Designing a Scalable Network Analysis and Monitoring Tool with MPI Support

A Thesis

Presented in Partial Fulfillment of the Requirements for the Degree Master of Science in the Graduate School of The Ohio State University

By

Albert Mathews Augustine, B. E.

Graduate Program in Computer Science and Engineering

The Ohio State University

2016

Master’s Examination Committee:

Dr. D. K. Panda, Advisor
Dr. Radu Teodorescu
Dr. Hari Subramoni
Abstract

State-of-the-art high-performance computing is powered by the tight integration of several hardware and software components. While on the hardware side, we have multi-/many core architectures (including accelerators and co-processors) and high end interconnects (like InfiniBand, Omni-Path), on the software front we have several high performance implementations of parallel programming models which help us to take advantage of the advanced features offered by the hardware components. This tight coupling between both these layers helps in delivering the multi-petaflop level performance to the end application allowing scientists/engineers to tackle the grand challenges in their respective areas.

Understanding and gaining insights into the performance of the end application on these modern systems is a challenging task. Several tools have been developed to inspect the network level or MPI level activities to address this challenge. However, these existing tools inspect the network and MPI layer in a disjoint manner and are not able to provide a holistic picture correlating the data generated for network layer and MPI. Thus, the user can miss out on critical information which could have helped in understanding the interaction between MPI applications and the network they are running on. In this thesis, we take up this challenge and design OSU INAM. OSU INAM allows users to analyze and visualize the communication happening in the network in conjunction with the data obtained from the MPI library. Our experimental analysis shows that the tool is able to profile and visualize the communication with very low performance overhead at scale.
Acknowledgments

I would like to express my deepest gratitude to my advisor Dr. D. K. Panda for giving me the opportunity to work in his research group and the support he has given to me throughout my master degree. I am deeply indebted to him for his guidance and advice, and it has been a privilege to work with him.

I would like to acknowledge and thank Dr. Radu Teodorescu for agreeing to be part of my thesis defense committee.

I would also like to acknowledge and thank Dr. Hari Subramoni for being a great mentor and for guiding and encouraging me patiently throughout my research. Special thanks to Mark Arnold and Jonathan Perkins - the two most helpful people I have met, helping me troubleshoot many issues and providing valued and constructive suggestions. I am also grateful to Dr. Xiaoyi Lu for his advice and suggestions.

I would like to thank my colleagues Khaled Hamidouche, Jian Lin, Dip Sankar, Akshay Venkatesh, Ammar Ahmad Awan, Mingzhe Li, Sourav Chakraborty, Ching-Hsiang Chu, Jie Zhang, Jahanzeb Hashmi and Mamzi for all their support.

In the last couple of years, I was fortunate to meet some really nice people and make good friends. I would like to thank Gagan, Meraj, Sridhar, Kaushik, Ganga and Aditya for all their support and help.

Finally, I wouldn’t be what I am today without my parents, my sister, and brother, and my friends who are my greatest support and source of inspiration.
Vita

December 14, 1989                      Born - Mumbai, India

2011                                B.E., Information Technology,
                                      University of Mumbai,
                                      Mumbai, India.

2011-2014                          Software Engineer,
                                      JP Morgan Services India Pvt. Ltd.,
                                      Mumbai, India

2015-2016                          Graduate Research Associate,
                                      The Ohio State University

Fields of Study

Major Field: Computer Science and Engineering

Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>iii</td>
</tr>
<tr>
<td>Vita</td>
<td>iv</td>
</tr>
<tr>
<td>List of Tables</td>
<td>vii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>viii</td>
</tr>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Motivation</td>
<td>2</td>
</tr>
<tr>
<td>1.2 Problem Statement</td>
<td>4</td>
</tr>
<tr>
<td>2. Background and Related Work</td>
<td>7</td>
</tr>
<tr>
<td>2.1 InfiniBand - An Overview</td>
<td>7</td>
</tr>
<tr>
<td>2.2 Message Passing Interface</td>
<td>8</td>
</tr>
<tr>
<td>2.2.1 MVAPICH2</td>
<td>8</td>
</tr>
<tr>
<td>2.3 Omni-Path Network Architecture</td>
<td>9</td>
</tr>
<tr>
<td>2.4 OpenFabrics Enterprise Distribution</td>
<td>9</td>
</tr>
<tr>
<td>2.4.1 Mellanox OpenFabrics Enterprise</td>
<td>10</td>
</tr>
<tr>
<td>2.5 MPI Tool Information Interface</td>
<td>10</td>
</tr>
<tr>
<td>2.6 SLURM</td>
<td>10</td>
</tr>
<tr>
<td>2.7 PhantomJS</td>
<td>11</td>
</tr>
<tr>
<td>2.8 VisJs</td>
<td>11</td>
</tr>
<tr>
<td>2.9 Related Tools</td>
<td>11</td>
</tr>
</tbody>
</table>
3. Design of InfiniBand Network Analysis and Monitoring Tool

