Multi-core and Network Aware MPI Topology Functions

Mohammad Javad Rashti[1], Jonathan Green[1], Pavan Balaji[2], Ahmad Afsahi[1], and William Gropp[3]

[1] Queen's University, Kingston, ON, Canada
[2] Argonne National Laboratory, Argonne, IL, USA
[3] University of Illinois at Urbana-Champaign, IL, USA

Abstract. MPI standard offers a set of topology-aware interfaces that can be used to construct graph and Cartesian topologies for MPI applications. These interfaces have been mostly used for topology construction and not for performance improvement. To optimize the performance, in this paper we use graph embedding and node/network architecture discovery modules to match the communication topology of the applications to the physical topology of multi-core clusters with multi-level networks. Micro-benchmark results show considerable improvement in communication performance when using weighted and network-aware mapping. We also show that the implementation can improve communication and execution time of the applications.

Keywords: MPI, virtual topology, physical topology, multi-core, network.

1 Introduction

With the emerging many-core architectures and high performance interconnects offering more parallelism and performance, clusters are expected to move towards exascales in the next few years [1]. Such systems are becoming increasingly hierarchical in their node architecture and interconnection network. Communication at various hierarchies demonstrate different performance levels. It is therefore critical for the communication libraries to efficiently handle the communication demands of High Performance Computing (HPC) applications on such hierarchical systems.

Message Passing Interface (MPI) [2] is the predominant messaging standard for HPC applications. MPI provides a set of interfaces that are designed to assist the library to construct a virtual topology out of the application's communication pattern, and to support remapping (reordering) the processes to the available cores in a way that optimizes performance. However, most MPI libraries merely provide a trivial implementation of these functions and lack the support for the remapping feature. In this work, we have designed the MPI non-distributed topology functions (MPI_Graph_create and MPI_Cart_create) for efficient process remapping over hierarchical clusters. We have integrated the node physical topology with network architecture and used graph embedding tools inside MPI library to override the current trivial implementation of the topology functions and efficiently reorder the initial process mapping. We have evaluated our implementation on two different InfiniBand [3] clusters using micro-benchmarks and MPI applications. Micro-benchmark results show up to 60% communication time improvement for Cartesian topologies with data exchange between the neighbors. The application results show up to 48% communication time improvement, and up to 26% runtime improvement.

The rest of this paper is organized as follows. We briefly describe the MPI topology functions and the motivation behind this work in Section 2. Related work is covered in Section 3. Section 4 discusses our design and implementation. Section 5 presents the experimental results, and Section 6 concludes the paper.

2 Background and Motivation

MPI defines a set of virtual-topology definition functions for graph and Cartesian structures [2]. MPI_Graph_create and MPI_Cart_create are collective calls that accept a virtual topology and return a new MPI communicator enclosing the desired topology. If the user opts for reordering, the function may reorder the process ranks for an efficient process-to-core mapping. The topology accepted by these functions is in a non-distributed form, meaning that all nodes have a full view of the entire structure and pass the whole information to the function. Recently, distributed graph topology functionality has been added to the MPI standard [2] to support large-scale systems. In these functions, each node has a limited neighborhood view of the graph, and all processes collectively construct the virtual topology in a distributed fashion.

Although process topology functionality is not new to the MPI standard, HPC applications that utilize such functionality use them mainly for the constructed virtual topology (e.g., a Cartesian topology). Thus, the ability of this interface to support process reordering for better communication performance has widely remained unutilized, mostly because MPI implementations merely construct the virtual topology, and have no process remapping for performance improvement. In this paper, we focus on the design and implementation of MPI non-distributed topology functions to improve the performance of the applications. We will cover the distributed topology functions in a future work.

3 Related Work

There has been some past research on topology-aware communication in MPI. The authors in [4] look at the algorithms for mapping virtual topology to physical topology in MPI using a hierarchical tree structure to represent the hardware topology. This work defines a cost function that is the sum of the communication costs over the links. This paper presents an implementation for a specific machine (an HP server), with architecture similar to multi-core machines.

