). Processes on the same board are at distance of '5', otherwise the distance is considered as being '6'. Distance range can be further extended if network-style routers or switches (Intel QPI or AMD HT) are interconnecting NUMA nodes or boards. At the inter-node level, the distance can take into account network adapters, links, and even switches and routers, by a simple and natural extension. However, in the context of this paper we limit the distance to '6'.

If we look at two particular hardware generations, we can exemplify the distances as follows. Zoot, described in detail in Section III, is a 16 core machine with 32GB of memory. The system has four sockets with a quad-core 2.40 GHz Intel Xeon Tigerton E7340 featuring 4 MB L2 caches shared between pairs of cores. A single SMP memory controller in the north-bridge chipset connects all the sockets with the global shared memory. There are several scenarios for distances between processes on Zoot. MPI processes can be bound to different cores on the same die, sharing a L2 cache (distance '1'), different dies on the same socket (distance '2') or on different sockets (distance '3').

With more cores, memory/cache hierarchies, and network-style interconnects integrated into modern multi-core processors, distance between processes becomes more complex. IG is a 48-core machine with 128GB of memory depicted as Figure 3. The system is composed of 8 sockets with a six-core 2.8 GHz AMD Opteron 8439 SE, 5 MB L3 caches and 16 GB memory per NUMA node. The sockets are further divided as two sets of 4 sockets on two separate boards

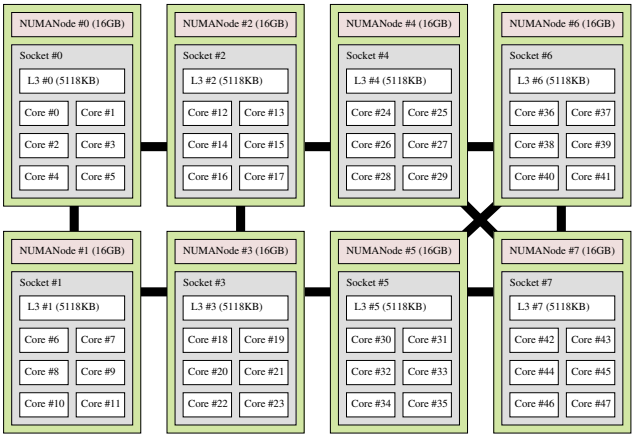

Figure 3. IG architecture, A node with AMD Istanbul 8-socket 6-core processors with 8 NUMA nodes each with 16GB memory.

connected by an interlink. Each core on IG has a 64KB L1 cache and a 512KB L2 cache not shared with other cores. Figure 3 trimmed these L1 and L2 caches to save space. Six cores inside one socket share L3 cache and one memory controller. Distances between processes bound to the 6 cores of the same socket are equally distance '1'. Processes on different NUMA nodes/sockets but on the same board, e.g. between core#0 and core#12, are assigned the distance '5'. Processes bound to cores on different boards, e.g. between core#0 and core#24 are at distance '6'.

Based on the runtime process distance information, we construct our distance-aware collective communication framework. As a proof of concept, we implemented two distance-aware collective communications: Broadcast and Allgather as an extension to the KNEM collective component. KNEM collective [14] is implemented as an Open MPI's intra-node collective component, which is directly based on kernel single-copy module: KNEM. The details about KNEM copy can be found in the papers [15], [16]. KNEM-based collectives mainly accelerate large messages' collective communication, and not small messages, because trapping into the kernel and distributing cookies introduce an overhead (which is equivalent to a 16KB broadcast or a 2KB Allgather on the platforms described above) [14].

B. Distance-Aware KNEM Broadcast

Algorithm 1 shows how the broadcast tree is constructed based on the distance between processes. Vertexes in the graph stand for processes participating in the communicator. Edge weight is the distance. This algorithm is similar to the Kruskal minimum spanning tree [17], with a single major change: the order of edges in the queue Q. The edges in the queue are sorted in an increasing order by weight. For the edges with the same weight, we check whether or not one of the edges includes the vertex of the root process. Edges

including the root process will be moved in the front of queue. For all edges with identical weight covering the root process, we order them by their non-root vertex's MPI rank in a non-decreasing order. For edges with the same weight without root vertex, we order them firstly by small MPI rank in one vertex and then by big rank in another vertex. The MAKE-SET(v) function constructs a set for vertex v. The FIND-SET(v) function returns the head node of the set including vertex v, which is the root process if it includes it, or a process (vertex) with the smallest MPI rank in each set if not. After sorting, the root process, or the process with the smallest MPI rank in the set, will be selected as a leader of each set. The union of all the sets build a tree with the minimum depth among all minimum weight spanning trees based on this sorting.