3.1 OSU INAM Daemon

3.1.1 Fabric Discovery Thread
3.1.2 Database Thread
3.1.3 MPI Data Collection Thread

3.2 OSU INAM Database

3.3 Java Web Server Application and Web Based Front End Application

3.4 Design Enhancements for Scalability For Large Scale HPC Systems

3.4.1 Network and MPI Load Generator
3.4.2 Optimized Network Rendering using PhantomJS and Clustering technique
3.4.3 Designs to Enhance Data Storage Performance

3.5 Re-designing the Data Collection Daemon to Handle Multiple Networks

4. Discussion of Features and Experiment Results

4.1 Discussion on Features of OSU INAM and its Impact

4.1.1 Network Level Monitoring
4.1.2 Job Level Monitoring

4.2 Experimental Results

4.2.1 Experimental Setup
4.2.2 Impact of Co-design of MPI Runtime with OSU INAM on the MPI application
4.2.3 Impact on OSU INAM post Scalability Enhancement Design

5. Conclusions and Future Work

5.1 Summary of Contributions

5.1.1 Design and Features of OSU INAM
5.1.2 Enhancing OSU INAM for Large Scale Clusters

5.2 Software Release and its Impact

5.3 Future Work

Bibliography
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Summary of additional MySQL optimizations</td>
<td>35</td>
</tr>
<tr>
<td>4.1 Comparison of communication buffer utilization for default and tuned scenarios for 512-process class D NAS parallel benchmarks</td>
<td>41</td>
</tr>
<tr>
<td>4.2 Details of various HPC systems</td>
<td>46</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Overall framework of proposed design</td>
<td>14</td>
</tr>
<tr>
<td>3.2 Overview of OSU INAM database design</td>
<td>21</td>
</tr>
<tr>
<td>3.3 Overall of OSU INAM web server and front-end</td>
<td>22</td>
</tr>
<tr>
<td>3.4 Full network view of Stampede@TACC and Comet@SDSC systems as depicted by OSU INAM</td>
<td>30</td>
</tr>
<tr>
<td>3.5 Full network view of Gordon@SDSC and OSC Clusters as depicted by OSU INAM</td>
<td>31</td>
</tr>
<tr>
<td>3.6 Proposed network abstraction API</td>
<td>36</td>
</tr>
<tr>
<td>4.1 Network view of a cluster</td>
<td>39</td>
</tr>
<tr>
<td>4.2 Link utilization: process level view</td>
<td>43</td>
</tr>
<tr>
<td>4.3 Live jobs page</td>
<td>44</td>
</tr>
<tr>
<td>4.4 Job level view</td>
<td>45</td>
</tr>
<tr>
<td>4.5 Node level view</td>
<td>45</td>
</tr>
<tr>
<td>4.6 Microbenchmark-level point-to-point (Latency) performance</td>
<td>47</td>
</tr>
<tr>
<td>4.7 Microbenchmark-level point-to-point (Message Rate) performance</td>
<td>48</td>
</tr>
<tr>
<td>4.8 Microbenchmark-level collective performance at 512 processes (Performance of Broadcast)</td>
<td>48</td>
</tr>
</tbody>
</table>
4.9 Microbenchmark-level collective performance at 512 processes (Performance of Alltoall) .................................. 49

