netloc: Towards a Comprehensive View of the HPC System Topology

Brice Goglin, Joshua Hursey, Jeffrey M. Squyres

netloc: Towards a Comprehensive View of the HPC System Topology

Brice Goglin*, Joshua Hursey†, Jeffrey M. Squyres‡
* *Inria Bordeaux - Sud-ouest – 33405 Talence cedex – France*
† *University of Wisconsin-La Crosse – La Crosse, WI 54601 – USA*
‡ *Cisco Systems, Inc. – San Jose, CA 95134 – USA*
brice.goglin@inria.fr, jhursey@uwlax.edu, jsquyres@cisco.com

***Abstract*—The increasing complexity of High Performance Computing (HPC) server architectures and networks has made topology- and affinity-awareness a critical component of HPC application optimization. Although there is a portable mechanism for accessing the server-internal topology there is no such mechanism for accessing the network topology of modern HPC systems in an equally portable manner.**

The Network Locality (netloc) project provides mechanisms for portably discovering and abstractly representing the network topology of modern HPC systems. Additionally, netloc provides the ability to merge the network topology with the server-internal topologies resulting in a comprehensive map of the HPC system topology. Using a modular infrastructure, netloc provides support for a variety of network types and discovery techniques. By representing the network topology as a graph, netloc supports any network topology configuration. The netloc architecture hides the topology discovery mechanism from the application developer thus allowing them to focus on traversing and analyzing the resulting map of the HPC system topology.

I. Introduction

Developers tuning HPC applications are faced with many challenges due to the complexity and scale of modern HPC systems. Exploiting the full potential of these machines requires detailed knowledge of the application and the behavior of the entire system.

Topology awareness is critical to taking full advantage of a complex HPC system, especially with current Non-Uniform Memory Access (NUMA)-based commodity servers. The Hardware Locality project ("hwloc") [1] provides an abstract, uniform representation of a single server's topology. Applications can use this topology information to discover which hardware resources are available, select which resources to use, and then bind themselves to processors that are "NUMA close" to those resources. System libraries, such as Message Passing Interface (MPI), also use this information to optimize single-server communication during collective operations [2], [3].

This locality information is limited to single-server topologies, and neglects the off-server network topology information. Since network communication is generally much more expensive than single-server communication, an application may be able to exploit further performance optimizations if it is able to access, analyze, and react to the topology of the network in which it is running. Specifically, the application and system libraries can make more intelligent decisions about process placement in the HPC system, with whom they wish to communicate, and how often.

Ideally, applications could simply connect to a service that provides a unified topology map that combines the network and server topologies from the entire HPC system. Early studies have shown that if such unified topology information was available, applications and communication libraries would be able to see considerable performance gains (up to 30% in one case) [4], [5]. Recent research has shown the effectiveness of network topology information for tree-structured InfiniBand networks [6], [7], and congestion management may even be more important on multipath networks such as Ethernet. There is a need for a more generic, publicly-available tool that supports a wide variety of network types and topologies, and combines that information with sets of server topology information to produce a unified, overall system topology map.

This paper presents the Network Locality ("netloc") project that provides such a tool to HPC application and middleware developers. Combining hwloc single-server topology data with network topology data, netloc provides a comprehensive view of the HPC system topology, spanning from the processor cores in one server all the way to the cores in another – including the complex network(s) in between. Using a modular infrastructure, netloc is able to read the topology of a variety of network types. Initially, this includes both Ethernet networks managed with OpenFlow [8] and InfiniBand networks. Since large scale HPC systems use a variety of network topologies (e.g., fat-trees, tori, hypercubes), the netloc project uses a portable, abstract graph representation that supports any type of network layout.

The remainder of the paper is organized as follows. Section II introduces the challenges and use cases for providing network topology information. Section III then presents the design of the netloc service while the API is described in Section IV. Implementation details are given in Section V before listing some examples of usage in Section VI, discussing related work, and concluding.

II. Scope of the Problem and Challenges

Before we discuss netloc, we must first understand the composition of the problem space. Overcoming the challenges of building a tool/service such as netloc will allow for more dynamic and tuned application performance on current and future HPC systems.

A. Generic Network Representation

The network topologies in large-scale HPC systems are often complex and involve multiple connections between switches and servers. One source of complexity emerges from the need to support low latency and high bandwidth operations while attempting to reduce congestion in the network. Although the core interconnection scheme can be regular for homogeneous clusters (e.g., fat-trees, clos, torus, dragonfly), it can still contain some irregularity when heterogeneous servers are added (e.g., small subsystem with GPU-enabled servers), which renders the overall system more difficult to model.

Routing through complex network topologies can be managed at different levels of abstraction, depending upon the design and type of network. The physical connections provide the lowest level of connectivity information. Some networks may employ logical routing techniques (possibly at multiple different levels) and additional routing to span multiple subnets within a larger interconnected network. Both the physical and logical routing paths should be available to application developers without having to deal with network physical layer-specific interfaces and tools.

Such information about the network topology may also be useful to network administrators for management purposes. Just like system administrators already use hwloc to quickly assess the topology of a server, the netloc service will allow network administrators to quickly assess the connectivity of networks, including how they map to single-server topologies, without requiring multiple network-specific tools (e.g. `ibnetdiscover` and `ibroute` for InfiniBand, `snmpwalk` for Ethernet, OpenFlow controllers, etc.).

B. Scalability and Failure Management

Link failure is a common issue when interacting with large networks. The topology information reported at any particular point in time is therefore a snapshot of the overall system's current status, and might change shortly thereafter. As such, any tool that provides network information must also provide the ability to give multiple, consistent snapshots of the network topology over time. Applications consuming these snapshots can then analyze and manage the performance impact of such network changes.

