Design of a Scalable InfiniBand Topology Service to Enable Network-Topology-Aware Placement of Processes*

H. Subramoni[1], S. Potluri[1], K. Kandalla[1], B. Barth[3], J. Vienne[1], J. Keasler[4], K. Tomko[2], K. Schulz[3], A. Moody[4], and D. K. Panda[1]

[1] *Department of Computer Science and Engineering, The Ohio State University*
{subramon, potluri, kandalla, viennej, panda}@cse.ohio-state.edu

[2] *Ohio Supercomputing Center, Columbus, Ohio*
{ktomko}@osc.edu

[3] *Texas Advanced Computing Center, Austin, Texas*
{bbarth, karl}@tacc.utexas.edu

[4] *Lawrence Livermore National Laboratory, Livermore, California*
{keasler, moody20}@llnl.gov

***Abstract*—Over the last decade, InfiniBand has become an increasingly popular interconnect for deploying modern supercomputing systems. However, there exists no detection service that can discover the underlying network topology in a scalable manner and expose this information to runtime libraries and users of the high performance computing systems in a convenient way. In this paper, we design a novel and scalable method to detect the InfiniBand network topology by using Neighbor-Joining techniques (NJ). To the best of our knowledge, this is the first instance where the neighbor joining algorithm has been applied to solve the problem of detecting InfiniBand network topology. We also design a network-topology-aware MPI library that takes advantage of the network topology service. The library places processes taking part in the MPI job in a network-topology-aware manner with the dual aim of increasing intra-node communication and reducing the long distance inter-node communication across the InfiniBand fabric.**

I. Introduction and Motivation

Across scientific domains, application scientists are constantly looking to push the performance envelope by running increasingly larger jobs on supercomputing systems. Supercomputing systems are currently comprised of thousands of compute nodes based on modern multi-core architectures. Interconnection networks have also rapidly evolved to offer low latencies and high bandwidths to meet the communication requirements of parallel applications. InfiniBand has emerged as a popular high performance network interconnect and is being used increasingly to deploy some of the top supercomputing installations around the world. Large scale supercomputing systems are organized as racks of compute nodes and use complex network architectures ranging from fat-trees to tori. The interconnect fabric is typically comprised of leaf switches and many levels of spine switches. Each traversal (hop) of a switch between two end points increases the message latency, and Table I shows the impact of alternate hop counts on the MPI communication latency as measured on the TACC Ranger system under quiescent conditions. Even under such ideal conditions, we observe that the latencies of the 5-hop inter-node exchanges are almost 81% higher than that of intra-rack exchanges. However, supercomputing systems typically have several hundreds of jobs running in parallel and it is common for the network to be congested, further affecting the communication latency. In this context, it is extremely critical to design communication libraries in a network-topology-aware manner.

TABLE I
MPI Communication Performance Across Varying Levels of Switch Topology on TACC Ranger (courtesy [1])

Process Location	Number of Hops	MPI Latency (*us*)
Intra-Rack	1 Hop in Leaf Switch	1.57
Inter-Rack	3 Hops Across Spine Switch	2.45
	5 Hops Across Spine Switch	2.85

The problem of efficiently detecting the network topology and leveraging this information to improve the performance of communication libraries is an active area of research. Hoefler et al. proposed a distributed graph interface to detect the topology of large scale InfiniBand networks and also studied the impact of network contention in large scale supercomputing systems [2]. Rashti et al. also proposed mechanisms to detect the InfiniBand network topology [3]. Previously, we proposed a network topology discovery service [4], which can be used by the MPI library to dynamically query the InfiniBand network's topology to form a topology model of the network and explored the benefits of designing topology-aware collective algorithms. However, some of the existing approaches have a time and space complexity of $\mathcal{O}(N_{\text{hosts}}^2)$ [3, 4] while others require administrative privileges to get the InfiniBand topology information which normal users of a supercomputing system will not have [2]. As supercomputing systems are aiming to scale to millions of processes, such techniques impose serious scalability and usability issues. These issues lead us to the following broad challenges:

1) Can we design a highly scalable network topology detection service for InfiniBand network?
2) How do we design the MPI communication library in a network-topology-aware manner to efficiently leverage the topology information generated by our service?
3) What are the potential benefits of using a network-

*This research is supported in part by U.S. Department of Energy grant #DE-FC02-06ER25755, National Science Foundation grants #CCF-0916302, #OCI-0926691, #OCI-0926574 and Teragrid #TG-CCR100034. This article has been authored by Lawrence Livermore National Security, LLC under Contract No. DE-AC52-07NA27344 with the U.S. Department of Energy. Accordingly, the United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this article or allow others to do so, for United States Government purposes. (LLNL-CONF-564812)

SC12, November 10-16, 2012, Salt Lake City, Utah, USA
978-1-4673-0806-9/12/$31.00

topology-aware MPI library on the performance of parallel scientific applications?

In this paper, we propose novel, scalable designs to expose dynamic routing and topology information of InfiniBand networks efficiently via a user accessible client/server design and Neighbor-Joining techniques (NJ) [5]. We propose design changes to the MPI communication library that leverages this information to improve communication performance. Experimental results show that the new neighbor joining algorithm is able to significantly reduce the network topology discovery time. We also speed up the compute intensive phases of the algorithm using OpenMP constructs, achieving parallel efficiencies of about 85%. Micro-benchmark level evaluations show that the proposed network-topology-aware MPI library can improve the latency for all message sizes by up to 40%. With *MILC* [6], we see up to 6% improvement in total execution time on LLNL's Hyperion system at 1,024 cores and up to 15% improvement on TACC's Ranger system at 2,048 cores. On Hyperion, we were able to increase the percentage of communication which is intra-node from 29.04% to 43.52%. On Ranger, the percentage of communication occurring intra-node goes up from 43.51% to 57.95%. In both cases, we have also uniformly reduced the number of 7, 5, 3 and 1 hop exchanges. We also see up to a 15% improvement in the runtime of the *Hypre* [7] linear system solver at 1,024 processes on Hyperion.

II. Driving Example

We employ a simple case-study to clearly motivate the need for network-topology-aware process mapping to improve the communication overheads in a 3D stencil (nearest-neighbor) code. In our example, each process communicates with 6 nearest neighbors, across three dimensions. Figure 1 presents the network topology of a 4 node allocation on the Hyperion cluster. Figure 2(a) shows the default process-to-node mapping of a 32 process MPI job (2x4x4 grid). This figure also denotes the number of switch hops incurred between the communicating processes. For example, communicating peers that are within the same leaf switch will incur 1 hop, whereas those that are mapped across different switches may incur 3 hops. Since the 3 hop latencies are higher than the intra-switch operations, improving the process placement pattern to lower the 3-hop exchanges could potentially improve the communication overheads.