4.10 Performance of class D NAS parallel benchmarks at 512 processes ............... 49

4.11 Performance breakup of UI load time .................................................. 51

4.12 Impact of MySQL optimization on performance of insert operations ........ 53
Chapter 1: INTRODUCTION

Across scientific domains, application scientists are constantly looking to push the envelope by running large-scale, parallel jobs on supercomputing systems. These 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. These systems employ thousands of computing units connected over a very high-speed interconnect making maximum use of the computing power at hand to provide multi-peta/exaflop calculations per second of performance. To enable scientific parallel applications to scale and perform well on exascale supercomputing systems, a significant focus is being given to enhance the performance and scalability of every component of a high performance computing (HPC) system both hardware and software. The vendors / industry partners are focusing on the hardware aspects through the introduction of novel multi-/many-core environments (Intel Xeons, Xeon Phis, NVIDIA GPGPUs and upcoming POWER architectures) and Remote Direct Memory Access (RDMA) enabled high-performance networking technologies like InfiniBand (IB) [22], and Omni-Path [23]. On the other hand, various researchers around the globe are looking at techniques to optimize different parts of the software architecture for these systems. It is also expected that the programming model of choice for developing high-performance scientific applications capable of running on
these systems will be based on a combination of the popular Message Passing Interface (MPI) [28] programming model and one or more of the multiple existing programming models currently in use or under development [25].

1.1 Motivation

As the modern networking hardware being used to deploy the emerging supercomputing systems and the high-performance MPI-based applications that use these systems become increasingly complex, understanding and analyzing the interplay between these components and reasoning about the performance obtained by these applications becomes a challenging task. Such an understanding of the interaction between interrelated components is critical not only for developers of HPC applications and the high performance MPI middleware they depend on, but also for the administrators of the emerging “pre-exascale” systems. Such understanding will enable all involved parties (application developers/users, system administrators, and MPI runtime developers) to maximize the efficiency and performance of the various individual components that comprise a modern HPC system and solve the various “grand challenge” problems. Unfortunately, relatively little focus is being given to this critical yet underrated component of the HPC ecosystem.

One of common questions system administrators tend to get from the users of the clusters they are managing is: “Why is my application running slower than usual now?” Interaction with a concurrent job in the network or network-based parallel file system is the most common cause for this behavior. Several tools exist in literature and as products which allow system administrators to analyze and inspect the IB fabric (eg: Nagios [7], Ganglia [1], Mellanox Fabric IT [35], BoxFish [14]). However, due to the lack of interaction with, and
knowledge about the MPI library, no existing IB fabric monitoring tool can correlate network level and MPI level behavior to classify traffic as belonging or being generated by particular MPI primitives (e.g.: Point-to-point, Collective, RMA). Furthermore, they cannot classify network traffic as belonging to a particular job due to the lack of interaction with the job scheduler. Such classification would allow the system administrators to pinpoint the source of the conflict at a much finer granularity than what is possible with the existing set of tools. Current generation high-performance MPI runtimes are complicated pieces of software with hundreds of performance-oriented features and knobs (e.g.: support for different high performance transport protocols, support for different collective communication algorithms and mechanisms, network topology aware communication, the hardware offloaded communication, network hot spot avoidance). Some of these features have interdependencies and interactions with others. While the default setting of these features will deliver about 80% of the maximum achievable performance in most cases, careful application specific tuning is required to extract that last 20% of performance. This requires in-depth understanding of the workings of the MPI library and how it interacts with the underlying communication fabric. Existing MPI level profiling tools (TAU [9], HPC-Toolkit [21], Intel VTune [17], IPM [3], mpiP [4]) give reasonable insights into the MPI communication the behavior of applications. However, they have no knowledge about the underlying IB fabric and thus are not able to correlate network level and MPI level behavior to identify that increased traffic levels on one link are causing performance degradation for an MPI job whose communication is going over said link.

Furthermore, most existing MPI profiling tools are unable to provide deep insights into the operations of the MPI library due to the lack of an interface that allows them to interact with the MPI library and identify the behavior of various internal components. The MPI
The MPI-T interface allows tools to interact with the MPI library through control and performance variables. Researchers have already begun to take advantage of this interface to provide optimization and tuning hints to the users [19]. However, these tools have no knowledge about the underlying network fabrics and thus suffer from the same drawbacks as other existing MPI tools.

As we can see, there is a gap in the support provided by existing network as well as MPI level profiling tools which must be filled. Any tool that is able to bridge this gap will enable end users to correlate the behavior of the network fabric and the MPI runtime to gain true insights into the performance being delivered by high-performance scientific applications. These issues lead us to the following broad challenge of designing a tool that enables the in-depth understanding of the communication traffic on the InfiniBand network through tight integration with the MPI runtime. Such a tool must: 1) be portable, easy to use understand, 2) have high performance and scalable rendering and storage techniques and 3) be applicable to the different network interconnects and high-performance MPI libraries that are likely to be used on existing large supercomputing systems and emerging “pre-exascale” systems.