Algorithm 1 Distance-Aware Broadcast Tree Construction

```
Define a forest T ← ∅
for each vertex v ∈ V[G] do
    MAKE-SET(v)
end for
First store the edges of E in queue Q by non-decreasing orders of weight, whether including root and then MPI ranks.
for each edge (u, v) ∈ E from Q and T has fewer than n-1 edges do
    if FIND-SET(u) ≠ FIND-SET(v) then
        T ← T ∪ (u,v)
        UNION(u,v)
    end if
end for
return T
```

Figure 4 shows an example of distance-aware broadcast with 12 processes across 12 cores distributed among 4 NUMA nodes under a random binding case. NUMA node 0 and 1 are on the same board connected to another board containing NUMA node 2 and 3. There are three kinds of process distances. Processes on the same NUMA node are at the shortest distance (2). Processes on different NUMA nodes but the same board are at distance (5). The longest distance (6) is between processes residing on different boards connected by a slow network. The distance-aware broadcast constructs a spanning tree with root process P5 as depicted in Figure 4. The steps from (1) to (11) in Figure 4 show a sequence of union between sets. The distance-aware broadcast algorithm minimizes the number of messages crossing the slowest links. In the case of Figure 4, only one chunk of message crosses the links that interconnect two boards. Between processes on the same distance set, the 'leader' of each set is the process with the smallest MPI rank or the root process(any other heuristic results in an equivalent outcome). These leader processes are used to communicate with processes on the upper levels with the shortest distance. The processes at the same distance with the leader process are connected directly to the leader process of each set. This guarantees a tree with a minimum depth among all minimum weight spanning trees attributable to the order of the edges in the queue.

In the case of large messages, a pipeline can be applied along the paths of a tree containing intermediate nodes, to reduce waiting time between processes on tree's intermediate nodes or leaf nodes. E.g. along the path of $P5 \rightarrow P1 \rightarrow P7 \rightarrow P10$ in Figure 4, the message is split into multiple chunks, and once such a piece of data is received, a process will notify its children in the tree, creating a pipeline effect.

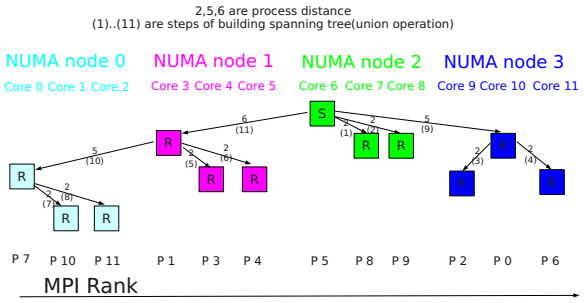

Figure 4. An example of a distance-aware broadcast with 12 processes on 4 NUMA nodes with random binding between processes and cores.

The distance-aware broadcast topology is different from 'fixed' topologies, which are statically decided based on the number of nodes and the expected fan-out. Our topology is dynamically adapting, varying with runtime process distribution, underlying architecture, and processes in the communicator used by the collective communication. It always provides the optimal topology regardless of process placement.

C. Distance-Aware KNEM Allgather

Algorithm 2 shows how an allgather ring topology is constructed based on the process distance. Similar to the broadcast algorithm, the distance-aware allgather uses a greedy algorithm to construct the ring. The edges in the queue Q are in increasing order, first by weights, and then by MPI ranks. Physical neighbor processes are clustered together in this greedy algorithm. Only processes on edges between sets will communicate with each other via the slower links.