Congestion is another common issue that becomes critical when running parallel applications on large HPC systems. Indeed, multistage networks such as fat-trees are known to suffer from congestion even with large theoretical bisection throughput configurations [9]. The effect is magnified if network links are over-subscribed. Reworking process placement among the servers based on the communication scheme has become a common method to help avoid network congestion [10]. Another way to reduce congestion and improve performance is to adapt the communication implementation itself to the network topology at run-time [11], [12]. These research efforts show that the network topology must be known to users, applications, and runtime systems so as to allow applications to better adapt to the actual behavior of the links and switches under pressure.

The third critical issue with network management in large HPC systems is scalability. When mapping topology information for large HPC networks, the resulting graph of information may be quite large. Indeed, storing just the representation of each server (with associated hwloc topology information) on the network can easily take a substantial amount of memory. Traversing and analyzing the network map in an efficient – and possibly distributed – manner while not requiring a large memory footprint is a critical design requirement for netloc.

C. Integration with Server-Internal Topologies

Just as important as network topology, server-internal topology is increasingly being utilized thanks to the widespread use of multicore processors, multiple levels of caches, and NUMA architectures—all of which make process locality critical for performance. Indeed, both network and server-internal topology information are actually used for similar purposes—but no integration of the two has been proposed. For example, network-aware process placement implementations such as LibTopoMap [10] do not take internal server topology into account. hwloc-based placement tools such TreeMatch [13] adapt to server topologies but still manually model the network as a hierarchy. Topology data are also used for better resource allocation in batch-schedulers [7], but, again, server and network topologies are not analyzed and utilized together. We believe that this distinction hinders optimization by forcing algorithms to consider these two sets of data separately. Therefore, there is a need for a global, unified view of the entire HPC system that includes all network and server topologies.

It can be challenging to match the network locality information with the single-server locality information provided by a tool such as hwloc. An application must discover which network interface card (NIC) ports are close to the set of processor cores where it is running, determine to which network subnets those NICs are connected, and then use all that information to determine which route is "best" (e.g., shortest) to a given peer. This matching of locality information can become even more complex when handling multiple network types and architectures. For example, a server is typically identified by a hostname which has no relation to its network identities—the format of which depend on

the specific network type. Additionally, multiple types of logical routing, such as subnets, VLANs, and interacting with non-standardized APIs to Software Defined Networking (SDN) [14] infrastructures make the generation of an overall HPC system map a daunting task for an application developer.[1] Merging network and hwloc information also has the advantage of immediately exposing NIC locality information that can be used for improving multirail communication [15] or hierarchical MPI collectives [16].

Given all of these challenges, we believe that there are many use cases for which providing portable and generic network topology information combined with a set of single-server topologies would be of great benefit. The following sections describe how the netloc tool was designed to address these challenges, and how it can actually be utilized in concrete use cases in Section VI.

III. The Design of the Netloc Service

The netloc service is composed of three components as depicted in Figure 1. First, there are the tools used to access the network topology for various types of networks (called *readers*). Next, there is a mechanism for merging the hwloc single-server topology data and the network topology data into a comprehensive map of the HPC system topology. Finally, there is the C Application Programmer Interface (API) for higher-level software to query and traverse the combined topology information.

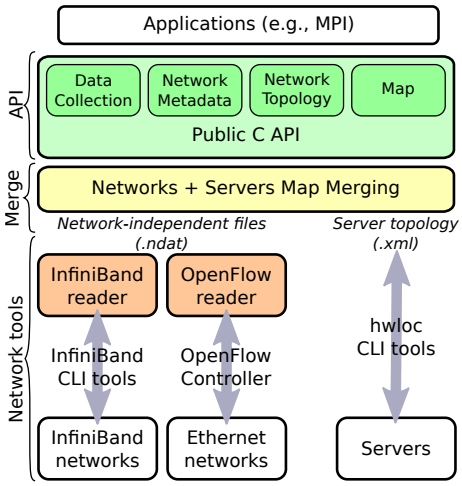

Figure 1. General organization of netloc.

A. Accessing Network Topologies: Readers

Each netloc reader plugin is responsible for discovering, accessing, and reading the network topology for its target network type. Readers generally use network-native tools and mechanisms to gather information about the specified target network.

[1]hwloc does not offer a specific interface for querying network information because it is designed to be a single-server, generic locality tool.

Once the reader has discovered the target network's topology, it invokes the netloc *data collection* API to convert its native network information into a netloc-independent data format, and then save it to a file for later use. The data collection API abstracts the readers away from the underlying data representation and file formats, thereby enabling netloc to decide how to most efficiently represent that data. Each file produced by a reader represents a single subnet of a single network type. Files are timestamped to allow tracking the network's topology state over time. Intermediate files are currently used as a convenient way to pass these snapshots between readers and the core netloc library. A direct connection may replace files in future netloc releases if this storage appears to be a limitation to scalability or flexibility.

netloc initially supports only Ethernet and InfiniBand technologies (see Section V) because they represent the majority of HPC systems in deployment. The architecture of netloc allows for readers to plug into the infrastructure in order to support additional network types and network discovery types. Support for Cray Gemini and Aries interconnects is underway since these technologies offer APIs that can be easily integrated into a netloc reader. Unmanaged Ethernet networks (where OpenFlow cannot be used) are also being studied and, based upon background research, we believe a new reader should be able to gather link and route information through LLDP and SNMP protocols respectively.