1.2 Problem Statement

Currently there is no tool which is capable of monitoring the network along with MPI jobs providing information about their interaction. We take up the challenge and design OSU INAM - a low-overhead profiling and visualization tool that is capable of presenting the profiling information obtained from the network and the MPI library in conjunction to
allow users to gain more insights than afforded by existing tools that profile/visualize the network/MPI disjointly.

The thesis is broadly divided into two parts dealing first dealing with the problem of designing and implementing a tool addressing the issues mentioned in the earlier sections. Second, exploring techniques and mechanism to resolve scalability issues for it to be usable in emerging exascale system.

We try to demonstrate how, through the profiling information provided by OSU INAM, administrators and users of the HPC installations, as well as, developers of high-performance middleware, can gain more insights into the communication characteristics of their run-times allowing them to further fine tune the performance on a per application/per run basis.

We try to show how, through the link analysis system administrators can pinpoint the cause of network performance issues to a granularity of a process. Our experimental evaluation shows that the OSU INAM is able to profile and visualize the communication with very low performance overhead at scale. To summarize, we focus on the following major features:

- Analyze and profile network-level activities with many metrics (data and errors) at user specified granularity
- Capability to analyze and profile node-level, job-level and process-level activities for MPI communication (Point-to-Point, Collectives and RMA)
- Capability to profile and report several metrics of MPI processes at node-level, job-level and process-level at user specified granularity in conjunction with the MPI runtime
- Capability to analyze and classify the traffic flowing in a physical link into those belonging to different jobs in conjunction with the MPI runtime
- Capability to visualize the communication map at process level and node level granularities in conjunction with the MPI runtime

- “Job Page” to display jobs in ascending/descending order of various performance metrics in conjunction with the MPI runtime

This thesis also explores intelligent designs that move the cost of rendering large networks out of the critical path and presents high-performance designs and optimizations for database access that increase the data storage performance many-fold. Novel abstraction interfaces are also proposed that enhances the applicability of the tool to multiple high-performance networks. The major contributions are:

- Significantly enhance the network rendering time seen by the user by pre-computing and caching the network graph using PhantomJS [13]

- Lock-free designs and optimized database schema for enhanced performance

- Optimize MySQL parameters for reduced insertion and query times

- Create modular designs to enable discovery and profiling of multiple network fabrics (InfiniBand and Omni-Path)

- Create a simple load generator to stress the various components of the tool — the data collection daemon, the database and the UI
Chapter 2: BACKGROUND AND RELATED WORK

In this chapter, we provide a brief background related to our work. We begin with a brief introduction to some of the technologies/libraries we would be exploring. Later we discuss a few web technologies which we would be using. Finally, we take a look at the past and ongoing related work in our area.

2.1 InfiniBand - An Overview

InfiniBand Architecture (IBA) [2] defines a switched network fabric for interconnecting processing nodes and I/O nodes, using a queue-based model. The I/O devices and compute nodes are connected to the switch fabric using one or more channel adapters (CA). Every fabric may consist of one or more subnets which are interconnected by routers. Each subnet is managed in an autonomous way. InfiniBand specifications do not enforce the usage of any specific network topology or routing algorithm. IB has multiple transport services including Reliable Connection (RC) and Unreliable Datagram (UD), and supports two types of communication semantics: Channel Semantics (Send-Receive communication) over RC and UD, and Memory Semantics (Remote Direct Memory Access - RDMA communication) over RC. Both semantics can perform zero-copy data transfers, i.e, the data can directly be transferred from the application source buffers to the destination buffers without additional host level memory copies. IB also proposes link layer Virtual Lanes (VL)
that allows the physical link to be split into several virtual links, each with their specific buffers and flow control mechanisms. This possibility allows the creation of virtual networks over the physical topology. InfiniBand is being used by almost 41\% of the Top500 Supercomputing systems [34] according to the June’16 listing.