The work in [5] presents a set of graph embedding algorithms with hardware topology represented using a hierarchical tree similar to [4], but with a more comprehensive mathematical analysis for different architectures. A major contribution is the optimization of two different cost functions. The first function is the sum of the communication costs over the interconnection links, and the second is a load balancing function that minimizes the number of expensive communications by any one process. This work experiments with a set of NEC SX-6 machines, each with up to eight shared-memory vector processors, connected through a proprietary network.

The author in [6] argues that the MPI topology-aware functionality at the time lacks precision and accuracy. The paper suggests an extension to MPI where weighted communication graphs can be used in order to produce a better solution when using the topology functionality. The paper also suggests extensions to allow dynamic process reordering. No implementation of the suggested functionality is attempted.

The authors in [7] summarize the work in [5] and [6], suggesting some additional changes to MPI as in [6]. The paper also shows that a non-trivial implementation of the MPI topology functions can provide great performance gains on SMP systems.

In [8], functionality is implemented to map weighted communication graphs to weighted node architecture graphs using *Scotch* graph partitioning software [9]. The communication graph weights are chosen based on the total communication volume. Unlike our work in this paper, the work in [8] does not use or implement MPI topology functions. It rather calculates the mapping outside MPI, before the application is started. It also does not consider network hierarchy.

In [10], the authors propose *TreeMatch* algorithm to calculate a near-optimal mapping of processes to resources on a NUMA cluster. Similar to the work in [8], this paper is concentrated on node architecture and does not consider network hierarchy. It does not use or implement MPI topology functions either. It rather calculates the mapping using MPICH2 process manager and hwloc [14]. The paper presents MPI results using simulation and NAS benchmarks on a 4 NUMA-node 96-core machine.

Recently, the authors in [11] have explained the distributed topology functions in MPI 2.2 and discussed possible methods for implementing them in the future. In [12] the authors propose an automated framework to detect regular communication patterns in applications. The framework finds the dimensions of a possibly regular pattern and maps it to mesh/torus processor topologies.

4 Design and Implementation of MPI Topology Functions

4.1 Design of the Graph Topology Function

Our design of both MPI graph and Cartesian topology functions is based on an underlying graph structure. We also use graphs to represent both virtual and physical topologies inside the MPI implementation. Using graphs at the underlying layer, we can use static mapping of virtual to physical topology graphs in order to find the sub-optimal mapping of processes to processor cores.

Virtual topology is constructed as a graph in which vertices represent processes and links represent the existence (or significance) of the inter-process communications. We use the normalized total communication volume between two processes as the metric for communication significance. MPI_Cart_create and MPI_Graph_create functions do not support weighted edges, meaning there is no differentiation among edges of the virtual topology. This is a critical shortcoming, since it is usual that the communication between some pairs is more significant than others. To realize how supporting weighted graphs can increase the effectiveness of process reordering, we use edge replication in MPI_Graph_create input to account for weighted edges. This approach, although not much scalable, can support realistic non-uniform communication patterns.

The graph representation for the cluster's physical topology consists of two distinct but integrated parts: node architecture and network architecture. The node architecture graph includes weighted edges that represent the communication performance between any two cores. We assume higher communication performance between the cores with closer proximity. Our representation of the network comprises network distances between any two cores. The network architecture includes weighted graph edges that represent the communication performance of the network path between the nodes on which the cores reside.

4.2 Implementation of Topology Functions

In this section, we present details about the implementation of the MPI_Graph_create and MPI_Cart_create functions inside MVAPICH2 [13]. We use the Scotch graph processing library [9] to map virtual to physical topology graphs in MPI_Graph_create. The library defines an undirected source graph, which can represent the virtual topology. Each vertex and edge of the source graph is weighted, to account for the computation and communication weight of the corresponding process and link, respectively. Similarly, the target machine architecture is represented by an undirected architecture target graph with weights for vertices and edges representing the processing power of the processor and communication performance of the link, respectively [9].

The *hardware locality* (hwloc) library [14] provides a portable abstraction of the underlying machine architecture. It detects architectural components of the nodes such as processor sockets, cores, caches, memory, SMT and NUMA architecture. The architecture is represented as a tree, with nodes at the top level and logical cores at the leaves. This library can assist MPI to construct the physical topology of the machine.

