Topology-Aware Rank Reordering for MPI Collectives

Seyed H. Mirsadeghi and Ahmad Afsahi
ECE Department, Queen's University, Kingston, ON, Canada, K7L 3N6
Email: {s.mirsadeghi, ahmad.afsahi}@queensu.ca

***Abstract*—As we move toward the Exascale era, HPC systems are becoming more complex, introducing increasing levels of heterogeneity in communication channels. This leads to variations in communication performance at different levels of hierarchy within modern HPC systems. Consequently, communicating peers such as MPI processes should be mapped onto the target cores in a topology-aware fashion so as to avoid message transmissions over slower channels. This is especially true for collective communications due to the global nature of their communication patterns and their vast use in many of parallel applications. In this paper, we exploit the rank reordering mechanism of MPI to realize run-time topology awareness for collective communications and in particular MPI_Allgather. To this end, we propose four fine-tuned mapping heuristics for various communication patterns and algorithms commonly used in MPI_Allgather. The heuristics provide a better match between the collective communication pattern and the topology of the target system. Our experimental results with 4096 processes show that MPI rank reordering using the proposed fine-tuned mapping heuristics can provide up to 78% reduction in MPI_Allgather latency at the micro-benchmark level. At the application level, we can achieve up to 34% reduction in execution time. The results also show that the proposed heuristics significantly outperform the Scotch library which provides a general-purpose graph mapping library.**

I. Introduction

As we move toward the Exascale era, the number of nodes in a system as well as the number of cores within each node are rapidly increasing. As a consequence, the physical topology of modern multicore HPC systems is becoming more complex at both the intra- and inter-node layers. We see multi-level hierarchical memory and cache organizations, as well as different network topologies. This translates to increasing levels of heterogeneity in communication channels that leads to various levels of communication performance within the system. In particular, inter-node communications are generally slower than the intra-node communications that use the shared memory. Within the network itself, messages that pass across a larger number of links suffer more in terms of latency. Similar issues exist within modern multicore nodes due to Non-Uniform Memory Access (NUMA) effects and multiple levels of cache hierarchies.

Topology-aware mapping can help to increase communication efficiency in presence of heterogeneous communication channels by exploiting the topology information. Topology information includes both the hardware physical topology and the application process topology which represents the communication pattern of an application. Topology-aware mapping has been shown to be a promising and necessary technique for efficient utilization of communication resources in modern large-scale systems [1]–[4].

Topology-aware mapping can also be geared toward collective communications specifically. Collective communications constitute a major portion of parallel applications' communications and their performance plays a critical role in the performance of applications [5]. Accordingly, various algorithms have been proposed for efficient implementation of collective communications. However, the core merit of such algorithms is only achieved with an appropriate mapping of processes as the performance of a given collective can significantly change under different mappings of processes [6]. In modern large-scale systems a job can initially be mapped in quite a large number of different ways. Resource management tools such as SLURM [7] and Hydra [8] provide various options for choosing the number and order of nodes, sockets, and cores assigned to a job.

Variations of communication performance at different levels of system hierarchies along with the variety of mapping options available at large-scale systems necessitate topology and mapping awareness for collective communications. In recent years, various attempts have been made to bring topology awareness to collective communication design [6], [9]–[11] and the results prove its effectiveness. Our work in this paper is along those lines. We contribute by exploiting MPI rank reordering to realize topology awareness for MPI_Allgather specifically. We propose four fine-tuned mapping heuristics for various communication patterns commonly used in both hierarchical and non-hierarchical MPI_Allgather algorithms. Two of the proposed heuristics can also be used for MPI_Bcast and MPI_Gather operations. Moreover, in our design we take into account both the intra- and inter-node topologies of the target system. For our heuristics, we pursue two important goals: 1) they should be capable of modifying the initial layout of processes so as to closely match it to each collective communication pattern even if the initial mapping is quite far from ideal, and 2) they should not cause performance degradation if the initial layout of processes is already a good match for the target collective communication pattern.

We conduct extensive experiments on a large-scale InfiniBand cluster to evaluate the impact of our proposed heuristics on the performance of MPI_Allgather under different initial mappings of processes. We compare our topology-aware MPI_Allgather against MVAPICH2 and the Scotch [12] mapping library. The results confirm the above-mentioned capabilities for our heuristics and suggest that we can achieve up to 78% and 34% performance improvement at micro-benchmark and application levels, respectively. Moreover, the proposed heuristics are shown to have a better performance, as well as a significantly lower overhead compared to Scotch.

The rest of the paper is organized as follows. In Section II, we review the background material. Section III presents related work. Section IV provides an overview of rank reordering for collective communications. Section V is devoted to rank reordering for MPI_Allgather where we propose our mapping heuristics. Experimental results are provided in Section VI, followed by conclusion and future directions in Section VII.

II. Background

The Message Passing Interface (MPI) [13] is the de facto standard for parallel programming. Communication is realized by explicit movement of data from the address space of one process to another. MPICH [14], MVAPICH [15] and Open MPI [16] are the most popular implementations of MPI. In MPI, *communicators* define the scope and context of all communications. Within a communicator, each process is assigned a *rank* between 0 and $N-1$, with N denoting the number of processes. MPI supports point-to-point, collective and one-sided communications. Collectives, the focus of this paper, involve communications among a group of processes. For instance, in the MPI_Allgather collective operation, data contributed by each process is gathered on all processes.

Collectives are often designed and implemented as a series of point-to-point communications scheduled over a sequence of stages. In this regard, there exist various approaches and algorithms for each collective operation. In practice, MPI libraries exploit a combination of such algorithms and choose one based on various parameters such as message and communicator size. In addition, collective implementations can be classified into hierarchical and non-hierarchical approaches.

Traditionally, collective algorithms have been designed in a non-hierarchical approach for flat systems and executed across all processes in the system. Recursive Doubling and Ring [17] are two of the most commonly used algorithms for MPI_Allgather, in this regard. Recursive doubling consists of $\log_2 N$ stages where N denotes the total number of processes in the communicator. At each stage s, where $s = 0, 1, \ldots, \log_2 N - 1$, rank i exchanges data with rank $i \oplus 2^s$, where $\oplus$ represents the binary XOR operator. Consequently, each rank such as i exchanges messages with ranks $i \oplus 1, i \oplus 2, i \oplus 4, \ldots, i \oplus \frac{n}{2}$ throughout the stages of the algorithm. Fig. 1 shows an example with 8 processes where the three communication stages have been distinguished from each other by three different colors. The numbers on each edge denote the specific stage numbers. For instance, rank 0 and 1 exchange their input buffers in stage 0. In stage 1, rank 0 and 2 exchange their input buffers as well as the data they received in the previous stage, and so on. Note that the volume of the exchanged messages is doubled at each stage of the algorithm. It is also worth mentioning that recursive doubling is mainly used for a power-of-two number of processes.

In the ring algorithm, all processes are organized into a logical ring in order of their ranks. At each stage, rank i receives a message from rank $i-1$ and sends a message to rank $i+1$. In the first stage, rank i sends its own data to rank $i+1$. In the following stages rank i forwards to rank $i+1$ the data it received from rank $i-1$ in the previous stage. With N processes, the algorithm runs for $N-1$ stages.