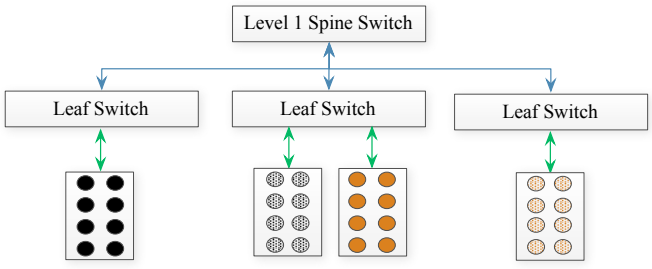


Fig. 1. Sample network hierarchy

In Figures 2(b) and 2(c), we show that it is possible to manually control the process mapping pattern to achieve this objective. In Figure 2(b), we show that we can increase the amount of intra-node communication by mapping 2x2x2 process sub-grids within a node. In Figure 2(c), we show that this can be further improved by leveraging network topology information. Table II shows the split up of the communication exchanges based on the number of network hops. In the default case, our example required as many as 48 3-hop exchanges. We could improve this to 32 3-hop transfers by considering the intra-node communication patterns. Further, we could lower this to 24 3-hop transfers, by mapping the processes in a network-topology-aware manner. In Figures 3(a), 3(b) and 3(c), we graphically compare the number of intra-node, 1-hop and 3-hop transfers across the three process placement patterns. We can observe that the number of 3-hop transfers are much lower in Figures 3(b) and 3(c), when compared to the default case. We also see that we have fewer 3-hop transfers and more 1-hop transfers in Figure 3(c), when compared to Figure 3(b). Although it is possible to manually map the processes in a network-topology-aware manner for such small cases, it becomes prohibitively complicated as we go to larger scale real-world application runs. However the benefits from such re-organization also increase as we scale to thousands of nodes. This strongly motivates the need for a scalable topology service that can support a network-topology-aware MPI library and can be used by applications in a portable fashion.

TABLE II
Split of messages based on number of hops

	Default Figure 2(a)	Optimized for intra-node Figure 2(b)	Optimized for intra-node and inter-node Figure 2(c)
Intra-node	80	96	96
1 Hop	0	0	8
3 Hops	48	32	24

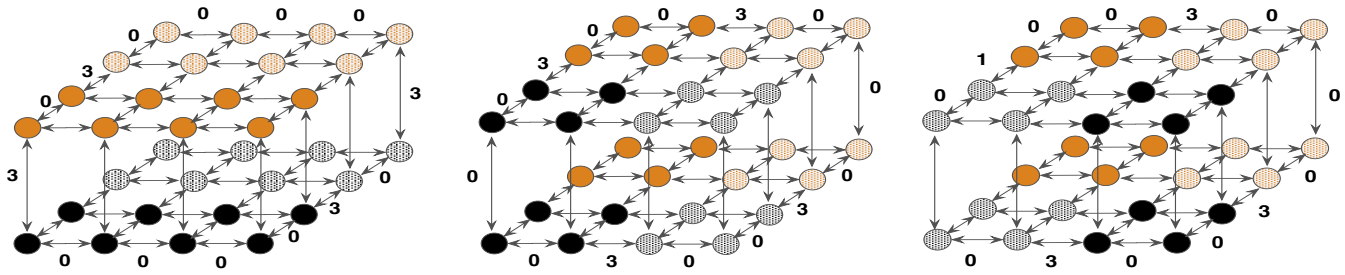
III. Background

A. Network Architectures

Large scale supercomputers such as Ranger at TACC and Hyperion at LLNL have thousands of compute nodes. These nodes are organized across different racks, with each rack consisting of many compute nodes. The network fabric may also be hierarchical with many levels of leaf and spine switches. While we focus on tree-based topologies in this paper, we believe these ideas are also directly applicable to other types of networks, such as the 3-D Torus [8] and 5-D torus found in IBM BlueGene architectures.

B. InfiniBand Network Topology Detection Service

InfiniBand is a very popular switched interconnect standard being used by 41.8% of the Top500 Supercomputing systems [9]. InfiniBand network topology data is not available in a mode easily used by other programs. The physical connections between entities in the fabric are discoverable with the `ibnetdiscover` utility from the OFED distribution [10]. The logical routing data is available by querying each switch in turn and dumping its Linear Forwarding Table (LFT) with the `ibroute` utility from OFED. We leverage the basic topology detection service proposed in [4], but have since extended

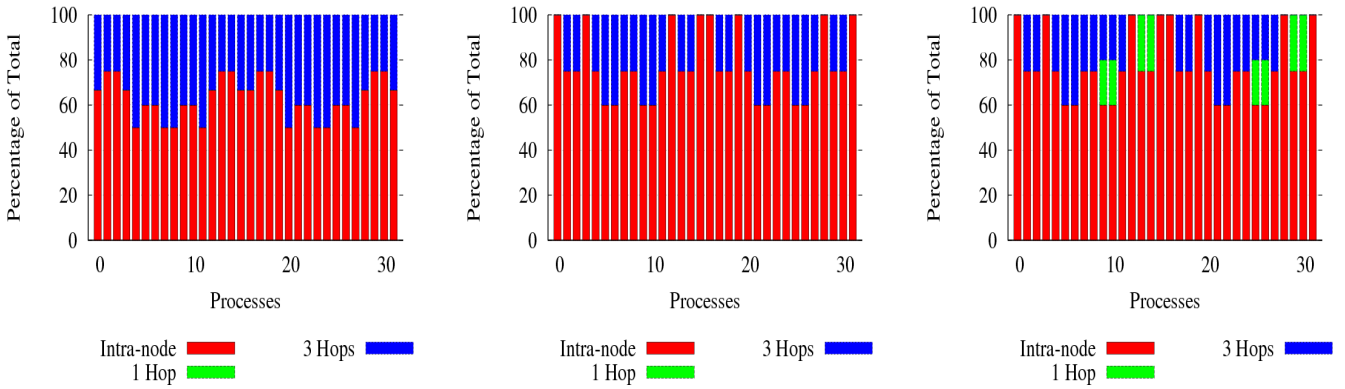


(a) Default process-to-node mapping used by supercomputers

(b) Node-aware manual mapping

(c) Node and network-aware manual mapping

Fig. 2. Communication pattern with various manually created process to host mappings



(a) Default process-to-node mapping used by supercomputers

(b) Node-aware manual mapping

(c) Node and network-aware manual mapping

Fig. 3. Split-up of physical communication in terms of number of hops with various manually created process to host mappings

the method to query routing information directly within the OpenSM subnet manager using a plugin interface.

C. Neighbor Joining

The neighbor-joining method (NJ) of Saitou and Nei [5] is a technique in computational biology for constructing phylogenetic trees which represent the evolutionary relationships among biological species based on similarities and differences in their characteristics. The authors' exposure to these techniques inspired the use of agglomerative minimum-distance complete-linkage hierarchical clustering in this work [11]. These algorithms generate binary trees from weighted complete graphs to identify closely-connected or closely-related regions. The algorithm starts with a weighted complete graph and an associated symmetric distance matrix giving the distance (the edge weights) between all pairs of nodes. At each step of the process, an auxiliary symmetric matrix is computed from the distance matrix, and then this matrix is searched for its smallest entry. The pair of nodes corresponding to this entry are "joined" by removing them from the graph and matrices and replacing them by a single new entry. New distances and auxiliary values are computed between the new node and the remaining nodes. The process repeats until only one node remains in the graph. The binary tree is then constructed by connecting all joined nodes to the new node they create.

D. Graph Partitioning Software

Jostle [12] and ParMETIS [13] are parallel multilevel graph-partitioning software packages which efficiently partition graphs containing millions of nodes. Jostle was developed by Dr. Walshaw et al. at the University of Greenwich and ParMETIS by Dr. Karypis, et. al of the University of Minnesota.

E. Applications

a) MILC: is a set of parallel numerical simulations developed by the MIMD Lattice Computation [6] collaboration for studying quantum chromodynamics (QCD). Codes from the *MILC* set are commonly used for benchmarking [14–16]. We use the ks_imp_dyn code from *MILC* which employs the conjugate gradient (CG) method on a 4D lattice. Communication is primarily nearest neighbor point-to-point and Allreduce as described in [17]. The influence of the layout on the communication performance is discussed in [18].

b) Hypre: is an open-source, high performance and scalable package of parallel linear solvers and preconditioners [7]. The solvers in Hypre are robust, numerically stable and scalable [19] and it may also be used as a framework for algorithm development.

c) AWP-ODC: is a community model used by researchers at the Southern California Earthquake Center (SCEC) for wave propagation simulations, dynamic fault rupture studies, physics-based seismic hazard analysis and improvement of the structural models [20]. It is a Fortran MPI code with a 3D nearest-neighbor communication pattern.