To have a complete view of the cluster's physical architecture and go beyond a flat network assumption, we add a network discovery part in our physical architecture discovery. This module extracts the network distance between any two nodes in the cluster and merges that information with the node architecture extracted by hwloc. In this paper, our network discovery module uses InfiniBand subnet manager [3] tools (i.e., *ibtracert* utility) to discover the network distance between Infiniband nodes.

4.2.1 Implementation of Graph Topology Function. To supply MPI with the application virtual topology, we extract the exact amounts of data transfer between processes by profiling the applications using probes inside the MPI library. We give the highest weight to the maximum pairwise communication volume. The normalized weights range from 1 to 10. We use edge replication to represent edge weights. MPI_Graph_create calls the Scotch library if the user opts for reordering. Scotch builds a weighted graph out of the user-supplied graph. The topology table of a server node is also created using hwloc at each process. The communication performance between two logical cores on a node is calculated based on the depth of their common ancestor in the node topology tree. For example, if two cores reside on different sockets, their common ancestor will be the node itself, with the lowest depth in the tree, translating into the lowest communication performance.

The process with rank zero (in MPI_COMM_WORLD) will perform a network discovery operation using *ibtracert* to extract the distance between any two InfiniBand nodes. This information is then scattered to other processes to be integrated into the node architecture to have the full physical topology architecture. Topology graph edge loads (input to Scotch) represent the performance of the communication path between the connecting vertices. Network distance, defined as the number of hops between two nodes, is used to calculate the physical topology edge loads. The farthest nodes get the load of 1 on their graph edge. The closest nodes get the maximum network hop count as their edge load. The intra-node graph loads are calculated in a way that are always larger than the closest network distance, indicating the fact that intra-node communication is cheaper than inter-node communication.

Fig. 1 shows an example of how graph loads are assigned based on the system and network architecture. *N1-N4* are multi-core (here, 2-way quad-core) nodes, connected through three switches (*S1-S3*) in a tree-like network. In this architecture, *d1* (the path between two cores of the same socket) will have the highest load value in the graph, while the path between *N2* and *N3* (*d4*) will have the lowest load value, indicating the lowest performance path in the network. Thus we have: *d1 > d2 > d3 > d4 = 1.*

4.2.2 Cartesian Topology Implementation. Since the Scotch library does not support Cartesian topology, we internally convert the MPI_Cart_create topology to a graph and use MPI_Graph_create for reordering. In this conversion, the user view of the Cartesian topology remains intact. Cartesian topologies do not specify weights for graphs. Therefore, the converted topology will be a non-weighted graph.

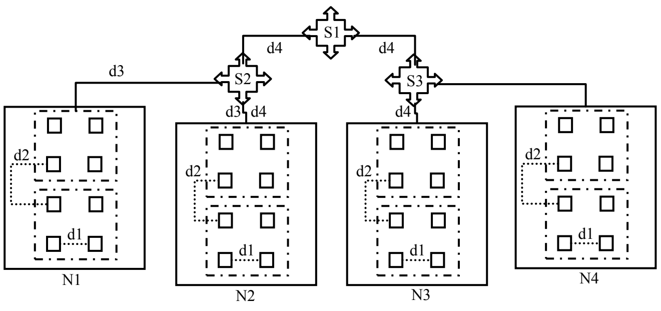


Fig. 1. An example of physical topology distances

5 Experimental Results

We have conducted our experiments on two clusters. The first cluster has four servers, each with two quad-core 2GHz AMD Opteron 2350 processors (a total of 32 cores). The servers have 512KB L2 cache per core, 2MB shared L3 cache per processor and 8GB memory. They are interconnected through Mellanox ConnectX InfiniBand cards [15] via three Mellanox InfiniBand switches, similar to Fig. 1. The nodes run Linux Fedora 12 kernel version 2.6.31. The machines use OFED 1.5.2 to access the InfiniBand network. For the second cluster, we have 16 servers, each with two hexa-core 3GHz Intel Xeon X5670 processors (a total of 192 cores). There is a 12MB multi-level cache per processor, and 24GB memory per machine. The servers use Mellanox ConnectX2 InfiniBand cards. Eight servers are connected to a Mellanox InfiniBand switch, and the remaining servers are connected to another switch. Each of these switches is connected to two upper layer InfiniBand switches. The machines run Redhat Enterprise Linux 5 with kernel version 2.6.18-194. The nodes use OFED version 1.5.2. On both clusters, we use MVAPICH2 1.5 [13] as our code-base.