2.2 Message Passing Interface

The Message Passing Interface (MPI) [28] has been the de-facto parallel programming model for the past two decades. MPI has been widely adopted across various scientific domains and is very successful in implementing regular, iterative parallel algorithms with well-defined communication patterns. MPI libraries provide basic communication support for a parallel computing job. In particular, several convenient point-to-point and collective communication operations are provided. High-performance MPI implementations are closely tied to the underlying network dynamics and try to leverage the best communication performance on the given interconnect. The advantages of developing message passing software using MPI closely match the design goals of portability, efficiency, and flexibility. MPI has been widely adopted across various scientific domains and is very successful in implementing regular, iterative parallel algorithms with well-defined communication patterns.

2.2.1 MVAPICH2

The MVAPICH2 software [5] supports MPI 3.1 standard, delivering the best performance, scalability and fault tolerance for high-end computing systems and servers using InfiniBand, Omni-Path, Ethernet/iWARP, and RoCE networking technologies. This software is being used by more than 2,625 organizations in 81 countries worldwide to extract the potential of these emerging networking technologies for modern systems. As of Jul
’16, more than 381,000 downloads have taken place from this project’s site. This software is also being distributed by many vendors as part of their software distributions.

The MVAPICH2 software is powering several supercomputers in the TOP500 list. Examples (from the June ’16 ranking) include: 1) 12th, 462,462-core (Stampede) at TACC 2) 15th, 185,344-core (Pleiades) at NASA 3) 31st, 74,520-core (Tsubame 2.5) at Tokyo Institute of Technology

2.3 Omni-Path Network Architecture

Intel’s Omni-Path Fabric [23] defines three layers. The Link Transfer Protocol supports traffic flow optimization, packet integrity protection, and dynamic lane switching. The Link Layer supports 24-bit fabric addresses, congestion management, and QoS support. Lastly, the Data Link Layer enables fabric addressing, switching, resource allocation, and partitioning support. Additionally, each device contains a Performance Manager Agent (PMA) with 64-bit counters.

2.4 OpenFabrics Enterprise Distribution

The OpenFabrics Enterprise Distribution (OFED) [18] is open-source software for RDMA and kernel bypass applications. It includes kernel-level drivers, channel-oriented RDMA and send/receive operations, kernel bypasses of the operating system, both kernel, and user-level application programming interface (API) and services for parallel message passing (MPI) and storage and file system/database systems. It supports the following network and fabric technologies providing RDMA performance: legacy 10 Gigabit Ethernet, iWARP for Ethernet, RDMA over Converged Ethernet (RoCE), and 10/20/40/100 Gigabit InfiniBand.
2.4.1 Mellanox OpenFabrics Enterprise Distribution for Linux

Mellanox OFED is OFED implementation provided by Mellanox technologies. It supports two interconnect types: InfiniBand and Ethernet which uses RDMA and kernel bypass APIs.

2.5 MPI Tool Information Interface

The MPI Tool Information Interface (MPI_T) provides a standard mechanism for MPI tool developers to both inspect and modify the various internal settings and performance characteristics of MPI libraries. The MPI_T interfaces define two types of objects. The first type of object is the performance variable. It allows to determine the state of the MPI library and how it is being affected by the MPI application. The second type of object is the control variable. This type of object is tied to a modifiable parameter of the MPI library. Accessing and modifying these will allow the software to change the behavior of the MPI library. The MPI tool information interface provides the necessary routines to find all variables that exist in a particular MPI implementation, to query their properties, to retrieve descriptions about their meaning, and to access and, if appropriate, to alter their values. This interface is primarily intended for performance monitoring tools, support tools, and libraries controlling the applications environment.

2.6 SLURM

SLURM [8] is an open-source resource manager designed for Linux clusters of all sizes. It provides three key functions. First, it allocates exclusive and/or non-exclusive access to resources (computer nodes) to users for some duration of time so they can perform work. Second, it provides a framework for starting, executing, and monitoring work (typically a
parallel job) on a set of allocated nodes. Finally, it arbitrates contention for resources by managing a queue of pending work.

2.7 PhantomJS

PhantomJS [13] is a headless web browser without a graphical user interface used for test automation and simulation. It supports various web standards: (Document Object Model(DOM) handling, CSS selector, JSON, Canvas, and SVG).

2.8 VisJS

VisJS [11] is a popular visualization library for browser-based rendering of different types of data sets including large networks, timeline based displays, 2D / 3D graphs as well as unstructured data sets. It handles the physics simulation, moving the nodes and links to show them clearly. This capability in vis.js is used to determine the relative positions of the nodes and links in the network. The simulation relies on the Barnes-Hut model which is a quadtree based gravity model. This is the fastest, default and recommended solver for visualizing non-hierarchical layouts.