IV. Design

The overall design of our framework is presented in Figure 4. It consists of two major design components: (a) a topology query service and neighbor joining based representation

of IB networks and (b) a network-topology-aware MPI library. We will go into the details of each in the following sections.

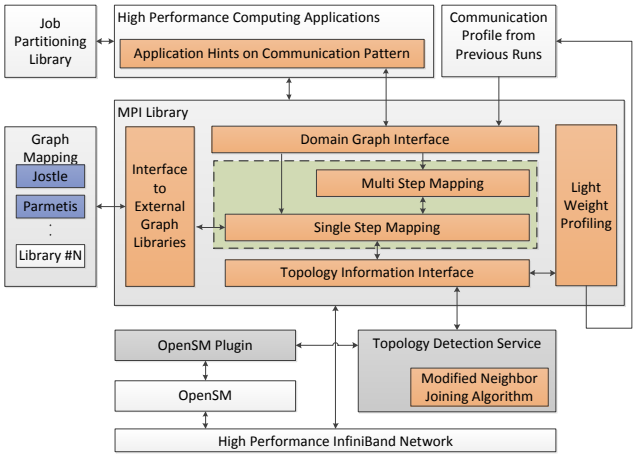


Fig. 4. Overall framework

A. Design of NJ based representation of IB networks

In [4], we proposed a network topology discovery service which can be used by the MPI library to dynamically query the InfiniBand network's topology and form a cost model of the network. In this paper, we enhance this basic infrastructure by extending the query service to obtain routing information more efficiently, via a direct plugin interface to the OpenSM subnet manager. We also extended the service to integrate a simplified form of the NJ algorithm into the topology detection service. Algorithm 1 shows the version of NJ algorithm we have used. The neighbor-joining algorithm implements agglomerative minimum-distance complete-linkage hierarchical clustering [11] to cluster the graph associated with the routes specified in the Linear Forwarding Tables (LFTs) for an IB fabric [10].

The multi-joining step is natural in this problem since we expect to have many hosts which are connected through a single switch ASIC in the leaves of the physical network. IB fabrics often have an identifiable leaf switch connecting 12 or 18 HCAs into the fabric. Our clustering algorithm will detect these closely-connected hosts because each host connected physically to this ASIC will have a logical route through this ASIC only. As a result, all hosts physically connected to this switch will have unit distances to each other and will be subsequently joined together by the modified NJ algorithm. Figures 5(a) and 5(b) depict the physical network topology and the output (neighbor joined tree) of the NJ algorithm for the Ranger supercomputer at TACC. Figures 6(a) and 6(b) show the same results for the Gordon supercomputer at SDSC.

The NJ algorithm is implemented as part of an integrated IB fabric query service developed to support the discovery of routing and switching information about IB [4]. The routing query server is a multi-threaded network program designed to handle large numbers of simultaneous connections on large machines with many jobs simultaneously requesting data. The NJ algorithm requires $\mathcal{O}(N^2_{\text{hosts}})$ routing queries to populate the initial distance matrix. Our approach only executes this algorithm once at startup or whenever changes are detected in the fabric connectivity. Note that we detect routing changes automatically via heavy sweep trigger events with the OpenSM plugin and automatically update the query service with new data after SM remaps have completed. Other approaches, which do not utilize such a centralized service, will have to incur this cost every time an application or MPI library wants to perform network-topology-aware activities.

Input : Weighted Network Topology Graph
Output: NJ tree

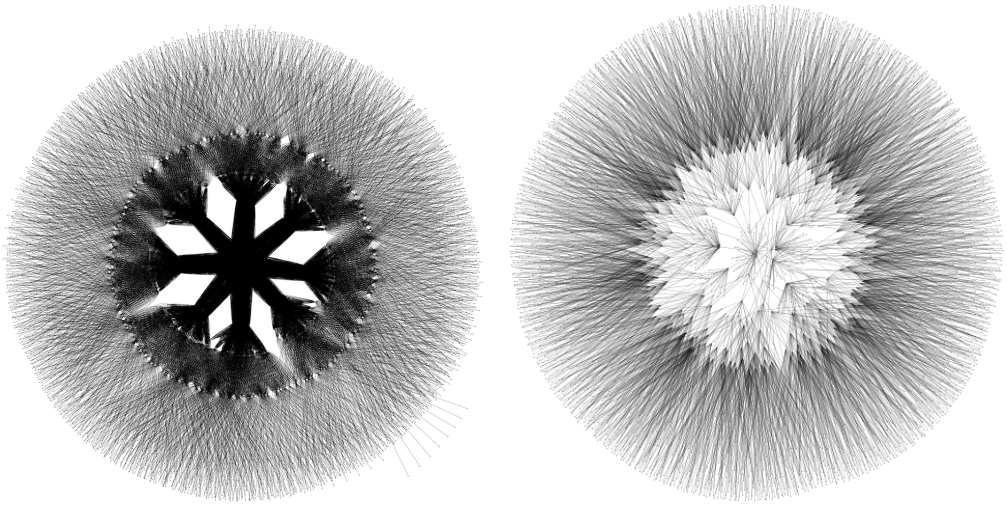
`/* Initialize */`
$d(i,j)$ mapping node pairs to distances;
`/* This is a reverse map of d */`
$rd(m)$ mapping distance m to a list of node id pairs;
$a(i)$ active node ids;
`/* A tree with one root node and active nodes as children */`
t;

repeat
  for $l = 1, 2, \ldots$ **do**
    foreach *pair* $(i,j) \in rd(l)$ **do**
      if $a(i)$ && $a(j)$ **then**
        break;
      else
        remove inactive pairs from rd;
      end
    end
  end
  Add (i,j) to empty list L;
  foreach *row* i **do**
    Find, other $d(i,k)$==$d(i,j)$ and add k to L;
  end
  foreach $i \in L$ **do**
    foreach $j \in L$ **do**
      if $d(i,j) != \text{minimum}$ **then**
        Remove j from L;
      end
    end
  end
  foreach *i in L* **do**
    $a(i)$=false;
  end
  Create new tree node n and insert into t;
  Connect n to nodes in L as parent and root as child;
  set m=max((x,y) in LxL: d(x,y));
  foreach *a(i)==true* **do**
    $d(i,n)$=m;
    add (i,n) to $rd(m)$;
  end
until *no new entries added to t*;

Algorithm 1: Neighbor Joining Algorithm

B. Design of Network-Topology-Aware MPI Library

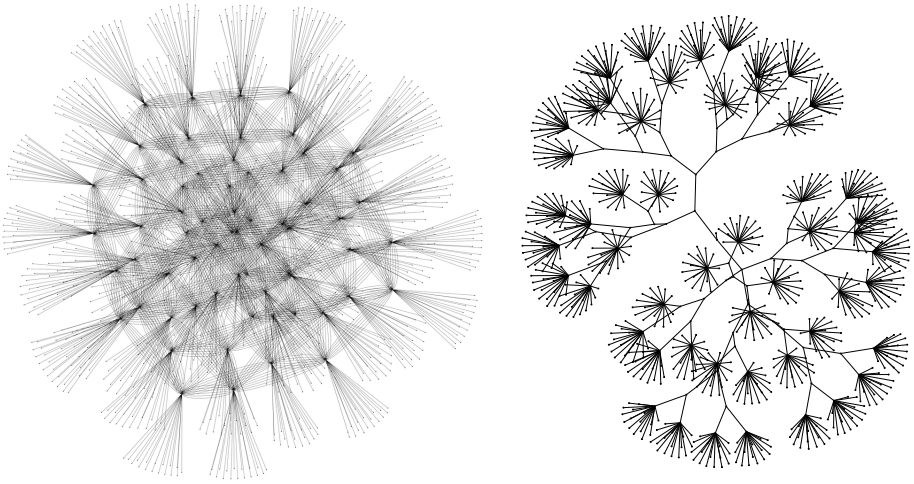
We next describe the various design changes made to the MVAPICH2 [21] MPI library in order to support network-