5.1 Micro-benchmark Results

We start our evaluation with two micro-benchmarks that put processes in Cartesian arrangements such as Torus and Hypercube. The tests are run on the first (32-core) cluster. The first micro-benchmark (*Cartesian-model exchange*) constructs a 2D/3D torus or a 5D hypercube. Each process runs 1000 iterations, each with a computation followed by exchanging messages with the neighbors in all dimensions. We report the average iteration time. We find the virtual-topology graph of each test by profiling it in an initial run. The graph is then supplied to the same program as input for MPI_Graph_create. To evaluate the effect of graph weights, we carry significantly heavier communication on one of the dimensions.

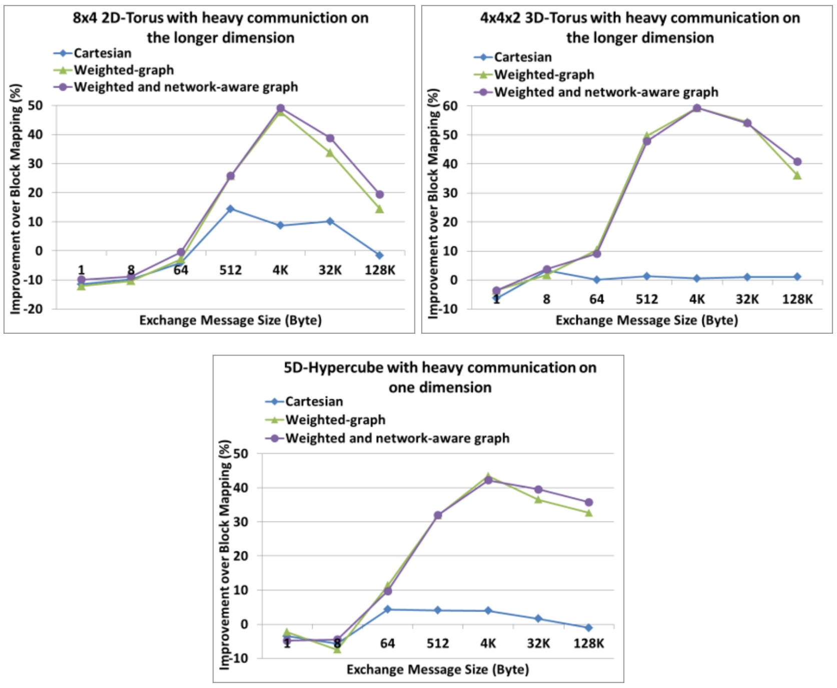
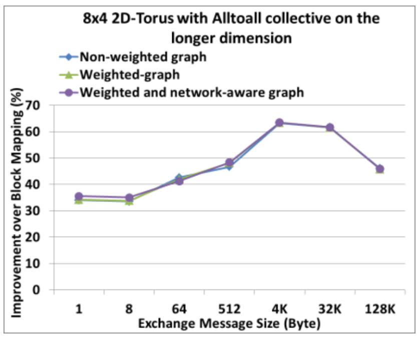


Fig. 2. Runtime improvement of topology-aware mapping over block mapping for 2D-torus, 3D-torus and 5D-hypercube in the Cartesian-model exchange micro-benchmark

Fig. 2 shows the improvement of the topology-aware mapping compared to *block* mapping for an 8×4 2D-torus, a 4×4×2 3D-torus and a 5D-hypercube. The processes communicate more heavily on one dimension (the longer dimension for torus). For the micro-benchmarks, to particularly show the effect of network-aware mapping, we put two subsequent nodes on different switches. As shown in Fig. 2, for all topologies the weighted graph shows significant improvement compared to the block mapping. It is because with a weighted graph the library differentiates between heavier and lighter dimensions and will try to map the processes of the heavier dimension on one node to take advantage of shared-memory performance. On the other hand, the non-weighted Cartesian topology does not make any differentiation between different dimensions. Consequently, it may map the heavier dimension across the network leading to worse performance. The other observation is the improvement when using network-aware mapping, especially for larger message. It shows that even when the difference in network distance is as little as two switches we can observe some improvement.