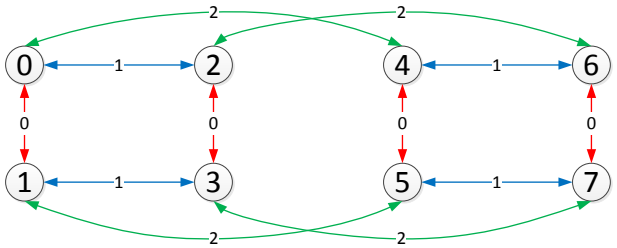


Fig. 1. The recursive doubling communication pattern with 8 processes. There exist 3 communication stages: stage 0 (red), stage 1 (blue), and stage 2 (green).

Hierarchical algorithms are performed across the nodes/sockets rather than the processes to take advantage of hierarchical communication costs. Typically, for each node a communicator is created to contain all the processes residing on that node. One process on each node is selected as the leader of that node. The allgather algorithm could then proceed in three phases: 1) gathering intra-node messages into node leaders, 2) exchanging the gathered data between all the node leaders, using a recursive doubling/ring-based allgather algorithm, and 3) broadcasting data from the leaders to intra-node ranks. The gather and broadcast in phase 1 and 3 might use a direct linear pattern or an indirect non-linear algorithmic design. In the linear design, all ranks directly send (receive) data to (from) the root of gather (broadcast), whereas in the non-linear design, message transmissions follow a particular communication pattern used by the underlying algorithm. For instance, binomial tree is one of the well-known communication patterns used for both gather and broadcast in the non-linear design.

III. Related Work

Almási et al. [18] propose machine-optimized versions of MPICH2 collective communications for IBM BG/L systems. Sack and Gropp [11] present non-minimal versions of the recursive-doubling and bucket algorithms to decrease the amount of congestion and exploit multiport communications within fat-tree and 3D torus networks. Unlike our work, [11] still ignores how the processes have been mapped onto the system. Moreover, they only consider the network topology and assume there is only one process on each node. Closest to what we propose in this paper is the work by Subramoni et al. in [10]. They use rank reordering to design a network-aware broadcast algorithm. However, they only consider the network topology for improving the inter-node component of a hierarchical broadcast algorithm and hence do not take into account the intra-node topology of the target system. In this work we consider both the hierarchical and non-hierarchical allgather approaches and in each case take into account the system topology at both the intra- and inter-node layers. In addition, we propose fine-tuned mapping heuristics for four of the major communication patterns that are commonly used in MPI_Allgather, whereas [10] uses the classic DFT/BFT algorithms for the binomial tree communication pattern only.

Li et al. [9] propose three shared-memory NUMA-aware algorithms for the intra-node component of MPI_Allreduce. The algorithms are targeted for a thread-based implementation of MPI ranks and attempt to minimize inter-socket data transfers

within each node. Ma et al. [6] focus on optimization of the intra-node component of collective communications based on various levels of hierarchies within each nodes. Two algorithms are proposed for building broadcast trees and allgather logical rings based on the physical distances among processes. The algorithms have also been used in [19] for the optimization of hierarchical collectives. However, the main focus in [19] is to increase the overlap between the intra- and inter-node phases of hierarchical collectives by taking advantage of single copy mechanisms in OS kernel. Faraj et al. [20] propose an algorithm for building contention-free allgather rings among nodes in switched clusters. The ring is built by a DFS traversal of a spanning tree within the network topology. Intra-node topology is not considered. Subramoni et al. [21] propose network topology-aware communication schedules to reduce congestion for all-to-all FFT operations in InfiniBand clusters.

IV. Rank Reordering for Collectives

Topology awareness can be applied to collectives in two general ways: a) by changing the communication pattern of the underlying algorithms, or b) by changing the mapping of processes. Our work in this paper falls under the second category. In particular, we keep collective algorithms intact, and reorder the ranks with respect to a mapping from the collective communication pattern to the physical topology of system. Generally speaking, the objective of the mapping is to decrease the physical distance among the communicating processes, specially those that communicate more often and/or use larger messages. For the physical topology, we extract the distances among all cores using the hwloc library [22] and InfiniBand [23] tools for the intra- and inter-node distances respectively. Note that in this work, we are not concerned about the access privileges for using such tools. We assume physical distances are extracted once, and saved for future references.

To do the mapping, various approaches can be used. For instance, one could use an external graph mapping library such as Scotch [12] which employs graph partitioning techniques to recursively partition and map a given *guest* graph (communication pattern) onto a *host* graph (physical topology). Alternatively, fine-tuned heuristics can be devised for each collective communication pattern so to achieve better mapping results, and avoid some of the overheads (such as creating process topology graphs). In Section V, we propose four of such heuristics for various communication patterns commonly used in MPI_Allgather. Two of the proposed heuristics can also be used for MPI_Bcast and MPI_Gather operations.

Having found the mapping, a new communicator is created with process ranks reordered based on the mapping results. The whole process can be repeated to create reordered communicators for each desired collective communication pattern. Later on, any subsequent calls to the corresponding collective in an application will be conducted over the reordered copy of the given communicator. It should be noted that a similar framework has been used in [10] to design a network topology-aware broadcast algorithm. However, as stated before, they only consider the network topology of the target system. It is also worth mentioning that the whole rank reordering process happens only once at run-time. In addition, we could also use an `info` key to allow the programmer to enable/disable the whole approach for each communicator separately.

V. Rank Reordering for MPI_Allgather

In general forms of topology-aware mapping, the communication pattern changes from one application to another. Therefore, mapping heuristics should be general enough to handle various arbitrary communication patterns. To this end, the communication pattern is provided to the mapping algorithm in terms of a weighted graph which is later on traversed (usually by a greedy approach) to figure out the desired mapping. For the allgather operation we improve this general approach from two aspects[1]. First, for a particular collective such as allgather, the communication patterns are fixed and hence there is no need to stick to one general mapping algorithm for all of them. Instead, we could utilize fine-tuned heuristics for each pattern so as to achieve better mapping results. Second, with fine-tuned heuristics, it is not required to build a process topology graph to describe the communication pattern. Communication patterns can be systematically derived from the corresponding algorithms used by the MPI library. Thus, we can jump right to the mapping step. However, with a general mapping library such as Scotch, we still need to build the collective topology graph first.

A. Mapping heuristics

Algorithm 1 illustrates the general scheme of our proposed mapping heuristics for MPI_Allgather. The algorithm outputs a mapping array M representing the new rank for each process. In the first step and without loss of generality, the process with rank 0 is fixed on the core already hosting it. Next, in steps 3 to 8, we iteratively choose a new process and find a target core for hosting it. The target core is always chosen with respect to what we call a *reference core*. The reference core is a core that has already been assigned to one of the processes in some previous iteration of the algorithm. The target core is always chosen to be a free core having a minimum distance from the reference core. If more than one core satisfy this condition, one of them is chosen randomly. Having mapped the new process, the reference core *might* be updated in step 7.