(a) Physical network topology graph - 4,037 end points (input)

(b) Neighbor joined tree (output)

Fig. 5. Graphical overview of NJ algorithm created within the topology query service on Ranger



(a) Physical network topology graph - 1,086 end points (input)

(b) Neighbor joined tree (output)

Fig. 6. Graphical overview of NJ algorithm created within the topology query service on Gordon

topology-aware mapping of processes. The design has the following major components: (a) Topology Information Interface and (b) Network-topology-aware mapping schemes.

We create multiple communicators to aid us in mapping the communication topology to the network topology. We describe these communicators in Section IV-B2 and their use in Section IV-B4. We design an abstraction layer called the Domain Graph Interface (DGI) to isolate the upper level users from the internal implementation details. In a similar fashion, we use a predefined set of interface routines to access external graph partitioners such as ParMETIS and Jostle. This allows us to easily switch between alternate graph partitioners without making major changes to our network-topology-aware process placement techniques. We also implement a light weight profiling module in the MPI library to analyze the communication pattern of applications from the point of view of number of hops traveled in the network. We show several results from this module in Section V.

1) Design of Topology Information Interface: The Topology Information Interface (TII) is designed to abstract the details of the underlying topology from the MPI library - be it inter-node network topology or intra-node processor topology. All of the APIs exposed by the interface depend on the network topology detection service described in Section III-B. A brief description of the more important APIs is given below:

- `tii_get_network_depth`: Retrieves the depth of NJ tree described in Section IV-A
- `tii_num_switch_clusters_at_depth`: Retrieves the number of unique switch clusters the job as a whole is connected to at a given level of the NJ tree
- `tii_connected_switch_cluster_at_depth`: Retrieves the unique ID of the switch cluster a particular rank is connected to at a given depth of the NJ tree

Currently, the TII does not take the intra-node processor topology into account and returns equal communication costs

for all processors within a compute node. However, we are working on introducing this support into the TII by leveraging features provided by external libraries such as the Portable Hardware Locality (hwloc) [22].

2) Communicator Design to Support Graph Mapping: As shown in Figure 7, we create multiple sub-communicators to reflect the network topology of the system. We build upon the idea proposed in our earlier work [1]. Network topology information is obtained using the APIs provided by the TII described in Section IV-B1. To prevent overwhelming the topology discovery service, MPI rank 0 performs the required queries for the entire job and broadcasts it to rest of the processes. Note that in the MVAPICH2 MPI stack, the process with rank 0 will also be a node-level leader. Rank 0 will query and obtain the depth of the NJ tree, the number of switch clusters at a particular level of the tree and the unique ID of the switch clusters at various levels of the NJ tree each node is connected to. All of these parameters are identified by the network topology detection service using the NJ algorithm described in IV-A. If there are multiple switch clusters at a level, we partition the communicator to create separate communicators for switch clusters as depicted in Figure 7. We use the unique ID of the switch cluster as the `color` to split the communicator.

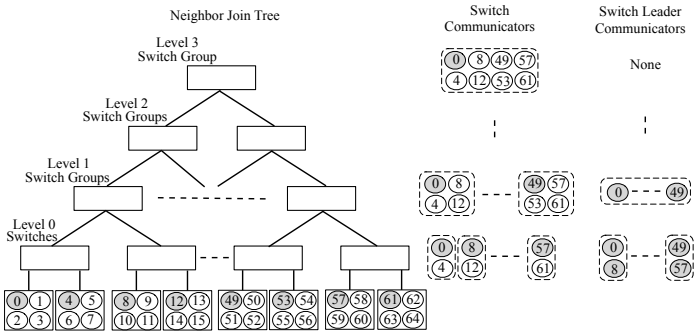


Fig. 7. Communicator design to support network-topology-aware process placement

3) Single Step Mapping: The single step mapping module shown in Figure 4 forms the base for all of our designs. Other mapping approaches such as the multi step mapping described in Section IV-B4 rely on the single step mapping module to solve the graph partitioning problem. This module accepts a communicator and topology information as inputs and returns the network optimized problem domains embedded in the communicator object as output.

Once the library has obtained the necessary information regarding the communication topology of the application through the DGI, it attempts to minimize the amount of network communication between various processes by considering it as a k-way graph partitioning problem [12]. The MPI library relies on external graph partitioners to map the communication pattern of the application to the topology of the underlying network. We use both Jostle and ParMETIS for this. Most graph partitioners require the user to pass the relative weights of the various edges in the communication graph as input. We explored various metrics like network hops, frequency of communication, volume of communication and frequency / volume of communication weighted using network hops. In Section V, we observe that the volume of communication is the most suitable metric.

The relative communication volume between peers can either be passed onto the MPI library through the DGI, or it can be obtained by executing the job with some special profiling flags that instruct the MPI library to record the communication frequency between various processes in a file. During subsequent executions, the MPI library automatically uses this file to identify the frequency of communication between peers. If this file is absent or inconsistent with the current job size, and if the application did not provide it through the DGI, the library assumes that all processes communicate equally with their peers. The library then transforms the user input along with the network topology information, gathered through the TII, into a form that is acceptable for the interface to external graph partitioners.

As a single execution of the graph mapping library may not yield a balanced mapping of high quality, the single step mapping module calls the external graph mapping library multiple times until it yields a domain of sufficient quality for the application. The level of quality required can be defined by the user through an environment variable.

4) Multi Step Mapping: The multi-step mapping module shown in Figure 4 uses the single step mapping module described in Section IV-B3 as well as the various communicators described in Section IV-B2 to effectively map the various processes that are part of the MPI job onto the supercomputing system. We designed two different network-topology-aware process mapping schemes. For reference, Figure 7 identifies the location of each rank that is mentioned in the following discussion. Note that we assume that level 1 is the top level for purposes of describing the two schemes. In both schemes, the node level leaders (0, 4, 8, 12, 49, 53, 57 and 61) gather data from the other intra-node processes (0 will gather from 1, 2 and 3).

Scheme 1: In this scheme, we follow a top-down approach to the graph partitioning problem, recursively partitioning at each switch level. The goal is to reduce the number of long distance communications over the network and to concentrate communication within individual switch clusters as much as possible. All data pertaining to the application communication pattern like vertices, edges, edge weights, etc. are gathered by the members of the top level (say level 'n') switch leader communicator (0 and 49). The leaders then call the single step mapping module with this aggregated information in order to reduce the number of inter-domain communication. Inter-domain communication in the context of the top level switch leaders correspond to the communication that spans the longest distance over the network for the job in question (5 hops in this example). Once the partitioning is done, the top level leaders scatter the data to the members of switch communicators at that level (0 scatters to 4, 8 and 12; 49 scatters to 53, 57 and 61). The members of switch-leader communicators at level 'n-

1' (0 and 8; 49 and 57) aggregate this information through the switch communicator at that level (0 gathers from 4; 8 gathers from 12; 49 gathers from 53 and 57 gathers from 61). These leader processes then call the single step mapping module again to reduce the inter-domain communication at that level (3 hops in this example). The process is repeated until we reach the set of communicators representing the switches at the lowest level.