The second micro-benchmark (*dimensional collectives*) examines the case where processes form a Cartesian arrangement and perform collective communications on one dimension of the topology. We arrange 32 processes in an 8×4 2D-torus and engage them in MPI_Alltoall collective operations on the longer dimension. Fig. 3 clearly shows the improvement when using topology-aware mapping. The reason for on par performance between weighted graph and non-weighted graph is that the communication is done only on one dimension; therefore both graphs result in the same mapping. The network-aware mapping also shows similar improvement, because all eight processes of the collective dimension are mapped to the same node.

Fig. 3 also shows the results for a 16×2 2D-torus. These results are reported since the longer dimension (on which collectives are performed) has the length of 16, which does not fit into one node, therefore network communication is inevitable even after topology-aware remapping. As the results suggest, network-aware mapping shows improvement compared to other graph mappings, especially for larger message sizes.



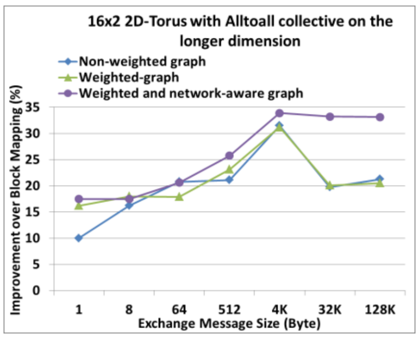


Fig. 3. Runtime improvement of topology-aware mapping over block mapping for 2D-torus in the dimensional collective (Alltoal) micro-benchmark

5.2 Evaluation Results for MPI Applications

To see the effect of our implementation on MPI applications, we have adapted some MPI application benchmarks (NAS and LAMMPS) to use MPI graph topology function. We profiled the original applications to discover their virtual topology graph in order to supply them to MPI_Graph_create function. In LAMMPS, processes communicate in a 2D-torus. Processes in CG.32 also form a logical 8×4 2D-torus, however they do not always follow the torus links for communication. Processes in MG.32 communicate in the form of a reordered 5D-hypercube. To be consistent, regardless of application's logical structure, we always use a graph topology function.

Fig. 4 shows the improvement of the topology-aware mapping for these applications compared to block and cyclic mappings on the 32-core cluster. This excludes the time in MPI_Graph_create to create the communicator. For most applications, the benefit over cyclic mapping is considerable while there is less benefit over block mapping. This is because processes mostly communicate with their adjacent ranks, thus the ideal mapping is close to block mapping. In LAMMPS (especially the *friction* workload), where the communication volume is not equal on different links, we can see more improvement with weighted and network-aware graphs, compared to non-weighted graph where sometimes it is even worse than the block mapping. We are currently investigating the reasons behind the poor performance of the weighted graph results for LAMMPS-Couple.