B. Preprocessing Network Data

As already described, the first stage of a netloc reader is to discover and process the target network, and then generate a map of the network. An optional second stage (enabled by default) is to preprocess network map graph before writing the final netloc-independent data files for that network.

In this stage, netloc first calculates the shortest path through the network subnet from each server node in the graph to all other server nodes using Dijkstra's algorithm. This information is stored in a separate file allowing the API to access this cached information when needed. Additionally, if logical routing information is available, then logical paths are calculated and stored in another file.

For large networks, these graph algorithms may take a nontrivial amount of time and computational resources. Netloc therefore runs these algorithms immediately after the topology discovery phase in the reader.

Caching pre-processed physical and logical path information in separate files serves two purposes: 1) it allows lazy loading when a netloc-enabled application invokes APIs to load topology information, thus reducing the memory footprint requirements, and 2) the computational load of netloc's library query interface is significantly reduced.

However, one of the netloc project's design decisions was to limit the amount of network analysis that is performed on

the data. Shortest-path discovery is currently the only pre-processing performed on the data. Additional analysis (such as graph partitioning) is left to higher level applications and libraries. As a general design guideline, netloc focuses on the discovery and efficient representation of the HPC topology information.

C. Mapping Network and Server Topologies

In the next stage, the network subnet graphs must be merged with hwloc server topologies to make a unified map of the HPC system.

First, hwloc topology information must be collected from each server, typically by exporting the topology in XML format. This XML is then either saved to a file or transmitted over a network to a central entity. Open MPI [17], for example, gathers hwloc XML information from all servers to `mpirun` at startup.

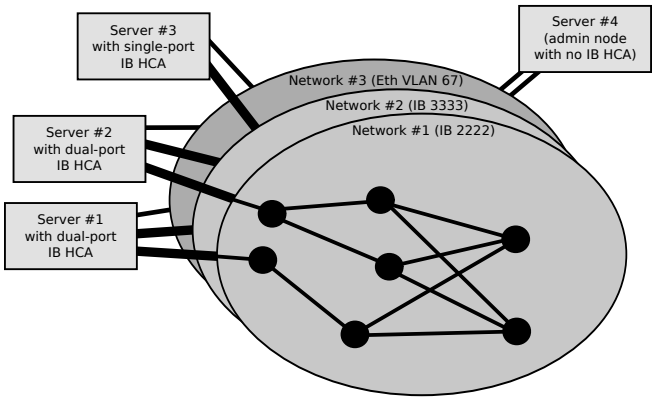

Figure 2. A netloc map comprised of three networks (two IB subnets of IDs 2222 and 3333, and one Ethernet subnet VLAN 67) and four servers. All servers are connected to the Ethernet administration subnet, through either one or two NICs. Compute nodes are also connected to one or two IB subnets via their single or dual-port HCAs.

Once gathered, the hwloc data is fed into netloc. For each network interface on each hwloc-described server, netloc searches subnet maps for matching network identifiers (e.g., Ethernet MAC addresses and InfiniBand GUIDs). Once found, the hwloc and netloc data are joined at the matching graph nodes. In this way, netloc is able to build a unified *map* of the interconnect's servers and disjoint subnets. Figure 2 shows an example map. Each port retains information about the corresponding hardware device and fabric, therefore allowing applications to find out network and physical device attributes behind the netloc logical map.

D. User Interface

Much of netloc's user interface is currently only available through its C API and associated data types. More information about this interface is described in Section IV.

However, netloc currently also provides a command line tool to list all available network information: `lsnettopo`. The `lsnettopo` tool will display either a summary or detailed view of the network information available on the system. Additionally, this tool allows for the export of the network graph into a few standard graph formats: GraphML and GEXF (with a modular infrastructure to add additional export formats). GraphML and GEXF files can then be loaded into visualization software, such as NetworkX [18].

IV. API Structure

The netloc API is divided into four sections, each with a specific purpose. There are also two *handle* and two *graph element* data structures that allow the user to access and manipulate the network topology information.

A. Data Collection

The *data collection* API is used by the readers to create the netloc-independent topology files. This API can also be used by other system services (e.g., resource manager, MPI) to produce files that represent a subset of the network, or to further annotate the network information. By maintaining a flexible data collection API, netloc enables services to interact with the data without the need to have any knowledge of the file format involved. Additionally, by hiding the file format, netloc may optimize when and how the data is written to the file(s).

B. Network Metadata

The *network metadata* API is used by the application to access high level information about the network types and subnets available on the system. This API uses the back-end network-independent files to summarize the information available. An application may use this API to search for a specific type of network or subnet, or to request information about all available networks.

The network metadata API provides the user with a *network handle* datatype that represents a single network type and subnet at a single point in time. The network handle is a lightweight data structure holding basic metadata about the network and is used to access the actual graph later. It also contains versioning information to allow updated network information to be tagged and made available in a structured manner. This information can be seen as a tuple of information: network type, network subnet, and snapshot version.

C. Network Topology Query

The *network topology query* APIs uses the network information contained in the back-end files to recreate the network graph in memory.

The application uses the *network handle* to retrieve the actual network graph composed of *edge* and *node* datatypes. A *node* represents a device on the network – currently either a switch or a NIC port in a server. A *edge* represents a single network connection between two nodes.

It should be noted that a single server may correspond to multiple netloc nodes if it contains multiple NIC ports connected to switches in the network. The mapping API (described next) provides the programmer with the ability to group all of the netloc nodes associated with a single server using hwloc knowledge of server internals.