The key steps in algorithm 1 are steps 4 and 7. Step 4 determines the order in which we select the processes for mapping, whereas the choice of the reference core in step 7 implicitly designates the group of processes that the selected process will be placed closer to. The strategies used in these two steps depend on the underlying communication pattern among the processes and hence, vary with each particular allgather algorithm. That is why in step 7 of Algorithm 1 we say the reference core is updated *if necessary*. In the following, we will propose a specific version of Algorithm 1 for different communication patterns commonly used in allgather. In particular, we cover recursive doubling, ring, binomial tree broadcast, and binomial tree gather. It is important to note that hereinafter, we interchangeably use process ranks to refer to a particular process or the core hosting it.

1) Recursive doubling: At each stage of the recursive doubling algorithm, processes communicate in pairs and the communications happening at further stages of the algorithm represent larger messages. Therefore, in our heuristic for recursive doubling we will try to choose the new process based

[1]In fact, with an appropriate set of mapping heuristics, the improvements apply to other collectives as well.

Algorithm 1: General scheme of our mapping heuristics

Input : Number of processes p, Physical topology distance matrix D
Output: Mapping array M representing the new rank for each process

1 Fix rank 0 on its current core;
2 Choose 0 as the reference core;
3 **while** *there exist more processes to map* **do**
4 Select a new process to map;
5 Find a target core among the free cores. The target core is a core that has not already been assigned to any other process and has a minimum distance from the reference core;
6 Map the new process onto the target core;
7 Update the reference core with the target core *if necessary*;
8 **end**

on the communications of further stages as much as possible. Accordingly, we start with the pairs of communications that fall in the last stage. The first process selected for mapping (after rank 0) will be the one that exchanges messages with rank 0 in the last stage. We know this would be rank $0 \oplus \frac{p}{2}$, where p denotes the total number of processes and $\oplus$ represents the binary XOR operator. Obviously, rank 0 will be our reference core since it is the only process that has already been mapped. Thus, in the first iteration, rank $0 \oplus \frac{p}{2}$ is mapped as close as possible to rank 0. For the next process, we have two options: 1) choosing with respect to rank 0, or 2) choosing with respect to the recently mapped process, i.e., rank $0 \oplus \frac{p}{2}$. Sticking with rank 0 as our reference core, we choose the new process based on the communications of rank 0 in the second-to-last stage of recursive doubling. We know that rank 0 communicates with rank $0 \oplus \frac{p}{4}$ in that stage. Thus, rank $0 \oplus \frac{p}{4}$ will be the new process and is mapped as close as possible to rank 0. For the next new process, we will again have two options: 1) choosing with respect to rank 0, or 2) choosing with respect to the recently mapped processes, i.e., rank $0 \oplus \frac{p}{4}$ or rank $0 \oplus \frac{p}{2}$. This time, we choose the new process with respect to rank $0 \oplus \frac{p}{4}$ and update our reference core with rank $0 \oplus \frac{p}{4}$ accordingly. The reason is two-fold:

1) If we want to choose the next process with respect to rank 0 or rank $0 \oplus \frac{p}{2}$, then we have to choose from communications of the third-to-last or second-to-last stages respectively. However, with rank $0 \oplus \frac{p}{4}$ we can choose from the communications that belong to the last stage (using larger messages).
2) The process that exchanges data with rank $0 \oplus \frac{p}{4}$ in the last stage, also communicates with rank $0 \oplus \frac{p}{2}$ (already mapped) in the second-to-last stage. Thus, it communicates with a larger number of processes that have already been mapped.

With rank $0 \oplus \frac{p}{4}$ as our new reference core, we repeat the above procedure in the next two iterations so as to map two more new processes before updating the reference core again. Algorithm 2 depicts our proposed mapping heuristic (RDMH) designed based on the above discussion. Lines 5-8 correspond to step 4 of Algorithm 1. Starting with $i = p/2$ in line 3, RDMH gives a higher priority to those ranks that communicate with the reference core in further stages of recursive doubling. With the loop in lines 5-7, new ranks are chosen from an earlier stage only if the ones corresponding to further stages have already been mapped. Lines 11-14 correspond to step 7 in Algorithm 1. At the end of each iteration, the reference core is updated with the new process *only if* two processes have already been mapped with respect to the current reference core.

It is worth mentioning that MVAPICH2 also exploits some sort of rank reordering for recursive doubling. However, it is limited to a specific layout of processes only. There is no actual mapping heuristic or topology awareness in the sense that we discuss in here. The rank reordering in MVAPICH2 just changes a block initial layout of processes to a cyclic one. With our heuristic, we *calculate* an appropriate mapping by taking into account the initial mapping of processes as well as the physical topology of the system at both the intra- and inter-node layers.

Algorithm 2: RDMH - Mapping heuristic for the recursive doubling communication pattern

Input : Number of processes p, Physical topology distance matrix D
Output: Mapping array M representing the new rank for each process

1 M[0] = 0 ; // Fix rank 0 on its current core
2 ref_rank = 0 ; // Choose 0 as the reference core
3 i = p / 2 ; // Starting from the last stage
4 **while** *there exist more processes to map* **do**
5 **while** *ref_rank* $\oplus$ *i is already mapped* **do**
6 i = i / 2;
7 **end**
8 new_rank = ref_rank $\oplus$ i;
9 target_core = find_closest_to(ref_rank, D) ; // Find the free core closest to the reference core
10 M[new_rank] = target_core ; // Map the new process onto the target core
11 **if** *already mapped two processes with respect to ref_rank* **then**
12 ref_rank = new_rank ; // Update reference core
13 i = p / 2 ; // Restarting from the last stage
14 **end**
15 **end**

2) Ring: Our heuristic for ring (RMH) is quite straightforward. This is mainly because in the ring algorithm, each process communicates with only one other process that is fixed across all the stages of the algorithm. For ring, Algorithm 3 chooses the processes for mapping in a simple increasing order of their ranks. Moreover, the reference core is updated with the new process at every iteration of the mapping algorithm. Having fixed rank 0, RMH maps rank 1 as close as possible to 0. Rank 1 would then be the reference core with rank 2 being the next process for mapping. Rank 2 is mapped as close as possible to rank 1 and is set to be the new reference core. This procedure is repeated until all processes are mapped.

Algorithm 3: RMH - Mapping heuristic for the ring communication pattern

Input : Number of processes p, Physical topology distance matrix D
Output: Mapping array M representing the new rank for each process

1 M[0] = 0 ; // Fix rank 0 on its current core
2 ref_rank = 0 ; // Choose 0 as the reference core
3 **while** *there exist more processes to map* **do**
4 new_rank = (ref_rank + 1) % p;
5 target_core = find_closest_to(ref_rank, D);
6 M[new_rank] = target_core;
7 ref_rank = new_rank ; // Update the reference core
8 **end**

3) Binomial broadcast: We cover binomial tree since it could be used in the final broadcast phase of a hierarchical allgather. However, the proposed heuristic can also be used for

MPI_Bcast. Binomial broadcast has the advantage of having a fixed message size across all the stages of algorithm. Therefore, we do not need to worry about the size of communicated messages in the mapping heuristic. Hence, it would be enough to traverse the tree in some way and map the nodes that we come across as close to each other as possible. For example, we could use a pure Depth-First Traversal (DFT) or Breadth-First Traversal (BFT) algorithm. However, we are interested to know if there are any particular traversing fashions that could heuristically represent better candidates.