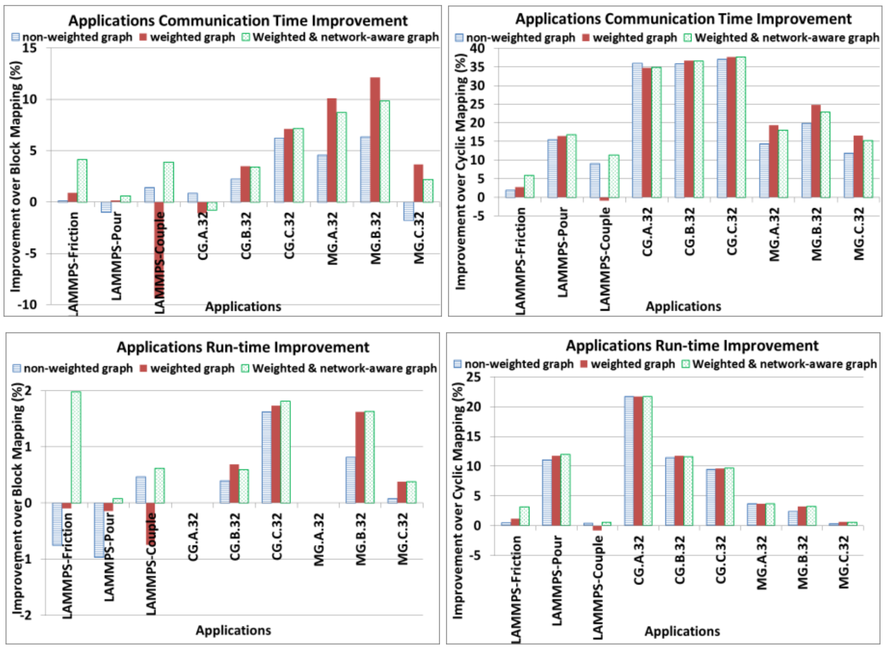


Fig. 4. Application communication time and runtime improvement of topology-aware mapping over block and cyclic mappings on the 32-core cluster

To show the benefits of our design on a larger test bed, we have also evaluated our work on a second cluster with 16 nodes (using 8 cores per node, for a total of 128-cores) using some of the application workloads. The results are presented in Fig. 5.

In Fig. 5 we observe more improvement for most of the workloads, which indicates the performance scalability with the machine size. Higher improvement in the 128-core case for some workloads such as LAMMPS-*friction* is partially because some of the neighbors that would fall on the same node in block mapping in 32-core case fall on different nodes in 128-core case. Therefore reordering is more effective for the latter case. LAMMPS-*couple* shows a considerable difference in communication pattern between 32-core and 128-core cases. While for 32 cores, processes communicate to their neighbors almost symmetrically, the pattern becomes asymmetric in 128-core case (a process mostly communicates with two partners). Thus we see higher difference between non-weighted and weighted/network-aware results for 128 cores. Such behavior is also observed for *pour* workload to some extent, leading to more improvement compared to block mapping.

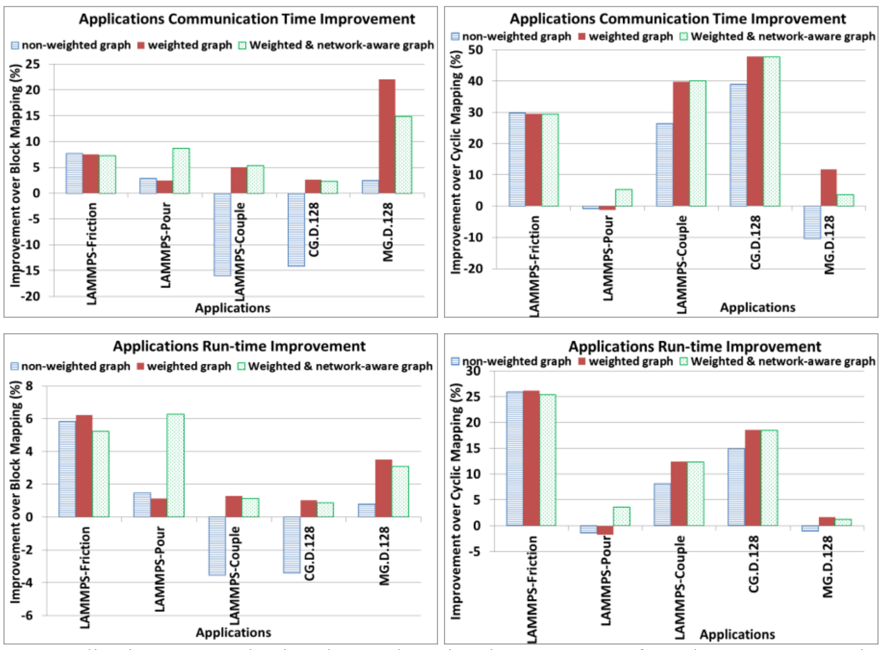


Fig 5. Application communication time and runtime improvement of topology-aware mapping over block and cyclic mappings on the 128-core cluster

5.3 Implementation Overhead

Our implementation of MPI_Graph_create imposes an overhead compared to the trivial implementation. Table 1 shows the approximate overhead and its scalability in LAMMPS application. This one-time overhead is amortized in application runtime.