Scheme 2: The second scheme is built atop scheme 1, but contains a critical difference in that we partition the data between all node level leaders (0, 4, 8, 12, 49, 53, 57 and 61) first. Once this step is done, we aggregate all the information contained in the vertices inside one node to create a new `super-vertex`. The idea is to not allow the individual vertices to be moved from one node to another independently. We then perform the same steps outlined in scheme 1 with this new `super-vertex`. The goal in this approach is to have as much communication as possible within one node and prevent it from being disturbed by further rounds of partitioning.

V. Experimental Results

In this section, we describe the experimental setup, provide the results of our experiments, and give an in-depth analysis of these results. All numbers reported here are averages of multiple runs conducted over the course of several days. For the rest of this section *default* scheme refers to default process placement scheme in place on modern supercomputing systems and *topo-aware* scheme refers to our proposed network-topology-aware process placement.

A. Experimental Setup

We used multiple high performance computing systems to obtain the results for this paper:

Ranger: Ranger is comprised of 3,936 16-way SMP compute nodes providing a total of 62,976 compute cores. Each core operates at 2.3 GHz and has 32 GB of memory with an independent memory controller per socket. Ranger has two 3,456 port SDR Sun InfiniBand Datacenter switches at its core. The interconnect topology is a 7-stage, full-CLOS fat tree.

Hyperion: This is a 1,400-core testbed where each node has eight Intel Xeon (E5640) cores running at 2.53 Ghz with 12 MB L3 cache. Each node has 12 GB of memory and an MT26428 QDR ConnectX-2 HCA. It has a 171-port Mellanox QDR switch, with 11 leafs, each having 16 ports. Each node is connected to the switch using one QDR link. Although Hyperion does have additional compute cores, they are not homogeneous in nature. Hence we restrict ourselves to the nodes described above.

B. Performance Results for Neighbor Joining

In this section, we provide various results to analyze performance of the neighbor joining algorithm.

1) Improving Performance of Neighbor Joining Algorithm with OpenMP Constructs: As we saw in Section IV-A, the NJ algorithm requires $\mathcal{O}(N_{\text{hosts}}^2)$ routing queries at startup to populate the initial distance matrix. For large fabrics, this step originally took several minutes on a single core and comprised over 80% of the runtime required to initialize the service. To alleviate this bottleneck, we optimized and threaded this portion of the algorithm using OpenMP constructs. Figure 8 shows the results of OpenMP parallelization from this effort for the Lonestar and Ranger fabrics using 3,942 and 1,901 end-point hosts for the NJ algorithm, respectively. In both cases, parallel scaling efficiencies of 85% or higher were obtained when running on a 6-core Westmere processor running at 2.67 GHz. The total time to initialize the service for the large Ranger fabric is approximately 12 seconds using 6 threads.

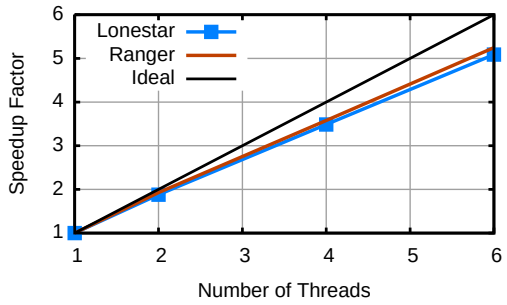


Fig. 8. OpenMP performance of host-pair queries showing parallel efficiencies of 85% on a single six-core Intel Westmere processor.

2) Overhead of Network Topology Detection: We compare the old scheme which required the MPI library to make $\mathcal{O}(N_{\text{hosts}}^2)$ routing queries at startup to populate the initial distance matrix, with the new Neighbor Joining scheme which only requires the MPI library to make $\mathcal{O}(N_{\text{hosts}})$ queries. Figure 9 shows the time taken in milliseconds for the older scheme and the newer NJ based scheme. As we can see, the newer scheme is far more scalable as the system size increases. As we go forward to exascale systems, such scaling performance will be extremely critical.

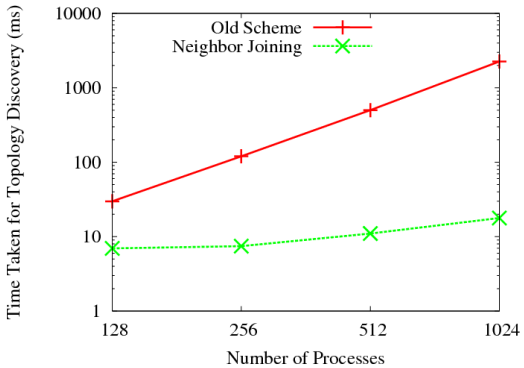


Fig. 9. Comparison of query times incurred in old scheme and NJ

C. Application Level Performance Results Showing Benefits of Network-Topology-Aware Process Placement

In the following sections we show the benefits obtained at the application level by using a network-topology-aware MPI library.

1) Overhead of Topology Discovery and Graph Mapping: Table III shows the time spent by the MPI library in various phases of topology discovery and graph mapping for different number of compute nodes for the *MILC* code. As we can see, the overhead caused due to the network-topology-aware

TABLE III
OVERHEAD OF TOPOLOGY DISCOVERY AND GRAPH MAPPING (TIME IN SECONDS)

Number of Nodes (Processes)		16 (128)	32 (256)	64 (512)	128 (1,024)
Distribution of Time for Topology Mapping	Discover Network Topology	0.006	0.007	0.011	0.018
	Communicator Creation	0.006	0.008	0.578	0.773
	Create Topology Aware Mapping	0.180	0.228	0.635	1.474
	Total Time	0.192	0.243	1.224	2.265
Total Execution Time of *MILC*		460.66	462.69	590.66	689.36
Time for Topology Mapping as a Percentage of *MILC* Execution Time (%)		0.04%	0.05%	0.21%	0.33%

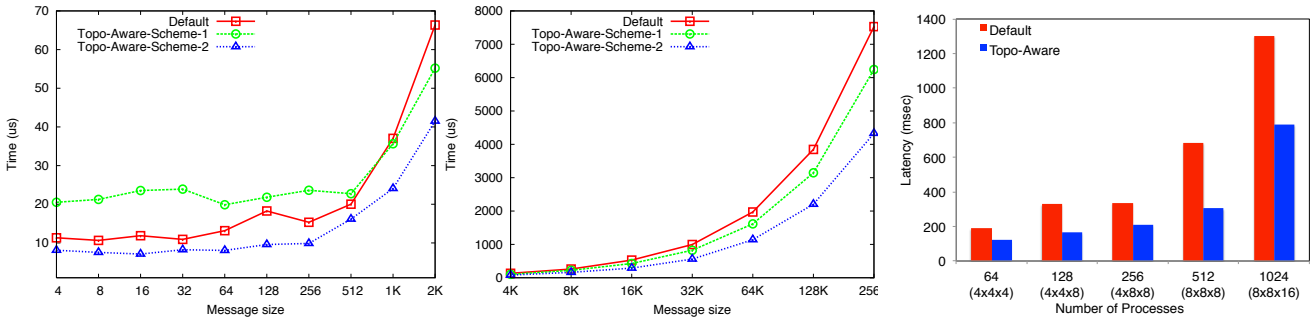
graph mapping scheme only forms a very small percentage of the total time taken by the application for varying job sizes. Another point to note is the amount of time taken for the communicator creation phase. This is orthogonal to our work. Several researchers have already proposed solutions [23, 24] for this which can be easily integrated into our design. We are currently working on enhancing the design of the topology mapping module to make it more robust and scalable.