A potential approach is to traverse the tree so as to visit the nodes with larger subtrees sooner than others. Each node is then mapped as close as possible to its corresponding parent node in the binomial tree. This way, a higher priority (for choosing the next process for mapping) is given to those ranks on which a higher number of other ranks depend for receiving the message. This is essentially same as the rationale used in [10] for designing a network-aware broadcast algorithm. A second approach that we propose here is a variation of DFT which unlike the previous approach, visits the nodes with smaller subtrees first. This way, higher priority is given to communications that happen at further stages of binomial broadcast. The rationale behind is the fact that as we move toward the final stages of a binomial broadcast, the number of pair-wise communications increase. With p ranks, there is only one communication in the first stage, whereas in the last stage we will have $\frac{p}{2}$ communications. Therefore, communications in further stages are more likely to create contention. Hence, we want to map their corresponding ranks closer to each other.

In this paper, we will use the second traversal approach. Algorithm 4 (BBMH) depicts the details of our mapping heuristic for binomial broadcast. Without loss of generality we assume rank 0 is the root of broadcast. The mapping is done in a recursive fashion by initially calling the recursive Procedure `RecBinomialMap` with rank 0 as its argument. The loop in lines 4-10 determines how we choose a new rank for mapping with respect to each reference core r, and thus corresponds to step 4 in Algorithm 1. The conditions of the loop come from the underlying structure of the binomial tree and make sure that the new process represents a valid child of the reference core in the tree. The reference core is updated with the new rank by recursively calling `RecBinomialMap` in line 8, representing step 7 in Algorithm 1. It is worth mentioning that for medium and large messages, broadcast is commonly implemented by a scatter-allgather algorithm [17]. However, we do not propose a separate mapping heuristic for it since RDMH and RMH already cover the allgather phase, and the scatter phase is covered by the proposed heuristic for gather.

4) Binomial gather: Gather constitutes the first phase of the hierarchical allgtaher, and the binomial tree is again a major pattern used for it. Although the communication pattern is binomial again, the size of exchanged messages is not fixed any more and is increased as we get closer to the root of the tree. Accordingly, we want to pick the heaviest edge of the tree each time, and map its unmapped endpoint as close as possible to the mapped one. The reason is two-fold. First, this way we will be choosing a rank which communicates with some ranks that have already been mapped. Second, it represents a communication that uses larger messages. This is similar to the rationale used by Hoefler and Snir [3] in their

Algorithm 4: BBMH - Mapping heuristic for the binomial broadcast communication pattern

```
   Input : Number of processes p, Physical topology distance matrix D
   Output: Mapping array M representing the new rank for each process
1  M[0] = 0 ; // Fix rank 0 on its current core
2  Rec_binomial_map(0) ; // Calling the recursive
   mapping function with rank 0
1  Procedure RecBinomialMap(rank r)
2      ref_rank = r ; // Choose r as the reference core
3      i = 1;
4      while (ref_rank ∧ i == 0) and (i ≤ p / 2) do
5          new_rank = ref_rank + i;
6          target_core = find_closest_to(ref_rank, D);
7          M[new_rank] = target_core;
8          Rec_binomial_map(new_rank) ; // Calling the
           algorithm recursively for the new rank
9          i = i * 2 ; // Move on to the next child
10     end
```

greedy mapping heuristic for the general case of topology-aware mapping. Our proposed heuristic is different in the sense that we extract the communication pattern in a closed-form fashion without building any process topology graph. In other words, we systematically find the heaviest edges and their corresponding reference cores at each step.

Algorithm 5 illustrates the details of our mapping heuristic (BGMH) for binomial gather. Without loss of generality, we assume rank 0 is the root of gather. Line 12 and the loop in line 5 determine how the new process is chosen with respect to each reference core and hence, correspond to step 4 in Algorithm 1. The reference core is updated in each iteration of the loop in line 6, and thus is representative of step 7 in Algorithm 1. For each value of i, BGMH iterates over all potential reference cores in $\mathcal{V}$ and maps a new process as close as possible to each. In addition, every newly mapped rank is added to the set of potential reference cores in line 10. It is worth mentioning that BGMH can also be used for MPI_Gather.

Algorithm 5: BGMH - Mapping heuristic for the binomial gather communication pattern

```
   Input : Number of processes p, Physical topology distance matrix D
   Output: Mapping array M representing the new rank for each process
1  M[0] = 0 ; // Fix rank 0 on its current core
2  Initialize V ; // The set of potential reference cores
3  V ← 0 ; // Start with rank 0
4  i = p / 2;
5  while i > 0 do
6      for ref_rank ∈ V and ref_rank + i < p do
7          new_rank = ref_rank + i;
8          target_core = find_closest_to(ref_rank, D);
9          M[new_rank] = target_core;
10         V ← new_rank;
11     end
12     i = i / 2;
13 end
```

B. Preserving the correct order of the output buffer

At the end of an allgather operation, each process will have an output vector that holds the individual messages corresponding to every other process. The elements of this vector should appear in a correct order, i.e., in the order of the process ranks to which they initially belonged. Each underlying

allgather algorithm is designed so as to preserve such a correct order of the output buffer. However, rank reordering can disrupt the desired order because as soon as we change the rank of a given process from i to j, it will act as the process with rank j while actually having the input vector corresponding to rank i. In the following, we explain two different approaches for addressing this issue. Sack and Gropp also use similar techniques in [11] in their distance-halving recursive doubling algorithm.

1) Extra initial communications at the beginning: In the first approach, we use extra send and receive communications to move the data among the processes with respect to rank changes. This is done before the allgather algorithm issues any communications. For instance, if the process with rank 0 is going to be reordered to rank 1, we will have rank 1 send its input vector to rank 0. In addition, rank 0 will send its input vector to the process that is going to be reordered to 0. This way, the input vector of all the processes will be in correspondence to their new ranks and hence, the output vector will be in correct order.

2) Memory shuffling at the end: In the second approach, we do not exchange the input buffers and let the allgather operation proceed as usual. However, we shuffle the output buffer elements at the end based on how the process ranks have been reordered. For instance, if rank 0 has been changed to rank 1, then we know that the element at index 1 in the output buffer actually belongs to rank 0 and hence should be moved to the head of the output buffer.

It should be noted that in practice we use the above mechanisms in only two of allgather algorithms; recursive doubling and the binomial gather. For the ring and the binomial broadcast algorithms we will not have any extra overheads in terms of preserving the correct order of the output vector. The output vector order does not apply to a broadcast operation as the output buffer is not a vector any more. Moreover, in the ring algorithm we resolve the issue from within the algorithm itself. At each stage of the algorithm, we simply store the incoming message at the correct index of the output vector. This can easily be done for the ring algorithm because every process receives only one individual message at each stage for which we can figure out the correct offset in the output vector based on the mapping array.