D. Mapping

The netloc *map* API is built on top of the hwloc API and the main netloc API. A netloc *map* is created by specifying a set of network information files and a set of hwloc topology XML files that are first loaded in to memory, and then merged. A *map* is then made of three main objects:

1) Subnets correspond to network graphs in the main netloc API. Users can retrieve the list of subnets available in a map, and then query them as explained in Section IV-C.

2) Servers correspond to hwloc topologies (hwloc calls its top-level topology a "machine"). Users can consult the list of servers connected to a subnet, lookup by hostnames, etc. This is useful to allow querying and manipulating hwloc topologies with the rich native hwloc API.

3) Ports link the hwloc and netloc graphs together by pointing to both hwloc objects (Ethernet or InfiniBand network interfaces) and netloc edges and nodes. The netloc map API can convert between all of these types. For instance, it can return all netloc edges that are connected to a server, or connected close to one of its processor cores.

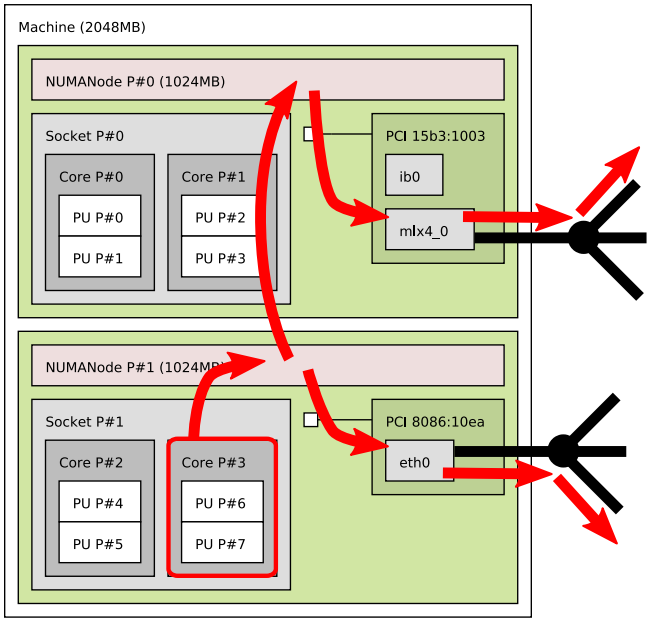

Figure 3. Paths from an hwloc core to a remote object. The Ethernet path goes directly from the local NUMA node to the `eth0` device, and then out to the wire. The InfiniBand path has to cross a NUMA link before it reaches the `mlx4_0` device, and then out to the wire.

There is also a generic type of *edge* that extends network edges to netloc maps. Inside a server, these edges express locality of NIC ports with respect to computing resources: is the NIC close to this processor socket? Or is there a NUMA link to traverse before reaching the inter-server network?

The netloc map API then offers a way to compute all paths between two hwloc objects in two different servers connected by at least one common network. Figure 3 shows a source processor core (`P#3`) and the beginnings of two possible routes to a peer: one through the Ethernet network and the other through the InfiniBand network. This API lets the caller configure whether the list of edges should be exhaustive or not: most users may only be interested in network and cross-NUMA-link edges, but it is also possible to explicitly list other hwloc edges such as PCI links, or edges from processor cores to NUMA nodes.

Finally, the netloc API allows looking for set of servers (or cores) in a neighborhood. It looks at the netloc information to find neighbor servers until it finds enough cores to satisfy the request.

V. Implementation Details

In this section, we discuss the implementation details of the netloc service.

A. Discovery: InfiniBand

The InfiniBand reader currently uses the information returned from OpenFabrics command-line tools to obtain both physical and logical networking topology information. The `netloc_gather_ib` command-line tool simplifies the process of discovering, querying, and loading InfiniBand subnet topologies into netloc. It must be run as root due to access restrictions from the OpenFabrics InfiniBand query tools that it uses.

`netloc_gather_ib` first automatically discovers all IB subnets by analyzing the set of provided hwloc server topologies, regardless of whether all servers are connected to all IB subnets or not. Then, for each discovered InfiniBand subnet, it invokes the InfiniBand tools `ibnetdiscover` to probe the physical connectivity, and `ibroutes` to query logical routing details from each switch.

Once the InfiniBand topologies have been captured, the netloc reader creates the corresponding back-end file by matching the GUID identifiers of the switch and host ports in the outputs of these tools to find out the links and routes within the fabric. Eventually, the reader will use a similar technique to other existing work: utilizing the OpenSM daemon plugin proposed in [6].

B. Discovery: OpenFlow for Ethernet

Although originally designed as a means to perform network research, OpenFlow-based Ethernet network management has been embraced by the networking industry. OpenFlow-managed networks have a management *controller* entity that has global visibility across one or more Ethernet subnets. The controller provides a programmatic interface

to both query and configure various aspects of the subnets that it governs. OpenFlow is an open, standardized protocol to program the various networking elements in an Ethernet environment [8]. The use of OpenFlow is also not confined to a single physical location. For example, ESnet has demonstrated its use in the effective management of large scale provisioning for data transfers between data centers [14]

Commercial OpenFlow controllers, such as the Cisco Extensible Network Controller (XNC)[2], are being deployed in data centers to manage Ethernet networks. Additionally, a growing consortium of networking companies are involved in the OpenDaylight project [19]. OpenDaylight seeks to provide a community-developed OpenFlow controller that supports a wide range of existing switching hardware.

The netloc OpenFlow reader connects to an OpenFlow controller to access physical and logical topology information about a managed Ethernet environment. The netloc OpenFlow reader currently supports the Cisco XNC and two of the leading open-source controllers: Floodlight[3] and OpenDaylight.