2) Performance with 3D stencil benchmark: The processes in the benchmark are mapped onto a 3D grid and each process talks to its neighbors in each dimension (6 neighbors). In every step, each process posts MPI_Irecv operations for all of the messages it expects and then posts all of the MPI_Isend calls. It waits for all of the transfers to complete with a single MPI_Waitall call. We compare both schemes we have outlined in Section IV-B4 against the case that does not have any network-topology-aware optimizations. Figures 10(a) and 10(b) compare the performance of the 3D stencil benchmark for small and large messages respectively at 512 processes. We see that for large messages both of the schemes perform better than the default scheme, with scheme-2 giving up to 40% improvements. But for smaller messages we see that scheme-1 performs worse when compared with the default case. Analysis of the performance from a network perspective showed that although we were decreasing the number of long distance network communications with scheme-1, the percentage of intra-node messages were also decreasing leading to poorer overall performance. We do not show the graphs showing the split up of physical communication due to lack of space. Scheme-2 on the other hand was able to increase the percentage of intra-node communication as well as decrease the total number of long distance network communication leading to the best benefits. Hence we use scheme-2 for all of our experiments at the applications level. To verify the scalability of our design, we compare scheme-2 with the default scheme at various system sizes for a fixed message size of 64 KB. As we can see from Figure 10(c), the topo-aware scheme consistently performs better at all system sizes. We do not show the trends on Ranger as they are similar.

3) Impact of Network Topology Aware Task Mapping on the Communication Pattern of AWP-ODC: To examine the effectiveness of our topo-aware process placement scheme further, we use AWP-ODC. We analyze the physical communication pattern in terms of number of hops for the default and the automatic topo-aware process placement scheme. Figures 11(a) and 11(b) depict the communication pattern in terms of number of hops for the default and topo-aware scheme, respectively. As we can clearly see, the topo-aware process placement scheme is able to improve the intra-node communication percentage from 33% to 66%. It is also able to reduce the number of 7, 5, 3 and 1 hop inter-node exchanges. Figure 11(c) summarizes the gains obtained in reducing the number of long distance communication. Although the topo-aware scheme does lead to modest gains in the performance of the application, further gains are limited by an inherent imbalance in the application due to boundary condition updates. Nevertheless, this shows the potential benefit that a topo-aware process placement can have.

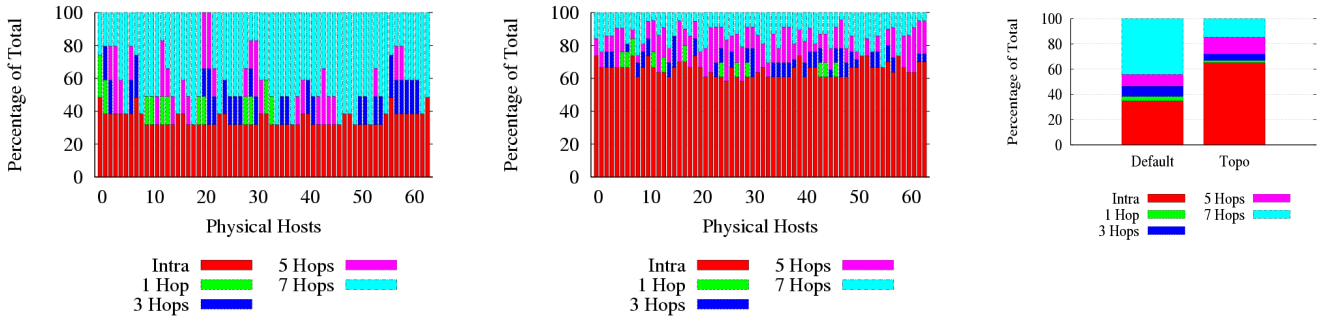
4) Impact of network topology on performance of Hypre: *Hypre* is an open-source, high performance and scalable package of parallel linear solvers and preconditioners. It mostly uses small to medium sized messages for communication. We evaluate the AMG pre-conditioner inside *Hypre* using the *new_ij* benchmark that is available with the *Hypre suite* for various input matrix sizes at a system size of 1,024 processes on Hyperion. We used the following parameters as input to run the benchmark "`-27pt -pmis -interptype 6 -Pmx 4 -amg_max_its 100`". Figure 12(a) compares the normalized performance of the topo-aware scheme with the default scheme for different matrix sizes. As we can see, the topo-aware scheme gives between 10% to 15% improvement over the default scheme that is unaware of network topology. We further analyze the physical communication characteristics of the application to gain insights into the performance benefits. Figures 12(b) and 12(c) depict the split up of physical communication in terms of number of hops for the default and topo-aware cases, respectively. As we can see, we are able to increase the percentage of intra-node communication (from 45.00% to 53.28%) uniformly across all of the physical hosts involved in the job. We can also see, in the inter-node communication, we are able to reduce the number of long distance 5 and 3 hop transfers in exchange for an increase in the short distance 1 hop transfer. Due to scheduling conflicts, we were unable to run *Hypre* at large scales on Ranger.

5) Impact of network topology on performance of MILC: We run the *ks_imp_dyn* code which simulates dynamical Kogut-Susskind fermions with the input that is available for it from the NERSC website [25]. For the cases we ran, *MILC* mostly used medium messages in the range for 4 KB - 64 KB. Figure 13(a) compares the normalized performance of the topo-aware and default schemes for various system sizes on Ranger. As we can see, the performance improvements



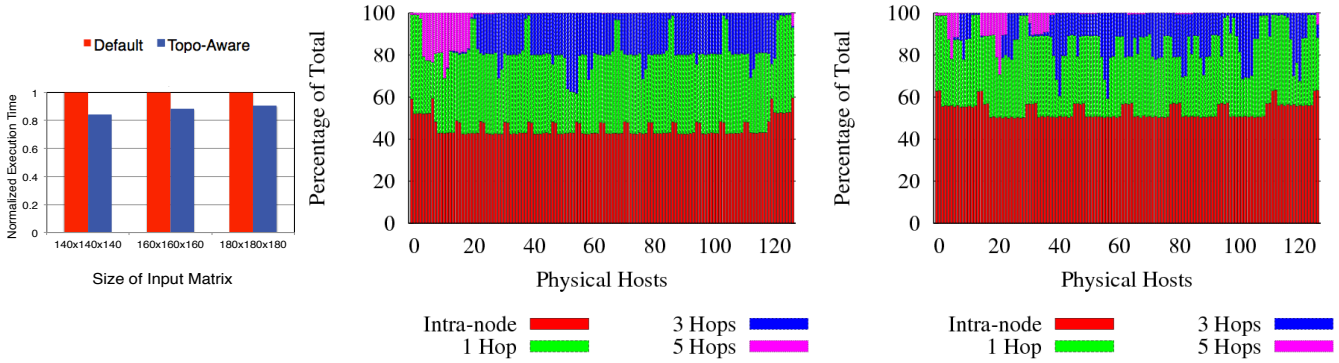
(a) Small Messages on Ranger (b) Large Messages on Ranger (c) Performance at scale on Hyperion