Figure 5 shows an example of distance-aware KNEM Allgather operations with 8 processes bound to a quad-socket dual-core node under a random binding case. Processes are organized in a ring structure and physical neighbor processes are arranged together along the ring. A local memory copy is executed at step (1) from sender buffer to receiver buffer with an offset of $rank \times type\_size \times count$. This is the same

Algorithm 2 Distance-Aware Allgather Ring Construction

Define a forest $R \leftarrow \emptyset$
for each vertex $v \in V[G]$ **do**
  MAKE-SET(v)
end for
First store the edges of E in queue Q by non-decreasing orders of weight, and then MPI ranks.
for each edge $(u, v) \in E$ from Q and R has fewer than n-1 edges **do**
  if FIND-SET(u) $\neq$ FIND-SET(v) and fan-out of vertex u and v in each set is less than 2 **then**
    R $\leftarrow$ R $\cup$ (u,v)
    UNION(u,v)
  end if
end for
R $\leftarrow$ R $\cup$ (u',v'), in which u' or v' are head or tail nodes in R.
return R

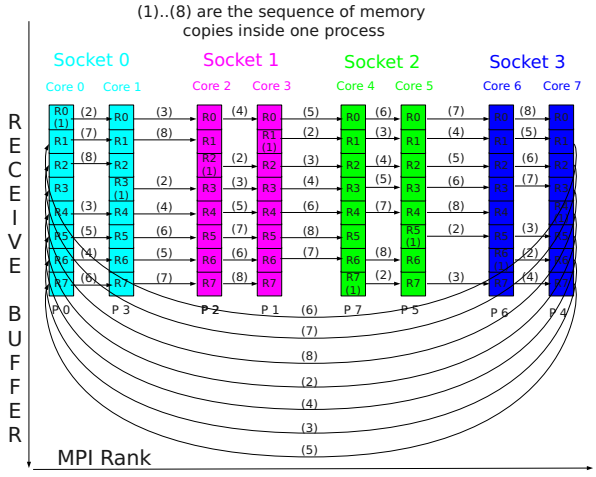

Figure 5. An example of distance-aware Allgather with 8 processes on a quad-socket dual-core node with random binding between processes and cores.

with the step (1) in Figure 5. After finishing step (1), each process sends its current working index (rank in step (1)) by out-of-bound transfer to its right neighbor to notify the neighbor that a buffer is available to be retrieved. KNEM copy will handle this one-sided RMA-style pull operation. Step (2) will be repeated $N - 1$ times until all buffers are copied, where N is the communicator size. The whole operation works like an out-of-order pipeline.

Let us take an example of Allgather operation with 48 processes bound to IG's 48 cores 3. No matter what process placement, KNEM Allgather always constructs a ring and organizes physical neighbor MPI processes together along the ring. Processes on the same socket/NUMA node (distance '1'), are clustered as a set; 8 sets are formed and processes in each set are arranged with a non-decreasing order of MPI ranks. These 8 sets are connected in a ring following the order [2, 0, 1, 3, 4, 6, 7, 5] of NUMA node identities. Physical neighbor processes are organized as closely as possible. Left and right neighbor process ranks are calculated in the way described above.

A theoretical analysis of the memory accesses is shown for a ring-style allgather operation. Let's assume an Allgather operation with $N \times P$ processes unfolded across N NUMA nodes with P cores in each NUMA node. Each read/write access touches an amount of $type\_size \times count$ memory. Each NUMA node has $P \times P \times N$ memory reads and $P \times P \times N$ memory writes. The overall remote memory accesses along the slow links are as small as possible in this Allgather, which is the product of the number of links between NUMA nodes, and $P \times N - 1$ ($links \times (P \times N - 1)$). Only processes on the edge of each set have remote memory accesses. This distance-aware KNEM Allgather is perfectly balanced in terms of workloads to each process and memory accesses to each NUMA node. Each process has $P \times N$ times of memory copies. And there is no hot-spot for any memory controller and memory accesses are distributed evenly across memory controllers. The overhead of this algorithm lies in the $N \times P$ times synchronization for each process to notify its right neighbor when its buffer is ready for transfer. Compared with a large data transfer time, this synchronization overhead is negligible in the intra-node case.

V. Experiments

A. Broadcast and Allgather

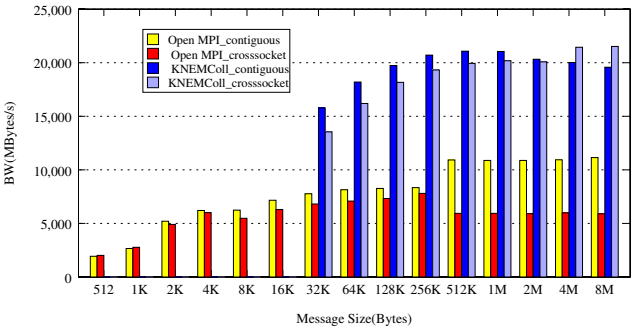

Figure 6. Bandwidth Comparison for Broadcast for Open MPI's tuned and KNEM collective on IG between 2 binding cases:contiguous case and cross socket case.