Each of the supported OpenFlow controllers provides a public, RESTful interface to query the network topology information. However, the structure of both the public interfaces and the data returned differs from controller to controller. The netloc OpenFlow reader uses the appropriate interfaces for a given controller to extract the switch topology information (physical paths) and current flows (logical paths). Currently, the user of the reader is required to know both the type of controller and its IP address. Although the IP address of the controller might be able to be extracted from the nearest switch (e.g., via SNMP or LLDP), the type of controller is more difficult to automatically determine as there is no standardized reporting mechanism for this information. This presents a usability issue for the netloc OpenFlow reader that is under active investigation.

C. Internal Data Structures

The network metadata structure (described in Section IV-B) provides information about the location of the various files and about the snapshot version of the network if multiple snapshots exist – as they might if the network topology changes due to failures and repairs.

The *edge* and *node* datatypes (described in Section IV-C) are used to represent network topology graphs. Edges are a single network connections between two graph nodes, containing the following pieces of information:

- Pointers to source and destination netloc graph nodes
- Link information (e.g., speed, width)
- Description (provided by the network, or other services)

Nodes represent devices on the network – currently either a switch or a NIC port. It contains the following pieces of information:

- Type of node (e.g., switch, NIC port)
- Physical identifier (e.g., MAC, GUID)
- Logical identifier, if any (e.g., IP address, LID)
- List of pointers to outgoing edges from this node.
- Physical paths to other nodes in the network (precomputed, cached)
- Logical paths to other nodes in the network, if any (precomputed, cached)
- Description (provided by the network, or other services)

The physical and logical paths are read in from the appropriate files, and stored in a lookup table keyed by the physical identifier (e.g., MAC, GUID) of the destination device for efficient access.

Since memory consumption may be of concern in some applications, an additional set of APIs can be used to traverse the map if it is not fully resident in memory. These APIs provide an opportunity for another library (currently future work) to provide a distributed data structure model. The choice between these two styles of traversing the network data is made by the application when considering performance versus memory consumption requirements. We are refining these APIs as on-going work to improve the scalability of the graph representation in a distributed environment.

D. Merging Network Topology with hwloc

Each time a new hwloc topology is loaded, netloc looks for hwloc I/O objects that may be connected to the supported network types. netloc then compares the hwloc object physical and logical identifiers with existing nodes in already-loaded network subnet topologies, looking for a match.

As a generic topology tool, hwloc only offers a rudimentary interface for specifying object attributes. Hwloc stores network attributes such as port identifiers or addresses as pairs of strings such as `(Address,5c:26:0a:53:da:86)` that must be manually parsed by netloc while scanning for a match. While Ethernet MAC addresses are directly available from hwloc and may be immediately compared to network node IDs, InfiniBand GIDs must be parsed into a subnet and node ID before they can be checked for a match.

Once a match is made, a *port* object is created to join the corresponding network graph node and hwloc object. The netloc map API then allows converting between all these types as explained in Section IV-D.

E. Scalability Considerations

Since all network graph nodes and edges are loaded into memory with pointers from one to the other, as described above, netloc must aggressively optimize its memory footprint requirements. For example, instead of maintaining an edge for each physical and logical path between each pair of map peers, a lookup table is used by the outgoing edges and physical/logical path references to link respective edges. Since each edge can be used in many different paths, copying

[2]http://www.cisco.com/go/xnc

[3]http://www.projectfloodlight.org/floodlight/

edge structures for each path would be tremendously space-inefficient.

Additionally, loading thousands of hwloc server topologies may cause scalability issues, given that a single topology easily occupies more than 100 kB of memory. Moreover the difference between two servers in typical HPC systems is usually very small: aside from hardware replacement or BIOS upgrades that change object discovery ordering, most differences consists in device identifiers or serial numbers. We therefore developed a new hwloc extension[4] that allows the *compression* of hwloc topologies by only storing the difference with a reference topology (both in memory and in XML files) [20]. The netloc service then dynamically uncompresses/recompresses the hwloc topologies based on whether their hwloc objects are in use or not.

As an example, 264 server topologies were loaded from the Avakas cluster at the University of Bordeaux facility. Only five distinct hwloc topologies were identified; the remaining 259 topologies were able to be classified as variants of one of the original five. This allowed for a high degree of compression: the memory consumption of the hwloc topologies dropped by a factor of 43, from 103 MB to 2.3 MB. The compression ratio increases with the number of nodes because the amount of distinct topologies does not increase linearly. Further, by ignoring hwloc topology differences that netloc does not need (netloc only requires knowledge of network port differences), memory consumption is reduced even more.

VI. Examples Use Cases

Of the many possible use cases for a unified network and server mapping service, we carefully selected only a few to guide the initial netloc design. Although we anticipate expanding netloc to support more use cases in the future, the use cases described below were used to generate netloc's first set of requirements.

A. MPI Collective and Topology APIs

Multiple areas of MPI functionality can utilize netloc functionality. Developers in the Open MPI community are currently investigating using netloc provide some of the functionality described in this section.

Collective operations: Research has shown that network topology information can be used to optimize MPI collectives by reducing network contention [11], [12] and latency. Single-server topology information can then be added to define multiple levels of neighbors as depicted in Figure 4. This "neighborhood" concept can be used in choosing intermediate processes within hierarchical or binary-tree collective implementations [16], [21]. Further, the same (server + network) neighborhood concept is analogous to the MPI 3 concept of neighborhood collective operations.

[4]The topology diff interface is available starting with the hwloc 1.8 release.