VI. Experimental Results

We conduct all the experiments on the GPC cluster at the SciNet HPC Consortium. GPC consists of 3780 nodes with a total of 30240 cores. Each node has two quad-core Intel Xeon sockets operating at 2.53GHz. Each socket forms a NUMA node with 8GB of local memory and 8MB of L3 cache. Approximately one quarter of the nodes are interconnected with non-blocking DDR InfiniBand, while the rest of nodes are connected with 5:1 blocked QDR InfiniBand. For our experiments, we only use the nodes with QDR InfiniBand. These nodes are interconnected via Mellanox ConnectX 40Gb/s InfiniBand adapters. The network topology is a fat-tree consisting of two 324-port core switches and 103 36-port leaf switches. Fig. 2 shows the corresponding details. The numbers on the links represent the number of links that connect two switches. In particular, each leaf switch is connected to 30 compute nodes, and has 3 uplinks to each of the two core switches. Each core switch itself is in fact a 2-layer fat-tree consisting of 18 *line* and 9 *spine* switches (all 36-port). Each line switch is connected to 6 leaf switches, and also has 2 uplinks to each of the 9 spine switches. Finally, the nodes run Centos 6.4 along with Mellanox OFED-1.5.3-3.0.0, and we use MVAPICH2-2.0, Scotch-6.0.0, and hwloc-1.8.1.

A. Micro-benchmark Results

We use the OSU Micro-Benchmarks [24] to measure the latency of MPI_Allgather with and without rank reordering. The results report the percentage of performance improvement over the default algorithms used in MVAPICH2. We also compare the performance of our proposed mapping heuristics against Scotch [12] which provides a general-purpose graph partitioning and mapping library. With 4096 processes, we only report the results for a maximum message size of 256KB due to memory restrictions on each node. In all figures, 'initComm' and 'endShfl' respectively refer to extra initial communications and memory shuffling at the end for preserving the correct order of the output buffer. Moreover, 'Hrstc' represents the results for our proposed heuristics.

We perform the experiments for four different initial mappings of processes: *block-bunch*, *block-scatter*, *cyclic-bunch*, and *cyclic-scatter*. We choose these four so as to verify the impacts of our heuristics under four different (and well-known) initial mappings. Note that the gain in performance by our heuristics (as well as any other rank reordering scheme) depends on how far the initial mapping is from the ideal mapping for each collective communication pattern. In the block mapping, adjacent ranks are mapped onto the same node as far as possible before moving to any other node. Similarly, a bunch mapping binds the adjacent ranks to the cores of the same socket inside each node. On the contrary, cyclic mapping distributes adjacent ranks across the nodes in a round-robin fashion. The scatter mapping uses a similar round-robin scheme to scatter ranks across all the sockets within each node.

1) Non-hierarchical approach: Fig 3 shows the results for the non-hierarchical approach. As shown in Fig. 3(a), we can achieve up to about 67% improvement for messages smaller than 1KB with our proposed mapping heuristics. The improvements correspond to RDMH as MVAPICH2 uses recursive doubling in this range of message sizes. We also see that Scotch performs quite poorly in this interval with the corresponding mapping being worse than the initial one. Moreover, for RDMH (up to 1KB), the improvement is increasing with message size as larger messages are affected more by adverse effects of congestion. Another observation is the better performance achieved by extra initial communications compared to memory shuffling.

Fig. 3(a) also shows that for messages larger than 1KB we cannot achieve any performance improvement. This is because MVAPICH2 uses the ring algorithm in this range of message sizes and the initial block-bunch mapping is already the best match. However, unlike Scotch, our heuristic (RMH) does not cause any performance degradation. Also, Fig. 3(b) shows that with a block-scatter initial mapping, RHM can provide about 50% improvement for messages above 1KB as the intra-node scatter mapping is not a good match for the ring algorithm.

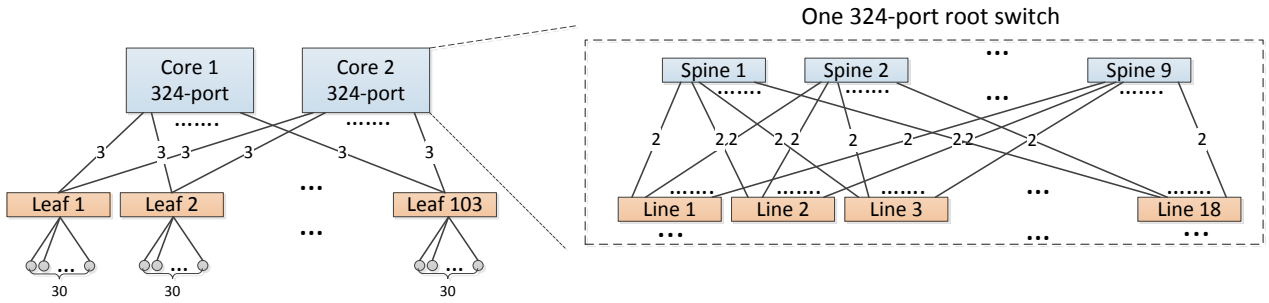
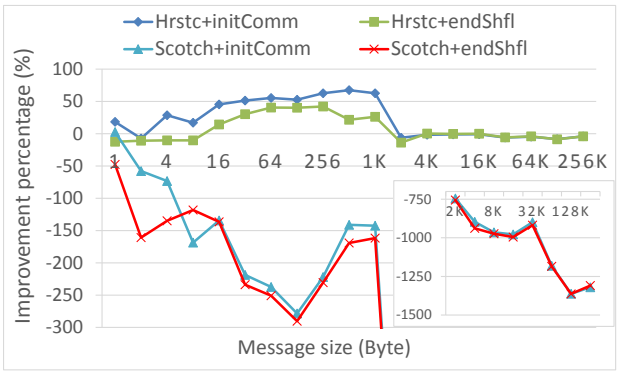
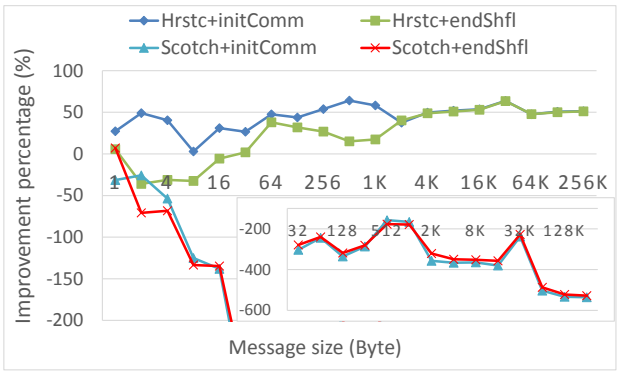


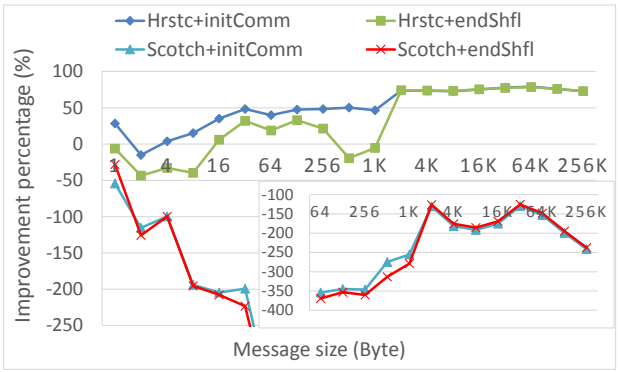
Fig. 2. The network topology of the GPC cluster at SciNet. The topology is a fat-tree consisting of 103 leaf switches and two core switches.