We use IG (see Figure 3) as our main experimental platform, due to it's complex hardware architecture (described in Section III). It represents a wide variety of multi-core and many-core designs, with several levels of memory hierarchies and physical distances, and can greatly benefit from a distance-aware collective communication framework. As distance-aware algorithms are only implemented in KNEM collective component, our experiment focuses on intra-node

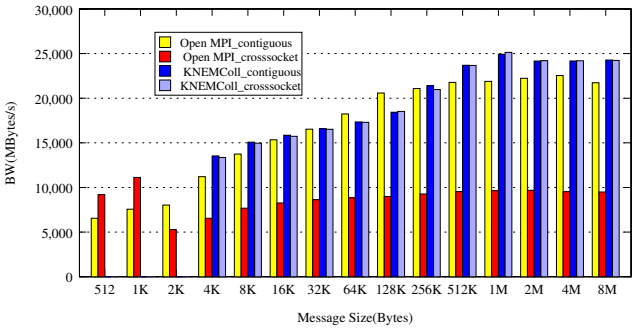

Figure 7. Bandwidth Comparison for Allgather for Open MPI's tuned and KNEM collective with 48 processes on IG between 2 binding cases:contiguous case and cross socket case.

communication. The software setup includes KNEM version 0.9.5 [18]. The Intel MPI benchmark suite IMB-3.2 [19] was used to assess the difference between the collective components. Because KNEM collectives have a smaller memory footprint than other components, cache reuse in the benchmark was disabled with the *off-cache* option for a fair comparison. We used the Open MPI's default collective component, *tuned*, to compare against. This collective component is not distance-aware, but it is considered as a state-of-the-art implementation for intra-node collective communications. In this particular context (on a single shared memory node), tuned collective is configured based on SM/KNEM BTL (byte transfer layer) as point-to-point communication underneath. SM/KNEM BTL uses KNEM copy to speedup point-to-point communication for messages larger than 4KB. Copy-in/copy-out based on shared memory is used in SM/KNEM BTL to deliver messages smaller than 4KB. In terms of point-to-point performance, the underlying technology used by the Open MPI tuned collective and our distance-aware collective component are similar, and therefore any difference in performance is expected to come from the factors non-related to the point-to-point protocol.

In order to minimize the number of results, and to present a clear picture, we decided to compare two possible process placement cases: a contiguous case and a cross socket case; they represent what is expected to deliver the best, and respectively the worst, performance for a non distance-aware collective component (such as *tuned*). The contiguous case is packing processes as closely as possible, similar to MPICH2's 'cpu' or 'cache' binding methods. E.g. in IG's contiguous case, 48 MPI processes are bound in the same order with core identities: with process i bound to core i in Figure 3. In the cross socket case, MPI processes from rank 0 to 47 are bound to cores in an order that maximizes the inter-socket exchanges. More precisely in the IG case the core c holds the MPI rank r iff $c = (r \mod 8) \times 6 + \lfloor r/8 \rfloor$.

Figure 6 shows the bandwidth comparison for the broadcast collective from Open MPI's tuned and KNEM collective on IG between the contiguous case and cross socket case. Tuned Broadcast's performance is not sensitive to process placement when message size is smaller than 1KB. When the message size is larger than 256KB, the difference from different binding cases become obvious. The bandwidth loss for Open MPI's tuned collective in cross socket case reaches more than 45%, when compared with in contiguous case. On the opposite spectrum, KNEM collective provides stable bandwidth regardless of process placement. The variance between contiguous and cross socket cases is less than 14% in KNEM collective, and can be attributed to the increasing cost of these synchronizations in the cross-socket case. Once the messages are large enough, the cost of this synchronizations become insignificant, and the performance of the cross-socket case is slightly better, even as the tree constructed in both cases is similar.

Figure 7 shows the bandwidth comparison of Allgather operations of Open MPI's tuned and KNEM collective on IG between contiguous case and cross socket case. The bandwidth variance of tuned Allgather between different binding cases can reach up to 58%, significantly more than in broadcast communication. This is because an Allgather is more communication-intensive than a Broadcast. In cross socket case, all memory accesses between neighbors in tuned Allgather are remote memory accesses, which are slower than local memory accesses. KNEM Allgather always provides a stable bandwidth regardless of the process binding.