2.9 Related Tools

Lightweight Distributed Metric Service (LDMS) [10] tries to provide a low overhead system which correlates MPI activities impact on the system. It provides regarding filesystem I/O activities, memory usage, CPU utilization, However, it does not monitor the IB or Omni-Path fabric as OSU INAM proposes to do. Similarly, HOPSA [12] is another tool which provides various metrics related to MPI focusing on the application. But, this tool does not provide a system-wide view and how various jobs can have an impact on each
other. TACC STATS [32], a tool developed by Texas Advance Computing Center is able to
provide information for application level profiling to explore job along with system usage.
However, Unlike OSU INAM which can provide this information in real time, users can
only analyze using TACC STATS post execution.

There are many tools that exist, which a system administrator can employ to monitor
an IB fabric and to analyze different network metrics. Nagios, Ganglia, Mellanox Fabric
IT and Box Fish are some of them. These tools provide information at a network level by
displaying information specific to devices but lack in-depth knowledge about the underly-
ing MPI library used by jobs. As a result, none of these IB monitoring tools can help in
analyzing the interaction between the network and the MPI jobs running on them. On the
other hand, we have many tool suite, to profile an MPI application, like TAU, HPCToolkit,
Intel VTune, IPM, and mpiP give reasonable insights into the MPI communication behav-
ior. However, these tools do not have any knowledge about the underlying IB fabric and do
not correlate network level and MPI level behavior. With OSU INAM, we try to overcome
the limitations encountered with the above-mentioned tools.
Chapter 3: DESIGN OF INFINIBAND NETWORK ANALYSIS AND MONITORING TOOL

The proposed OSU InfiniBand Network Analysis and Monitoring Tool aims to monitor the network, profile the MPI jobs and to analyze their impact on various components. From an implementation perspective, the three main high-level functionalities of the tool are to accumulate the required information from various sources, persist the data and provide a visualization interface for the user. Figure 3.1 shows the architecture of the InfiniBand Network Analysis and Monitoring Software. The OSU INAM tool comprises of three major components: 1) OSU INAM Daemon, 2) OSU INAM Database, and 3) Java based Web Server Application and Web-based front end for client-side. The architecture is designed in a very modular manner which allows the user to install these components on the same machine or separate machine based on their requirements.

3.1 OSU INAM Daemon

The OSU INAM daemon is responsible for the collection of all the data required for network analysis and job monitoring from different sources. The daemon is capable of querying the underlying InfiniBand fabric to acquire the topology information. It also queries for the different performance counters on all the switches. For the MPI jobs, the daemon provides an interface for the jobs to send MPI process data to and consume these
Figure 3.1: Overall framework of proposed design
data packets. The daemon is also responsible for persistence of all the data in the database layer. Based on the data collected the software is able to present information at Network, Job and Process granularities. The several tasks performed by the daemon are designed to execute in parallel and thereby, avoid any bottleneck. For this purpose, the main daemon thread creates the following three separate threads, each handling specific tasks.

3.1.1 Fabric Discovery Thread

The Fabric Discovery (FD) thread is responsible for collecting the InfiniBand topology information and for gathering the various counters from the IB components. The FD thread first tries to obtain the Fabric object (ibnd_fabric_t) by calling the ibnd_discover_fabric method, which is also used in the “ibnetdiscover” utility. After obtaining the Fabric object which contains the IB topology information, the FD thread loops over the list of network objects i.e. the nodes and the links, and extract their information. Once all the devices in the network have been identified, the thread proceeds to retrieve the routes information between each pair of these devices using their lids. The nodes, links, and routes information form the fabric information which OSU INAM requires. The FD thread keeps performing this querying task to obtain the topology information at a user specified interval to capture any changes in the IB fabric. Once this data is passed to the database thread for persistence, the FD thread switches over to collect the performance counters.