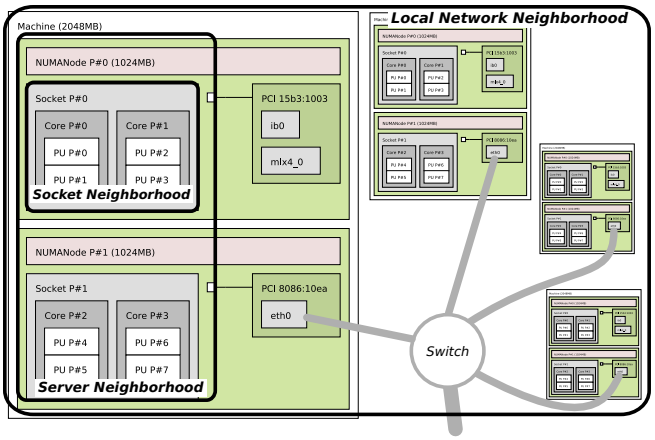

Figure 4. Hierarchy of neighborhoods within the server and the network.

Topologies: Cartesian, graph, and distributed graph topologies are not widely optimized in portable MPI implementations, at least partially because detailed network topology information has not been readily available. With netloc, portable MPI implementations have direct access to the unified map of the entire HPC system (including the subset where the individual MPI job is running). This information can be used to reorder communicators for optimized nearest-neighbor communication by Cartesian dimension or user-specified graphs.

B. Process Placement and Scheduling

Resource allocation and process placement are two related problems that both need to take server and network locality into account. The HPC resource manager can take advantage of network topology information to organize jobs such that they execute in subsets of the overall system that exhibit desirable network characteristics, such as minimizing latency (e.g., keeping all processes in the job within a single leaf switch), maximizing available bisection bandwidth, reduce network contention effects from other jobs, and so on. Once the resources are allocated, processes may then be placed in different ways across those resources that have different impacts on the network and intra-server traffic [7]. Message passing environments such as MPI and Charm++ can use this information to optimize the location of tasks, either at startup or dynamically through MPI topology communicator constructors and Charm++ load balancers.

Netloc's API for managing both network and single-server topologies enables portable generic global placement policy implementations. MPI and Charm++ placement algorithms (e.g., TreeMatch [13]) that currently benefit from single-server topology can provide further optimizations based upon the addition of network topology data. Further, existing network-aware algorithms [6], [10] can be easily extended to exploit single-server topology information.

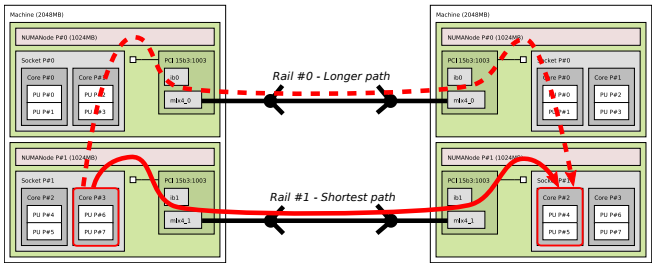

Figure 5. Communication across 2 InfiniBand rails that are connected closer to one of each server's processor sockets.

C. Multi-Path Communication

As the number of processor cores in commodity servers has steadily been trending upward, it is not uncommon for servers in an HPC system to contain multiple NIC ports connected to different switches. So called "multi-path" (or "multi-rail") situations provide a challenge to communication libraries as they have multiple choices as to which interface to send a message. With information about the distance from the source server to the destination server for each of these networks, the communication library can make an informed decision about which interface to use.

Figure 5 presents an example where each server contains one InfiniBand HCA close to each processor socket. Depending on process and HCA locality in each server, the ideal way to spread the traffic across both rails may vary. In the figure, both processes run in the second socket of their respective server. They have faster access to the second InfiniBand subnet due to their immediate proximity to rail #1. The netloc map API exposes that information by offering one shortest path through rail #1, and a longer path through rail #0. Therefore a MPI implementation may want to send most small, latency-sensitive messages on rail #1. Large messages may be split across both rails—but rail #1 should likely take care of a larger share [15].

If the processes do not run on cores close to network interfaces in the same subnet, both rails may be used equally. However, the locality of the buffer where data is written is more important on some AMD platforms [21]. Such effects may cause one rail to be preferred depending on the transfer direction.

Finally, if the network path (number of switches or link speed) through both rails is different, the intra-server locality may become negligible. As explained in Section IV-D, the netloc map API offers an easy way to compute these global paths to ease locality-aware communication optimization.

VII. Related Work

As the complexity of HPC systems increases, multiple research projects started focusing on understanding the topology. The intra-server topology is widely used as a key criteria for process placement thanks to tools as hwloc [1] and algorithms such as TreeMatch [13]. On the network side, Hoefler and Snir fully describe the topology mapping problem for applications seeking to harness topology information to optimize communication strategies [10]. In their discussion they briefly discuss the mechanisms available to access network topology information for various HPC targeted interconnects such as SeaStar (Cray XT), DCMF (BlueGene), Myrinet, and InfiniBand. They developed custom tools to access this information to support their analysis in the paper. Netloc focuses just on the exposing of the network topology information in an abstract manner, and not necessarily on the mapping problem. As such, netloc will take care of supporting those networks and others, but leave addressing the topology mapping problem to higher level libraries.

Subramoni, et al. describe how to use an InfiniBand network topology detection service to improve process placement [6]. The topology service is connected to a plugin in the OpenSM subnet manager to access InfiniBand topology information. Prior work by the same authors used the `ibnetdiscover` and `ibroutes` tools (that netloc currently uses) to access the topology information [3]. The topology discovery service only supports the exposure of tree-structured networks, and does not combine that network information with the single-server topology information. Moreover the code of this service is still not publicly available. Netloc, by using a generic graph representation, is able to support any network graph and combines the network topology with the single-server topology information from hwloc. It currently does not use the OpenSM subnet manager, but as part of our on-going work we are planning to integrate this OpenSM plugin.