Using the process distance information, KNEM Broadcast and Allgather construct topologies which reflect memory hierarchies and physical layout. No matter what binding map between cores and processes is decided at the MPI application level or in the process placement module, the distance-aware algorithms always provide stable and optimal bandwidth.

B. Discussion

The overhead of our distance-aware framework comes mostly from sorting the edges between processes on the topology information. Performance from Figure 6 and Figure 7 includes the overhead of collecting process placement information and constructing the collective topologies. This overhead of sorting up to thousands of edges is minimal in intra-node cases. However, on a large scale system, it's difficult for these greedy algorithms to scale well with fully-connected graphs. Actually, only directly connected processes are helpful to construct topologies, e.g. father, children, siblings and etc. In future work, we will explore how much process placement information is necessary for each process to construct an optimal or near-optimal topology. A distributed algorithm will be a feasible approach for a large scale system.

There is another question about whether any "distance" information is equally important as another, for a distance-aware collective component. If some of the hardware infor-

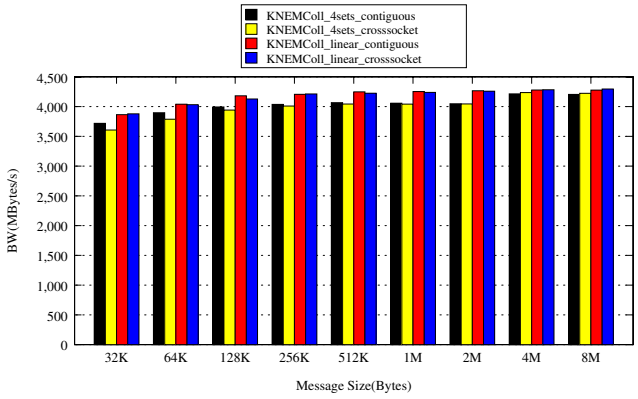

Figure 8. Bandwidth Comparison of KNEM Broadcast on Zoot with 16 processes over two different topologies:4 sets or linear, between 2 binding cases:contiguous case and cross socket case.

mation can be excluded when the internal topology is built, without losing accuracy on the decision, the decision will be greatly simplified on the next generation architectures, which are expected to have deeper memory hierarchies. Figure 8 shows a bandwidth comparison of a KNEM Broadcast on Zoot with 16 processes over two topologies: two levels hierarchical tree with 4 sets and the linear topology. When considering the distance between sockets, which is distance '3' on Zoot case, 16 processes on Zoot's 16 cores are split into 4 sets. Correspondingly, a two level hierarchical tree is built based on these four sets following the approach described in Section IV-B. Ignoring the distance '3' set, a linear topology will be generated making all non-root processes access the root's buffer simultaneously. The pipeline is unnecessary in a linear algorithm because all non-root processes are leaf nodes directly connected to the root node.

From Figure 8, KNEM linear topology outperforms KNEM hierarchical topology. Further splitting into 4 sets according to distance '3' does not help KNEM hierarchical Broadcast on this architecture. The most plausible reason is that, although these 4 sets of processes reside on different physical sockets, they share a single memory controller. If we look at the outcome of a broadcast communication, we can describe it as a certain number of reads from the single input buffer (the buffer at the root process), and a number of writes (flushing back into memory the buffers on each process). As a result, the single memory controller will be overloaded with write requests, and the potential benefit we can get on the read side by taking advantage of memory hierarchies, is totally annihilated. Therefore, splitting the broadcast tree on Zoot does not achieve extra bandwidth but increases the execution path by increasing the depth of the hierarchical tree. So distance '3' is of little importance for large message(bigger than 16KB) broadcast operation on this SMP node. However, small messages are still sensitive to physical distance between sockets, and distance '3' becomes useful again. This clearly shows that the message size is not only important for the selection of the collective algorithm, but also for the selection of how to map this topology on a particular hardware architecture.

More important information can be observed by comparing the Figure 2 presented in Section III with the above mentioned Figure 8. On the same environment, Zoot, the performance of our distance-aware broadcast communication, outperforms both Open MPI and MPICH2 implementations, and is independent of the process placement.