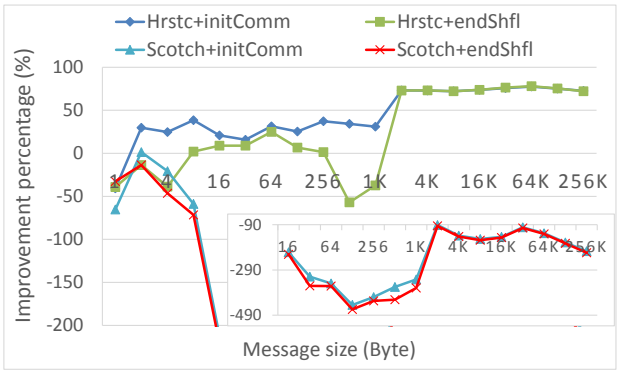
(a) block-bunch



(b) block-scatter



(c) cyclic-bunch



(d) cyclic-scatter

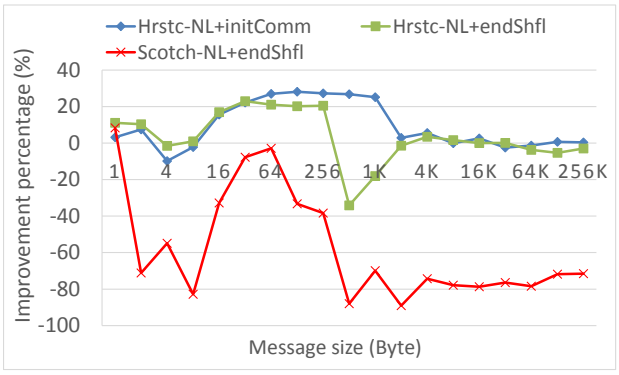
Fig. 3. Micro-benchmark performance improvements for non-hierarchical topology-aware allgather with four different initial mappings and 4096 processes.

Again, we see that Scotch performs poorly. We can also see that for messages above 1KB, initComm and endShfl result in a same performance as we actually do not use/need any of these mechanisms for the ring algorithm (see section V-B).

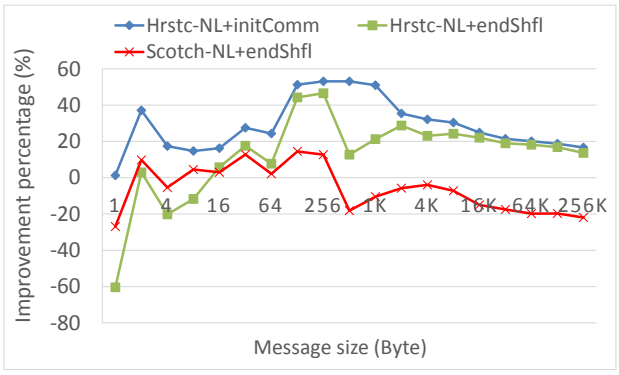
Fig. 3(c) and 3(d) show that with an initial cyclic-bunch or cyclic-scatter mapping, our heuristic can provide up to 50% and 78% improvement for messages below and above 1KB respectively. We also see that memory shuffling overheads can become quite costly for 512B and 1KB message sizes. Moreover, we see a higher improvement for messages above 1KB compared to Fig. 3(a) and 3(b). This is mainly because an initial cyclic (scatter) mapping along with the underlying ring algorithm result in higher congestion across network (QPI) links. At the same time we see a relatively lower improvement for messages below 1KB in Fig. 3(d) compared to Fig. 3(a) and 3(b). This is because an initial cyclic (scatter) mapping is better than block (bunch) for the recursive doubling algorithm. Thus, we see that a poor initial mapping for one algorithm can be relatively better for another. This encourages to employ run-time rank reordering for collective communications.

2) Hierarchical approach: With a hierarchical allgather, we only consider the two block-bunch and block-scatter initial mappings[2]. Also, we use 'L' and 'NL' suffixes to respectively refer to linear and non-linear intra-node broadcast/gather phases. Fig. 4 shows that the improvements are generally lower for the hierarchical algorithms. This is because a hierarchical approach per se provides a level of topology awareness by

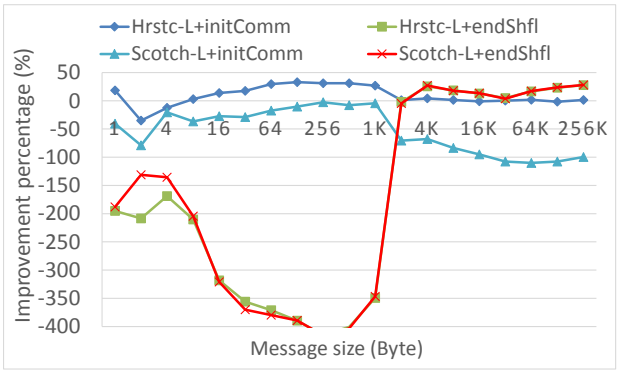
[2]Hierarchical allgather is not supported with cyclic mapping. Also, cyclic results in similar mapping to block for intra- and inter-node communicators.



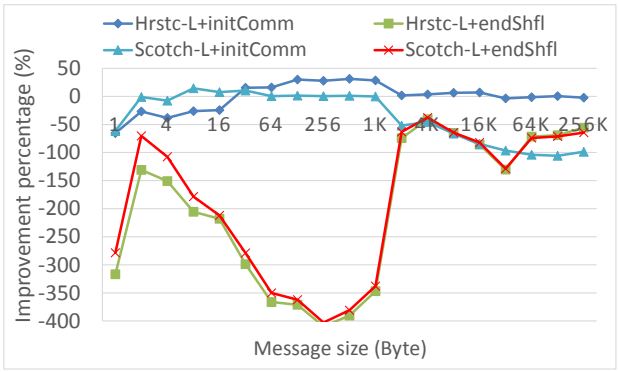
(a) block-bunch, non-linear



(b) block-scatter, non-linear



(c) block-bunch, linear



(d) block-scatter, linear

Fig. 4. Micro-benchmark performance improvements for hierarchical topology-aware allgather with two different initial mappings and 4096 processes.

restricting the inter-node communications to node-leaders only. Moreover, with a hierarchical approach, rank reordering is used at a smaller scale as it is applied to node-leaders and local processes separately. We can also see that Scotch performs relatively better with the hierarchical approach compared to the non-hierarchical one.

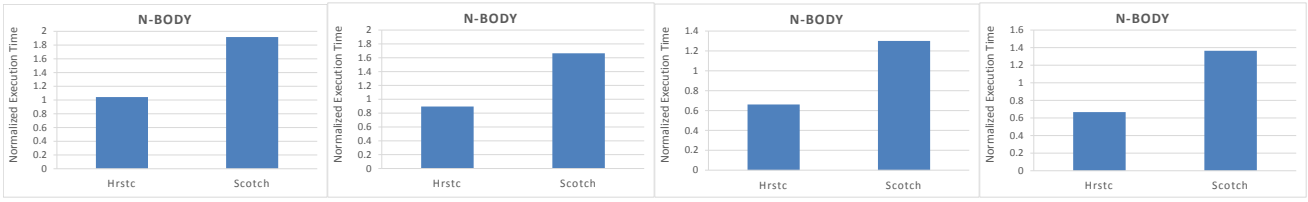
Fig. 4(a) and 4(b) show the results for a hierarchical allgather with a non-linear intra-node broadcast/gather. In particular, Fig. 4(b) shows that for messages above 1KB, we can attain up to 30% improvement over a block-scatter mapping. This improvement comes from the gather phase within each node. The BGMH algorithm reorders the ranks so that large-message communications of the intra-node binomial gather fall within a single socket, and thus benefit from a higher bandwidth. However, we see a decreasing improvement with increase in message size which is due to the overheads induced by the extra communications or memory shuffling. We think the intra-node impacts of our algorithms could be better studied in a system with higher number of cores per node.