That topology service has also been integrated as a plugin to the SLURM resource managers for refining the list of servers that may be used for [7]. Conflict lists are built by looking at shared network links, and updated by querying the network topology detection periodically to account for network events (switches or links going down). This work is specific to InfiniBand but we believe the idea is even more important for some slower networks such as Ethernet where contention occurs more frequently. Therefore we believe that providing such information in a generic manner would be a great benefit to batch resource managers.

The networking community has been researching techniques for discovering the physical and logical topology of Ethernet networks for many years [22]. Some of the research requires a network to be composed of (either entirely or predominately) managed switches providing access to either Simple Network Management Protocol (SNMP) or Link Layer Discovery Protocol (LLDP) information [23], [24]. These algorithms query the switches for the data and then use that to reconstruct the paths between these switches and servers in the network. Other research has been able to avoid this requirement if each server has an active agent on it

to listen for marker packets traversing the network in an orchestrated manner [25]. Research into using performance modeling to reconstruct the network topology has also been conducted using MPI in an HPC environment, although the supported networks are limited to tree structures [26]. Depending upon the technique used either the physical topology or the logical/routing topology is exposed. The netloc service currently uses OpenFlow for topology information (both logical and physical) because we believe future data centers will support it. But on-going work into the application of these techniques for discovering the Ethernet topology with minimal (or no) switch support is also underway.

Aside of accurate network representation based on the actual discovery of links and switches, some approach are rather based on analytical modeling of the network. Clauss et al. determine affine cost functions and a contention model from MPI benchmark outputs [27]. They use them in the SimGrid simulation tool for predicting parallel application performance as part of a more general network representation framework [28] for scalable large scale simulation. Netloc currently only embeds physical topology and logical routing information but we believe that adding such analytical information about the actual network behavior under load may enhance future uses.

Finally, one could argue that hwloc already has an interface for assembling multiple servers into a single cluster-wide topology with level of switches (the *Custom* or *Multi-node* interface). However, it is restricted to a tree structure which is not generic enough. Although hwloc knowledge of distances solves this issue when describing NUMA interconnects, it fails to describe large network graphs accurately. Moreover hwloc only allows assembling servers within a single network and the actual NIC locality cannot be matched with that assembly. The netloc service solves all of these issues by allowing network links to be directly connected to hwloc NIC objects.

VIII. Conclusion and Future Work

Due to the increasing complexity of HPC server architectures and networks topology- and affinity-awareness has become a critical component of HPC application optimization. The netloc project, presented in this paper, provides mechanisms for discovering and abstractly representing the HPC network topology, and further merges this information with the server-internal topology (provided by hwloc) to provide a comprehensive view of the HPC system topology.[5]

Divided into three components, netloc first provides tools to discover the network topology for a variety of networks using a modular infrastructure. Then, netloc provides the ability to merge network and server-internal topologies into a single unified map of the HPC system. Finally, netloc provides a portable, abstract representation of the network topology and unified map (represented as a graph) through the netloc C API.

This paper discussed a number of applications and problem domains for which the information provided by netloc can have a significant impact. Whether leveraged by research into resource allocation, process placement, or communication optimization, the portable comprehensive representation of the HPC system topology provided by netloc allows this research to more easily move between HPC systems expanding their impact on the HPC community.

The netloc project is still under active development with a number of active areas of on-going work. More network discovery readers are in development including support for non-OpenFlow Ethernet networks (e.g., via SNMP and LLDP), and Cray Gemini and Aries networks. The current network discovery mechanism is largely offline. We are investigating a daemon-based approach to provide live network updates via a callback mechanism. Finally, scalability to large HPC system is one of the primary goals of the netloc project. As such, we are investigating techniques to reduce the memory footprint and computational complexity of the library to support improved application efficiency at large scale.

[5] netloc is available at http://www.open-mpi.org/projects/netloc/.

IX. Acknowledgements

Some of the network topologies used in this study were provided by the computing facilities MCIA (Mésocentre de Calcul Intensif Aquitain) of the Université de Bordeaux and of the Université de Pau et des Pays de l'Adour.

Special thanks to Douglas MacFarland and Nicholas Buroker for their work on early prototypes of this project.

Research supported by the University of Wisconsin–La Crosse (UW-L) Faculty Research Grant program, and by the UW-L Computer Science Department. Additional computing resources and topology information provided by Cisco Systems, Inc.

References

[1] 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 *Proceedings of the 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP2010)*. Pisa, Italia: IEEE Computer Society Press, Feb. 2010, pp. 180–186.

[2] R. Graham, M. Venkata, J. Ladd, P. Shamis, I. Rabinovitz, V. Filipov, and G. Shainer, "Cheetah: A Framework for Scalable Hierarchical Collective Operations," *11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pp. 73–83, 2011.

[3] 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 *International Conference on Cluster Computing (IEEE Cluster)*, 2011, pp. 317–325.

[4] A. Bhatele and L. V. Kale, "An evaluative study on the effect of contention on message latencies in large supercomputers," *IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pp. 1–8, 2009.

[5] S. Ethier, W. M. Tang, R. Walkup, L. I. J. o. R. Oliker, and Development, "Large-scale gyrokinetic particle simulation of microturbulence in magnetically confined fusion plasmas," *IBM Journal of Research and Development*, vol. 52, no. 1, Jan. 2008.