VI. Conclusion and Future Work

The current trend in HPC is toward a large increase in the non-uniformity of a single node, both from the number of cores and the number of memory/cache hierarchies. This leads to more complex architectures, and more challenging scenarios for harnessing the full potential of such environments. As the most widely used HPC software, MPI libraries must have a complete view of process distance to construct optimal topologies to allow collective algorithms to work in an efficient way. This remains true even in the case where the process distribution is optimally decided to match the most common communication pattern in the application.

In this paper, we have presented a collective communication framework combining process distance, underlying hardware architecture, and runtime communicator composition together. We supplemented this framework with two distance-aware collective communications, Broadcast and Allgather, as examples. Moreover, we have demonstrated that our distance-aware collective component provides stable and optimal performance, regardless of process placement, significantly outpacing the state-of-the-art collective algorithms, in both Open MPI and MPICH2 libraries.

In future work, we will extend this distance-aware approach to include other collective communications such as Reduce and Allreduce. We plan to make all Open MPI's collective components distance-aware, not just intra-node communication like KNEM, SM collective, etc, but also clusters of multi-core mixing inter-node and intra-node communication together. To reach this goal, firstly we will extend the information provided by the HWLOC software to include a view of the global process placement, taking into account a simplified view of the network infrastructure. Secondly, we need to trim some unnecessary information making distance-aware framework scalable. A distributed algorithm to figure out direct connections instead of global topologies for each process will be more feasible for large scale systems. A trade-off between optimal topologies and available process placement information will be an interesting discussion.

References

[1] 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. [Online]. Available: http://doi.acm.org/10.1145/1183401.1183451

[2] E. Jeannot and G. Mercier, "Near-optimal placement of mpi processes on hierarchical numa architectures," in *Proceedings of the 16th international Euro-Par conference on Parallel processing: Part II*, ser. Euro-Par'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 199–210. [Online]. Available: http://portal.acm.org/citation.cfm?id=1885276.1885299

[3] L. P. Huse, "Collective communication on dedicated clusters of workstations," in *Proceedings of the 6th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. London, UK: Springer-Verlag, 1999, pp. 469–476. [Online]. Available: http://portal.acm.org/citation.cfm?id=648136.748785

[4] 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.

[5] 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.

[6] J. L. Träff, "Implementing the MPI process topology mechanism," in *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, ser. Supercomputing '02. Los Alamitos, CA, USA: IEEE Computer Society Press, 2002, pp. 1–14. [Online]. Available: http://portal.acm.org/citation.cfm?id=762761.762767

[7] D. Solt, "A profile based approach for topology aware mpi rank placement," http://www.tlc2.uh.edu/hpcc07/Schedule/speakers/hpcc_hp-mpi_solt.ppt.

[8] T. Kielmann, R. F. H. Hofman, H. E. Bal, A. Plaat, and R. A. F. Bhoedjang, "MagPIe: MPI's collective communication operations for clustered wide area systems," *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99)*, vol. 34, no. 8, pp. 131–140, May 1999.

[9] 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.

[10] 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.

[11] 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.

[12] T. Ma, G. Bosilca, A. Bouteiller, and J. J. Dongarra, "Locality and topology aware intra-node communication among multicore CPUs," in *Proceedings of the 17th European MPI users' group meeting conference on Recent advances in the message passing interface*, ser. EuroMPI'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 265–274. [Online]. Available: http://portal.acm.org/citation.cfm?id=1894122.1894158

[13] F. Broquedis, J. Clet Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst, "HWLOC: a Generic Framework for Managing Hardware Affinities in HPC Applications," in *The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, 2010. [Online]. Available: http://hal.inria.fr/inria-00429889/en/

[14] T. Ma, G. Bosilca, A. Bouteiller, B. Goglin, J. Squyres, and J. Dongarra, "Kernel Assisted Collective Intra-node Communication Among Multicore and Manycore CPUs," Research Report, 12 2010. [Online]. Available: http://hal.inria.fr/inria-00544872/en/

[15] 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. [Online]. Available: http://hal.inria.fr/inria-00390064

[16] S. Moreaud, B. Goglin, D. Goodell, and R. Namyst, "Optimizing MPI Communication within large Multicore nodes with Kernel assistance," in *CAC 2010: The 10th Workshop on Communication Architecture for Clusters, held in conjunction with IPDPS 2010*. Atlanta, GA: IEEE Computer Society Press, April 2010.

[17] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. MIT Press and McGraw-Hill, 2001, section 23.2: The algorithms of Kruskal and Prim.

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

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