Fig. 4(c) and 4(d) show the results for hierarchical allgather with linear intra-node broadcast/gather. Note that in this case, we cannot have any rank reordering at the intra-node level as there is no particular pattern to optimize the mapping for; all the processes directly communicate with the root process. Thus, there is less room for improvements. However, for messages below 1KB, we can still improve the performance up to 30% with the initComm version of our heuristic (RDMH). As expected, we cannot achieve any improvement for messages above 1KB because the initial block mapping is already a good match for the underlying ring algorithm. Fig. 4(c) and 4(d) also show that for messages below 1KB, the performance is quite poor for the endShfl version of both Scotch and our heuristics. This is mainly because in this case, the overheads of memory shuffling are too high that nullify the potential benefits of a better mapping. Note that memory shuffling in this case is done over combined (larger) messages gathered from all ranks within a node. Another observation is the improvement for messages larger than 1KB. We do not know where exactly this improvement comes from and need to investigate it further.

B. *Application Results*

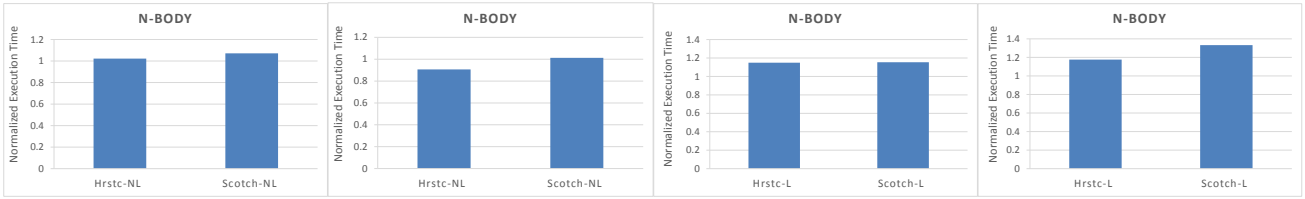
In this section, we evaluate the efficiency of our topology-aware rank reordering for N-BODY [25] as a real application. We measure the execution time of N-BODY with and without our rank-reordered versions of MPI_Allgather. Our profiling results with 1024 processes show that N-BODY makes 1358 calls to MPI_Allgather, making it a potentially good candidate to benefit from rank reordering. We report the results for 1024 processes as we could not scale the application to a larger number of ranks due to memory restrictions. Moreover, we only use extra initial communications for preserving the correct order of the output buffer as it was shown to outperform memory shuffling in the micro-benchmark section. In all figures, the results represent the average of 3 runs of the application.

Fig. 5 shows the normalized execution times of N-BODY with respect to the default non-hierarchical approaches used



(a) block-bunch (b) block-scatter (c) cyclic-bunch (d) cyclic-scatter

Fig. 5. Application execution time improvements for non-hierarchical topology-aware allgather with four different initial mappings and 1024 processes.



(a) block-bunch, non-linear (b) block-scatter, non-linear (c) block-bunch, linear (d) block-scatter, linear

Fig. 6. Application performance improvements for hierarchical topology-aware allgather with two different initial mappings and 1024 processes.

in MVAPICH2. As shown in Fig. 5(a), for the block-bunch mapping, our heuristics result in the same execution time as the default mapping. Again, this is because the initial mapping is already close to the optimal in this case. Scotch on the other hand causes an almost 2-fold increase in execution time. In fact, it can be seen that in all cases shown in Fig. 5, Scotch exacerbates the execution time of the application. With a block-scatter mapping (Fig. 5(b)), we can achieve about 10% reduction in execution time with our proposed heuristics. The highest improvement is seen in Fig. 5(c) and 5(d), where we can achieve about 30% reduction in execution time with our proposed rank reordering algorithms.

Fig. 6 shows the results with respect to the default hierarchical approach. In particular, Fig. 6(a) shows that with a non-linear intra-node pattern, we cannot see any improvement over the block-bunch mapping. For block-scatter however, we could achieve about 10% improvement with our heuristics. In addition, Fig. 6(c) and 6(d) show that rank reordering could not improve execution time of the application when a linear pattern is used for intra-node broadcast/gather. This is an expected behavior as the combination of a block mapping at the inter-node layer and linear intra-node patterns highly restrict the opportunity to benefit from rank reordering. We even see a slight increase in execution time which is not expected and requires further investigations to find the root cause of it.

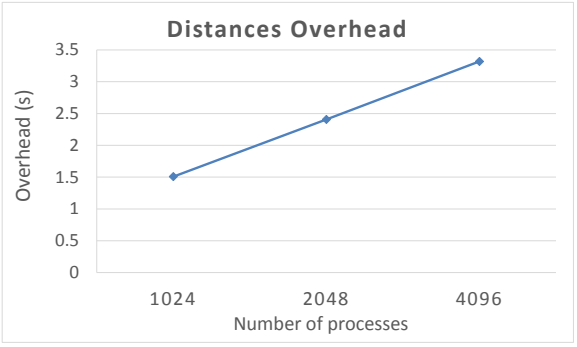
C. Overheads Analysis

In this section, we evaluate the total overhead of topology-aware rank reordering with our mapping heuristics, and compare it to Scotch. The two main components of total overhead are the time required for finding physical distances, and the time spent by the mapping algorithm. Finding the physical distances is a one-time overhead whereas we will have the mapping overhead once for each communication pattern. Our measurements show that our heuristics have almost the same amount of overhead. Therefore, we avoid reporting the overhead results for each one of them separately.

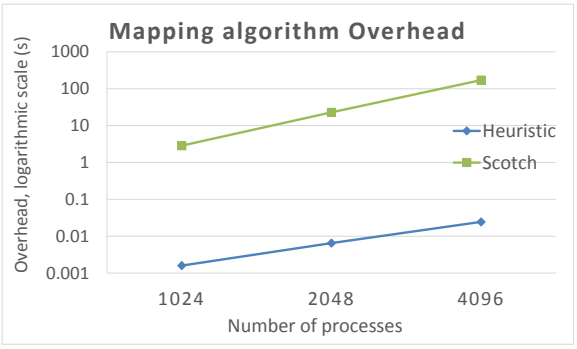
Fig. 7(a) shows the overhead of finding physical distances for three different number of processes. We can see a linear scaling as the number of processes increase. With 4096 ranks, it takes about 3.3 seconds to extract all distances. This overhead is similarly applied to both Scotch and our proposed heuristics. However, it is a one-time overhead which is *not* repeated every time that we perform rank reordering for a new pattern. Fig. 7(b) shows the time spent by the mapping algorithm itself once the distances have been extracted. Note the logarithmic scale (base 10) used for the vertical time axis. As shown, with our proposed heuristics we can achieve a significantly lower overhead as well as a much better scaling compared to Scotch. This is due to the simpler design of our heuristics, and the fact that unlike Scotch, we do not need to build a process topology graph for any of our mapping heuristics. With 4096 ranks, the overhead of our heuristics is about 24 ms. Also, with 1024 ranks, the total overhead including the physical distance extraction is less than 1.6 seconds which represents less than 4% of the total execution time of N-BODY with the same number of processes.