[6] H. Subramoni, S. Potluri, K. Kandalla, B. Barth, J. Vienne, J. Keasler, K. Tomko, K. Schulz, A. Moody, and D. K. Panda, "Design of a Scalable InfiniBand Topology Service to Enable Network-Topology-Aware Placement of Processes," in *Proceedings of the 2012 ACM/IEEE conference on Supercomputing*, Salt Lake City, UT, Nov. 2012.

[7] H. Subramoni, D. Bureddy, K. Kandalla, K. Schulz, B. Barth, J. Perkins, and M. Arnold, "Design of Network Topology Aware Scheduling Services for Large InfiniBand Clusters," in *International Conference on Cluster Computing (IEEE Cluster)*, Indianapolis, IN, Sep. 2013.

[8] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, p. 69, Mar. 2008.

[9] T. Hoefler, T. Schneider, and A. Lumsdaine, "Multistage Switches are not Crossbars: Effects of Static Routing in High-Performance Networks," in *Proceedings of the 2008 IEEE International Conference on Cluster Computing*. IEEE Computer Society, Oct. 2008.

[10] 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.

[11] 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. Springer-Verlag, 2011, pp. 50–60.

[12] B. Prisacari, G. Rodriguez, C. Minkenberg, and T. Hoefler, "Bandwidth-optimal Alltoall Exchanges in Fat Tree Networks," in *Proceedings of the 2013 ACM International Conference on Supercomputing (ICS'13)*. ACM, Jun. 2013.

[13] E. Jeannot, G. Mercier, and F. Tessier, "Process placement in multicore clusters: Algorithmic issues and practical techniques," *IEEE Transactions on Parallel and Distributed Systems*, 2013, to appear.

[14] I. Monga, E. Pouyoul, and C. Guok, "Software-Defined Networking for Big-Data Science - Architectural Models from Campus to the WAN," *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion*, pp. 1629–1635, 2012.

[15] S. Moreaud, B. Goglin, and R. Namyst, "Adaptive MPI Multirail Tuning for Non-Uniform Input/Output Access," in *Proceedings of the 17th European MPI Users' Group conference on Recent advances in the Message passing Interface (EuroMPI)*, vol. 6305. Stuttgart, Germany: Springer-Verlag, Sep. 2010, pp. 239–248.

[16] T. Ma, G. Bosilca, A. Bouteiller, and J. J. Dongarra, "HierKNEM: An Adaptive Framework for Kernel-Assisted and Topology-Aware Collective Communications on Many-core Clusters," in *Proceedings of the 26th International Parallel and Distributed Processing Symposium (IPDPS'12)*, Shanghai, China, May 2012.

[17] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004, pp. 97–104.

[18] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using NetworkX," in *Proceedings of the 7th Python in Science Conference (SciPy2008)*, Pasadena, CA USA, Aug. 2008, pp. 11–15.

[19] OpenDaylight Project, "OpenDaylight – An open source community and meritocracy for software-defined networking," April 2013, white paper. [Online]. Available: http://www.opendaylight.org

[20] B. Goglin, "Managing the Topology of Heterogeneous Cluster Nodes with Hardware Locality (hwloc)," in *Proceedings of 2014 International Conference on High Performance Computing & Simulation (HPCS 2014)*, Bologna, Italy, Jul. 2014. [Online]. Available: http://hal.inria.fr/hal-00985096

[21] B. Goglin and S. Moreaud, "Dodging Non-Uniform I/O Access in Hierarchical Collective Operations for Multicore Clusters," in *Workshop on Communication Architecture for Scalable Systems (CASS), held in conjunction with IPDPS*. Anchorage, AK: IEEE Computer Society Press, May 2011.

[22] B. Donnet and T. Friedman, "Internet topology discovery: A survey," *IEEE Communications Surveys & Tutorials*, vol. 9, no. 4, pp. 56–69, 2007.

[23] Y. Breitbart, M. Garofalakis, B. Jai, C. Martin, R. Rastogi, and A. Silberschatz, "Topology discovery in heterogeneous IP networks: The NetInventory system," *IEEE/ACM Transactions on Networking*, vol. 12, no. 3, pp. 401–414, 2004.

[24] J. Farkas, V. G. de Oliveira, M. R. Salvador, and G. C. dos Santos, "Automatic Discovery of Physical Topology in Ethernet Networks," in *International Conference on Advanced Information Networking and Applications (AINA)*, 2008.

[25] R. Black, A. Donnelly, and C. Fournet, "Ethernet topology discovery without network assistance," in *Proceedings of the 12th IEEE International Conference on Network Protocols (ICNP)*, Jan. 2004.

[26] J. Lawrence and X. Yuan, "An MPI tool for automatically discovering the switch level topologies of Ethernet clusters," *IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pp. 1–8, 2008.

[27] P.-N. Clauss, M. Stillwell, S. Genaud, F. Suter, H. Casanova, and M. Quinson, "Single Node On-Line Simulation of MPI Applications with SMPI," in *Proceedings of the International Parallel & Distributed Processing Symposium (IPDPS)*. Anchorange, USA: IEEE, May 2011.

[28] L. Bobelin, A. Legrand, D. A. G. Màrquez, P. Navarro, M. Quinson, F. Suter, and C. Thiery, "Scalable Multi-Purpose Network Representation for Large Scale Distributed System Simulation," in *Proceedings of the 12th IEEE International Symposium on Cluster Computing and the Grid (CCGrid12)*. Ottawa, Canada: IEEE Computer Society Press, May 2012.