Fig. 10. Latency Performance of 3D Stencil Communication Benchmark



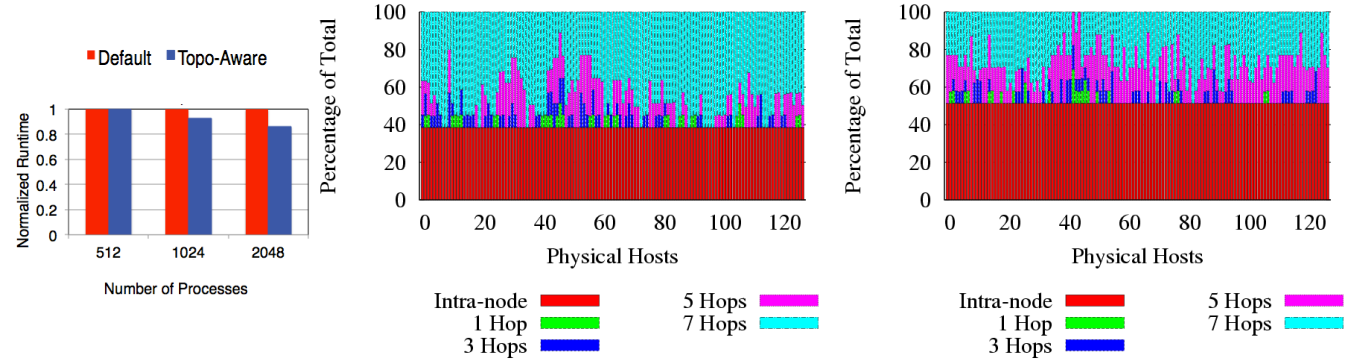
(a) Default (b) Topo-Aware (c) Summary

Fig. 11. Overall split up of physical communication for AWP-ODC based on number of hops for a 1,024 core run on Ranger



(a) Performance with varying matrix sizes (b) Default for 180x180x180 matrix (c) Topo-Aware for 180x180x180 matrix

Fig. 12. Overall performance and Split up of physical communication for *Hypre* based on number of hops for a 1,024 core run on Hyperion



(a) Performance for varying system sizes (b) Default for 2048 core run (c) Topo-Aware for 2048 core run

Fig. 13. Overall performance and Split up of physical communication for *MILC* on Ranger

seen increases as the size of the system increases, which is a very encouraging result. Overall we see that the topo-aware versions give up to a 10% improvement in performance when compared to the default scheme at at 1,024 cores and up to a 15% improvement at 2,048 cores. We analyze the communication pattern in terms of number of hops for the topo-aware and default versions. Figures 13(b) and 13(c) depict the communication split up in terms of number of hops for the default and topo-aware versions, respectively, at 2,048 cores on Ranger. As we can see, the topo-aware version is able to increase the percentage of intra-node communication (from 44.12% to 58.77%). We are also able to significantly reduce the number of long distance (7-hop) communication from 73.10% of inter-node exchanges to 48.67%. The topo-aware scheme brings in most of this communication to inside a node and the remaining to lower hop location by intelligent placement of MPI ranks. Due to sparse nature of the allocation and the communication pattern of *MILC* (128 nodes were allocated on up to 109 different leaf switches, these switches were connected up to 66 different switch clusters at level 1 of the NJ tree, leaving only 2 or 3 nodes per level 2 switch cluster), we were not able uniformly reduce the number of 5 hop and 3 hop exchanges as we did on Hyperion for *Hypre*.

VI. Related Work

Extensive research has been conducted around the mapping problems in parallel and distributed computing. The general idea is to map a task graph to a network graph while minimizing the overhead of communication and balancing computational load. This problem can be reduced to a graph embedding problem which has been proved to be $\mathcal{NP}-Complete$ [26, 27]. Researchers have proposed different solutions to solve the mapping problem based on heuristic algorithms [28–30] and physical optimization algorithms [31–34]. But most of these works targeted interconnect topologies such as hypercubes, array processors or shuffle-exchange networks, while modern system topologies are mainly meshes, tori and fat-trees. With petascale clusters, it is critical to improve the mapping of communication graphs to improve communication overheads of large scale parallel applications.

Bhatele et al. [35, 36] presented a framework for automatic mapping of parallel applications using a suite of heuristics. Their framework was based on obtaining the communication graph using profiling libraries and applying heuristics to obtain a mapping solution. The topology information was obtained through system calls. These works demonstrated the importance of topology-aware mapping for communication bound applications on tori networks and proprietary systems like BlueGene.

Mercier and Jeannot proposed the *TreeMatch* algorithm [37] which provides a near-optimal placement of MPI processes on NUMA architectures. They evaluated this algorithm using two different metrics [38] to assign weights to the edges: the communication volume and frequency. Nevertheless, this work does not take into account the physical topology of the network.

Rashti et al. [3] proposed a network discovery module based on *ibtracert* and used *Scotch* [39] to match the communication graph of the application to the physical topology. Hoefler et al. [2] proposed different topology mapping algorithms available for different network topologies. However, these approaches either relied on *ibtracert* and *ibnetdiscover*, which are privileged operations or, as discussed in Section I, were not scalable.

In this paper, we propose a scalable network discovery service for InfiniBand available at the user level. With this service, we design a network-topology-aware MPI library to provide a topology aware mapping of the MPI processes.

VII. Conclusions and Future Work

We have presented a novel, scalable InfiniBand network topology service based on the neighbor joining algorithm. To the best of our knowledge this is the first instance where the neighbor joining algorithm has been applied to solve the problem of InfiniBand network detection. We have demonstrated the applicability and scalability of our service by using it to design a network-topology-aware MPI library, and we have shown that its use can improve performance of real-world applications and libraries at scale.

Experimental results demonstrate that the new service reduces the application cost of network topology discovery time from $O(N_{\text{hosts}}^2)$ to $O(N_{\text{hosts}})$. We accelerated the initial distance matrix construction using OpenMP constructs, achieving 87% parallel efficiency on 6 cores on a Westmere node, further reducing discovery cost. Micro-benchmark level evaluations showed that the proposed topo-aware MPI can enhance the performance of all message sizes by up to 40%. With *MILC*, we see up to 6% and 15% improvement in total execution time on 1,024 cores of Hyperion and 2,048 cores of Ranger, respectively. On Hyperion we were able to increase the percentage of intra-node communication from 29.04% to 43.52%. On Ranger the percentage of intra-node communication went up from 43.51% to 57.95%. In both cases, we reduced the number of 7, 5, 3 and 1 hop exchanges. We also got up to a 15% improvement in the runtime of the *Hypre* linear system solver using 1,024 processes for varying sizes of the input matrix on Hyperion.

The continuing goal of this modified NJ approach is to aid batch scheduling decisions by identifying closely-connected subgraphs of free resources where the maximum number of switch ASICs traversed by network traffic within the subgraph is minimized. This should lead to performance improvements for codes whose performance is latency sensitive. We have modified benchmarks like IRS [40] and end applications such as ALE3D [41] to directly use the DGI rather than rely on the profiling information obtained from the MPI library. As part of future work, we plan to run these modified benchmarks and applications at scale on various supercomputing systems. We also plan to evaluate the impact our research can have on performance of end applications for other networks topologies such as 3D/5D tori.

REFERENCES

[1] K. Kandalla and H. Subramoni and D.K. Panda, "Designing Topology-Aware Collective Communication Algorithms for Large Scale InfiniBand Clusters : Case Studies wih Scatter and Gather," in *IPDPS*, 2010.

[2] T. Hoefler and M. Snir, "Generic Topology Mapping Strategies for Large-scale Parallel Architectures," in *Proceedings of the 2011 ACM International Conference on Supercomputing (ICS'11)*. ACM, Jun. 2011, pp. 75–85.

[3] M. J. Rashti, J. Green, P. Balaji, A. Afsahi, and W. Gropp, "Multi-core and Network Aware MPI Topology Functions," in *Proceedings of the 18th European MPI Users' Group conference on Recent advances in the message passing interface*, ser. EuroMPI'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 50–60.