Table 1- Time to create the communicator in MPI_Graph_create for LAMMPS

System	*Job size (#processes)*	*Trivial (ms)*	*Non-weighted Graph (ms)*	*Weighted Graph (ms)*	*Network-aware Graph (ms)*
Cluster 1	**8**	0.3	7.3	7.3	7.9
	16	0.3	7.6	7.7	8.1
	32	0.5	8.6	8.7	9
Cluster 2	**128**	5.1	31.3	31.7	31.7

6 Conclusions and Future Work

In this paper, we presented design and implementation of MPI non-distributed graph and Cartesian functions in MVAPICH2 for multi-core nodes connected through multi-level InfiniBand networks. The Cartesian-model micro-benchmarks show that the effect of reordering process ranks can be significant, and when the communication is heavier on one dimension the benefits of using weighted and network-aware graphs (instead of non-weighted graph / Cartesian functions) are considerable. We also modified some MPI applications with MPI_Graph_create. The evaluation results show that MPI applications can benefit from topology-aware MPI_Graph_create.

As for the future work, we intend to evaluate the effect of topology awareness on other MPI applications, design a more general communication cost/weight model for graph mapping, and design and implement MPI distributed topology functions for more scalability.

Acknowledgments. This work is supported in part by Natural Sciences and Engineering Research Council of Canada, Canada Foundation for Innovation, Ontario innovation Trust, U.S. Department of Energy and National Science Foundation. We thank Mellanox Technologies and HPC Advisory Council for the resources.

References

1. IESP: International Exascale Software Project, http://www.exascale.org/
2. MPI Forum: MPI: A Message-Passing Interface Standard, version 2.2, September 2009.
3. IBTA: InfiniBand Architecture Specification, 2007, http://www.infinibandta.org/
4. Hatazaki, T.: Rank reordering strategy for MPI topology creation functions. In: 5th Euro PVM/MPI Conference, pp. 188-195, Springger-Verlag, London, UK (1998)
5. Träff, J. L.: Implementing the MPI Process Topology Mechanism. In: ACM/IEEE Conference on Supercomputing, pp. 1-14, IEEE CS, Los Alamitos (2002)
6. Träff, J. L.: SMP-aware Message Passing Programming. In: 17th IEEE Parallel and Distributed Processing Symposium (IPDPS), pp. 56, IEEE CS, Washington (2003)
7. Berti, G., Träff, J. L.: What MPI could (and cannot) do for Mesh-partitioning on Non-homogeneous Networks. In: 13th Euro PVM/MPI Conference, LNCS 4192, pp. 293-302, Springer-Verlag, Berlin (2006)
8. Mercier, G., Clet-Ortega, J.: Towards an Efficient Process Placement Policy for MPI Applications in Multi-core Environments. In: 16th Euro PVM/MPI Conference, LNCS 5759, pp. 104-115, Springer-Verlag, Berlin (2009)
9. Pellegrini, F.: Scotch and libScotch 5.1 User's Guide. Bacchus team, INRIA Bordeaux Sud-Ouest, https://gforge.inria.fr/docman/view.php/248/5709/scotch_user5.1.pdf (2010).
10. Jeannot, E., Mercier, G.: Near-Optimal Placement of MPI processes on Hierarchical NUMA Architectures. Proceedings of EuroPar 2010 conference, Italy (2010)
11. Hoefler, T., Rabenseifner, R., Ritzdorf, H., Supinski, B. R. de, Thakur, R., Träff, J. L.: The Scalable Process Topology Interface of MPI 2.2. J. Concurr. Comp.-Pract. E., Vol. 23, Nr. 4, 293-310, John Wiley & Sons, Ltd. (2010).
12. Bhatele, A., Gupta, G., Kale, L. V., Chun, I. H.: Automated Mapping of Regular Communication Graphs on Mesh Interconnects. In 17th International Conference on High Performance Computing (HiPC), IEEE CS, Washington (2010)
13. MVAPICH: MPI Over InfiniBand, 10GigE/iWARP and RoCE, http://mvapich.cse.ohio-state.edu/
14. Broquedis, F., Clet-Ortega, J., Moreaud, S., Furmento, N., Goglin, B., Mercier, G., Thibault, S., and Namyst, R.: hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications. In Proceedings of the 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP'10), Italy (2010)
15. Mellanox Technologies, http://www.mellanox.com/