VII. Conclusion and Future Work

In this paper, we exploited rank reordering to make MPI_Allgather topology aware. In particular, we proposed four fine-tuned mapping heuristics for various communication patterns used in allgather. Experimental results showed that topology-aware rank reordering with our heuristics can provide considerable performance improvements at micro-benchmark and application levels. It was also shown that the proposed heuristics deliver a higher performance and a significantly lower overhead compared to a general mapping library such as Scotch. As for future work, we intend to extend our heuristics to other allgather algorithms such as Bruck [17] as well as other important collectives such as MPI_Allreduce. We also seek to evaluate the performance of our binomial broadcast and gather heuristics on systems having a more complicated intra-node topology with a larger number of cores per node.



(a) Overhead for extracting physical distances



(b) Time spent by the mapping algorithm

Fig. 7. Overheads of rank reordering for 1024, 2048, and 4096 processes.

Finally, devising an adaptive version of our proposed approach is another interesting venue for future research in which a runtime component is used to decide whether to use the reordered communicator for a given collective or not based on the potential performance improvements that each heuristic can provide for various message sizes.

ACKNOWLEDGMENTS

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada Grant #RGPIN/238964-2011, Canada Foundation for Innovation and Ontario Innovation Trust Grant #7154. Computations were performed on the GPC supercomputer at the SciNet HPC Consortium. SciNet is funded by: the Canada Foundation for Innovation under the auspices of Compute Canada; the Government of Ontario; Ontario Research Fund - Research Excellence; and the University of Toronto. We would like to especially thank Scott Northrup for his technical support regarding our experiments on GPC. We also thank Mellanox Technologies and the HPC Advisory Council for the resources to conduct the early evaluation of this research.

REFERENCES

[1] P. Balaji, R. Gupta, A. Vishnu, and P. Beckman, "Mapping communication layouts to network hardware characteristics on massive-scale blue gene systems," *Computer Science-Research and Development*, vol. 26, no. 3-4, pp. 247–256, 2011.

[2] M. J. Rashti, J. Green, P. Balaji, A. Afsahi, and W. Gropp, "Multi-core and network aware MPI topology functions," in *Proc. EuroMPI*, 2011, pp. 50–60.

[3] T. Hoefler and M. Snir, "Generic topology mapping strategies for large-scale parallel architectures," in *Proc. Int. Conf. on Supercomputing*, 2011, pp. 75–84.

[4] A. Bhatelé, G. R. Gupta, L. V. Kalé, and I.-H. Chung, "Automated mapping of regular communication graphs on mesh interconnects," in *Proc. Int. Conf. on High Performance Computing*, 2010, pp. 1–10.

[5] R. Rabenseifner, "Automatic MPI counter profiling of all users: First results on a CRAY T3E 900-512," in *Proc. message passing interface developers and users Conf.*, 1999, pp. 77–85.

[6] T. Ma, T. Herault, G. Bosilca, and J. J. Dongarra, "Process distance-aware adaptive MPI collective communications," in *Proc. Int. Conf. on Cluster Computing*, 2011, pp. 196–204.

[7] A. B. Yoo, M. A. Jette, and M. Grondona, "SLURM: Simple linux utility for resource management," in *Proc. Int. Workshop on Job Scheduling Strategies for Parallel Processing*, 2003, pp. 44–60.

[8] "The Hydra process management framework, http://wiki.mpich.org/mpich/index.php, last accessed 2015/12/21."

[9] S. Li, T. Hoefler, and M. Snir, "NUMA-aware shared-memory collective communication for MPI," in *Proc. Int. Symp. on High-Performance Parallel and Distributed Computing*, 2013, pp. 85–96.

[10] H. Subramoni, K. C. Kandalla, J. Vienne, S. Sur, B. Barth, K. Tomko, R. Mclay, K. Schulz, and D. K. Panda, "Design and evaluation of network topology-/speed- aware broadcast algorithms for InfiniBand clusters," in *Proc. Int. Conf. on Cluster Computing*, 2011, pp. 317–325.

[11] P. Sack and W. Gropp, "Faster topology-aware collective algorithms through non-minimal communication," in *Proc. ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 2012, pp. 45–54.

[12] F. Pellegrini and J. Roman, "Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs," in *Proc. Int. Conf. on High-Performance Computing and Networking*, 1996, pp. 493–498.

[13] "Message Passing Interface Forum, http://www.mpi-forum.org/, last accessed 2015/12/21."

[14] "MPICH: High-performance and widely portable MPI implementation, http://http://www.mpich.org/, last accessed 2015/12/21."

[15] "MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE, http://mvapich.cse.ohio-state.edu/, last accessed 2015/12/21."

[16] "Open MPI: Open Source High Performance Computing, http://www.open-mpi.org/, last accessed 2015/12/21."

[17] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in MPICH," *Int. Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 49–66, 2005.

[18] G. Almási, P. Heidelberger, C. J. Archer, X. Martorell, C. C. Erway, J. E. . Moreira, B. Steinmacher-Burow, and Y. Zheng, "Optimization of MPI collective communication on BlueGene/L systems," in *Proc. Int. Conf. on Supercomputing*, 2005, pp. 253–262.

[19] T. Ma, G. Bosilca, A. Bouteiller, and J. Dongarra, "HierKNEM: an adaptive framework for kernel-assisted and topology-aware collective communications on many-core clusters," in *Proc. Int. Symp. on Parallel & Distributed Processing*, 2012, pp. 970–982.

[20] A. Faraj, P. Patarasuk, and X. Yuan, "Bandwidth efficient all-to-all broadcast on switched clusters," *International Journal of Parallel Programming*, vol. 36, no. 4, pp. 426–453, 2008.

[21] H. Subramoni, K. Kandalla, J. Jose, K. Tomko, K. Schulz, D. Pekurovsky, and D. K. Panda, "Designing topology-aware communication schedules for alltoall operations in large infiniband clusters," in *Proc. Int. Conf. on Parallel Processing (ICPP)*, 2014, pp. 231–240.

[22] 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 *Proc. Euromicro Int. Conf. on Parallel, Distributed and Network-Based Processing*, 2010, pp. 180–186.

[23] InfiniBand Trade Association, "InfiniBand Architecture Specification," http://www.infinibandta.org/, last accessed 2015/12/21.

[24] "The OSU Micro-Benchmarks, http://mvapich.cse.ohio-state.edu/benchmarks/, last accessed 2015/12/21."

[25] H. Shan, J. P. Singh, L. Oliker, and R. Biswas, "Message passing and shared address space parallelism on an SMP cluster," *Parallel Computing*, vol. 29, no. 2, pp. 167–186, 2003.