[4] H. Subramoni, K. 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 *CLUSTER*, 2011.

[5] N. Saitou and M. Nei, "The Neighbor-Joining Method: A New Method for Reconstructing Phylogentic Trees," *Mol. Biol. Evol*, vol. 4, pp. 406–425, 1987.

[6] The MIMD Lattice Computation (MILC) Collaboration, http://physics.indiana.edu/~sg/milc.html.

[7] R. D. Falgout and U. M. Yang, "Hypre: A Library of High Performance Preconditioners," in *Proceedings of the International Conference on Computational Science-Part III*, ser. ICCS '02. London, UK, UK: Springer-Verlag, 2002, pp. 632–641.

[8] D. Chen, N. A. Eisley, P. Heidelberger, R. M. Senger, Y. Sugawara, S. Kumar, V. Salapura, D. L. Satterfield, B. Steinmacher-Burow, and J. J. Parker, "The IBM Blue Gene/Q Interconnection Network and Message Unit," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 26:1–26:10. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063419

[9] Top500, "Top500 Supercomputing systems," Jun 2011, http://www.top500.org/.

[10] "Open Fabrics Enterprise Distribution," http://www.openfabrics.org/.

[11] S. C. Johnson, "Hierarchical Clustering Schemes," *Psychometrika*, vol. 32, no. 3, pp. 241–254, September 1967.

[12] C. Walshaw and M. Cross, "JOSTLE: Parallel Multilevel Graph-Partitioning Software – An Overview," in *Mesh Partitioning Techniques and Domain Decomposition Techniques*, F. Magoules, Ed. Civil-Comp Ltd., 2007.

[13] K. Schloegel, G. Karypis, and V. Kumar, "Parallel Static and Dynamic Multi-Constraint Graph Partitioning," *Concurrency and Computation: Practice and Experience*, pp. 219–240, 2002.

[14] C. D. Spradling, "SPEC CPU2006 Benchmark Tools," *SIGARCH Comput. Archit. News*, vol. 35, no. 1, pp. 130–134, Mar. 2007.

[15] Müller, Matthias S. and van Waveren, Matthijs and Lieberman, Ron and Whitney, Brian and Saito, Hideki and Kumaran, Kalyan and Baron, John and Brantley, William C. and Parrott, Chris and Elken, Tom and Feng, Huiyu and Ponder, Carl, "SPEC MPI2007–An Application Benchmark Suite for Parallel Systems using MPI," *Concurr. Comput. : Pract. Exper.*, vol. 22, no. 2, pp. 191–205, Feb. 2010.

[16] The NERSC SDSA Benchmark Codes, http://www1.nersc.gov/projects/SDSA/software/.

[17] W. P. Nicholas J. Wright and A. Snavely, "Characterizing Parallel Scaling of Scientific Applications using IPM," in *10th LCI Conference*, Mar. 2009.

[18] He, Jun and Kowalkowski, Jim and Paterno, Marc and Holmgren, Don and Simone, James and Sun, Xian-He, "Layout-Aware Scientific Computing: A Case Study using MILC," in *Proceedings of the Second Workshop on Latest AdScalable Algorithms for Large-Scale Systems*, ser. ScalA '11. ACM, 2011, pp. 21–24.

[19] A.H. Baker, R.D. Falgout, T.V. Kolev and U.M. Yang, "Scaling hypre's Multigrid Solvers to 100,000 Cores," in *High Performance Scientific Computing: Algorithms and Applications - A Tribute to Prof. Ahmed Sameh, M. Berry et al., eds., Springer, LLNL-JRNL-479591*, 2012.

[20] Y. Cui, R. Moore, K. Olsen, A. Chourasia, P. Maechling, B. Minster, S. Day, Y. Hu, J. Zhu, A. Majumdar, and T. Jordan, "Toward Petascale Earthquake Simulations," in *Acta Geotechnica (in press), Springer*, 2008.

[21] MVAPICH2, http://mvapich.cse.ohio-state.edu/.

[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 *PDP2010*, 2010.

[23] P. Sack and W. Gropp, "A Scalable MPI_Comm_split Algorithm for Exascale Computing," in *Recent Advances in the Message Passing Interface*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2010.

[24] J. Dinan, S. Krishnamoorthy, P. Balaji, J. R. Hammond, M. Krishnan, V. Tipparaju, and A. Vishnu, "Noncollective Communicator Creation in MPI," in *EuroMPI*, 2011.

[25] The NERSC-6 Benchmarks, http://www.nersc.gov/research-and-development/benchmarking-and-workload-characterization/nersc-6-benchmarks/.

[26] S. H. Bokhari, "On the Mapping Problem," *IEEE Transactions on Computers*, vol. 30, pp. 207–214, 1981.

[27] F. Erçal, J. Ramanujam, and P. Sadayappan, "Task Allocation onto a Hypercube by Recursive Mincut Bipartitioning," *J. Parallel Distrib. Comput.*, vol. 10, no. 1, pp. 35–44, 1990.

[28] B. W. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *Bell System Technical Journal*, vol. 49, no. 2, pp. 291–308, 1970.

[29] S.-Y. Lee and J. K. Aggarwal, "A Mapping Strategy for Parallel Processing," *IEEE Trans. Comput.*, vol. 36, no. 4,

pp. 433–442, Apr. 1987.

[30] S. Radhakrishnan, R. Brunner, and L. V. Kalé, "Branch and Bound Based Load Balancing for Parallel Applications," in *Proceedings of the Third International Symposium on Computing in Object-Oriented Parallel Environments*, ser. ISCOPE '99. London, UK: Springer-Verlag, 1999, pp. 194–199.

[31] F. Berman and L. Snyder, "On Mapping Parallel Algorithms into Parallel Architectures," *Journal of Parallel and Distributed Computing*, vol. 4, pp. 439–458, 1987.

[32] S. W. Bollinger and S. F. Midkiff, "Heuristic Technique for Processor and Link Assignment in Multicomputers," *IEEE Trans. Comput.*, vol. 40, pp. 325–333, March 1991.

[33] N. Mansour and G. Fox, "Allocating Data to Multicomputer Modes by Physical Optimization Algorithms for Loosely Synchronous Computations," *Concurrency - Practice and Experience*, vol. 4, no. 7, pp. 557–574, 1992.

[34] T. Chockalingam and S. Arunkumar, "Genetic Algorithm Based Heuristics for the Mapping Problem," *Computers & Operations Research*, vol. 22, pp. 55–64, 1995.

[35] A. Bhatele, "Automating Topology Aware Mapping for Supercomputers," Ph.D. dissertation, Dept. of Computer Science, University of Illinois, August 2010.

[36] A. Bhatele, E. J. Bohm, and L. V. Kalé, "Optimizing Communication for Charm++ Applications by Reducing Network Contention," *Concurrency and Computation: Practice and Experience*, 2011.

[37] 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.

[38] G. Mercier and E. Jeannot, "Improving MPI Applications Performance on Multicore Clusters with Rank Reordering," in *Proceedings of the 18th European MPI Users' Group conference on Recent advances in the message passing interface*, ser. EuroMPI'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 39–49.

[39] F. Pellegrini and J. Roman, "Scotch: A Software Package for Static Mapping by Dual Recursive Bipartitioning of Process and Architecture Graphs," in *High-Performance Computing and Networking*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 1996.

[40] "Implicit Radiation Solver (IRS)," https://asc.llnl.gov/sequoia/benchmarks/#irs.

[41] "Arbitrary Lagrangian Eulerian in 3D (ALE3D)," https://wci.llnl.gov/codes/ale3d/.