Several approaches were considered for accumulating performance counters from the entire network. One of the options looked into was to use the OpenSM Plugin, for extracting the counter information. In this technique all the counter data are extracted for all the devices on the network. Many of these fields are already gathered by the MPI collection
thread at a deeper granularity. This leads to redundancy and also adds to the high priority management network traffic. To avoid this, the fabric thread queries a set of selected components in the cluster at a user specified interval. The fabric thread uses the method “pma_query_via” in libibmad same as the “perfquery” utility to query performance counters from all the ports of every switch in the network. After querying the data ports, the thread calls a method to reset these counters. The performance counters in the IB Switches are 32-bit counters, which run a risk of overflowing quickly based on the communication. Therefore, the administrator/user should try to set the interval for querying as low as possible to get the best result. After receiving performance counters data from all the switches, the fabric thread passes them as a linked list to the database thread for storing it.

3.1.2 Database Thread

The database thread is responsible for managing the database schema and receiving data from the fabric thread and the MPI Data collection thread to store in the database. The first time the database thread starts, it connects to the database and creates all the required tables in the database. The database name and the user should be pre-configured by the admin. Once all the tables are created, the database can insert the data that it receives from the Fabric Discovery thread (FD) and MPI data collection thread into these tables. The database thread also executes task at specified interval to purge the old data present in the table. The length of the duration for which the data should be present is configurable by the Admin. This helps in keeping the size of the database in check. If the daemon is being restarted or a newer version of the daemon is being started, this thread checks whether all the tables are present and update any table if required in the newer version.
3.1.3 MPI Data Collection Thread

Many of the network monitoring tools like Nagios, Ganglia, Mellanox Fabric IT, etc, as mentioned in Section 1.1 provide the information about various metrics like Xmit/Rcv counters at a Switch/Node level. However, they are not capable of capturing these values at a job or process level. Also, important information regarding the traffic such as how much of it is Point-To-Point, Collective communication, or RMA communication are not captured by these tools. OSU INAM tries to provide this information by trying to classify the traffic at a finer granularity. One of technique which was proposed was to retrieve per-process information. Each process can be assigned to a single virtual lane, the fabric can just query each virtual lane to obtain the process level information. However, the IB standard currently supports only 16 virtual lanes. Given the current trend of many core architecture, it is possible to have a node with more than 16 cores running more than 16 processes on it, hindering one to one mapping between the virtual lanes and processes. Along with this core fundamental implementation challenge, there are also other mundane issues such as: 1) Very few system administrator switch on the functionality of using virtual lanes in their cluster and 2) There are very few IB products which utilize virtual lane counters.

The second approach, which overcomes the limitations mentioned in the first one, is to let the MPI job itself send the various metrics pertaining to it, during the execution. For this, in the OSU INAM daemon, we have an additional thread, MPI Data Collection thread. This thread is responsible for receiving the job related metrics sent by the MPI jobs and pass it as a linked list to the database thread to save in the database. This thread acts as a listener waiting for packets from any MPI jobs running on the cluster. This way, it does not have to establish any connections or poll the MPI processes thereby avoiding a single point
bottleneck. The thread communication mechanism implements one of the IB transport protocols for high performance and low latency. In addition, the thread follows the interrupt mode rather than the polling mode, which reduces the CPU utilization significantly.

**Design Choices for IB Transport protocol**

As mentioned earlier, the MPI data collection thread uses IB for high performance and low latency communication with the MPI Jobs. As described in Section 2.1, several transport protocols based on IB are available such as Reliable Connected (RC), Unreliable Datagram (UD), Extended Reliable Connected (XRC), and Dynamic Connected (DC). Each of the transfer protocols has its own advantages and disadvantages. Previous studies [26, 33] have established that UD and DC perform better in terms of scalability and memory footprint. For OSU INAM, the main consideration is low latency and scalability. The MPI data collection thread should be able to service all the jobs running on the cluster without any negative impact on the resources. Reliability is not the major concern, as a few dropped packets are fine while trying to get the latest data. With this objective in mind, UD, which is typically used for low latency, high-performance small messages communication is preferred over DC [33].

**Co-designing the MPI runtime to work with OSU INAM**

As we saw in Section 2.5, the MPI\_T interface provides a convenient method to keep track of various internal states and metrics of an MPI library. We piggyback on this infrastructure and enhance it to enable monitoring for several more process level metrics. We introduce support in MVAPICH2-X to keep track of: 1) CPU utilization of each process including idle time, user time, system time and the rest; 2) memory utilization of each process including current and maximum size of virtual memory consumed; 3) inter-node and
intra-node communication buffers utilization including the maximum number of buffers that were required (high water mark); 4) intra-node bytes sent / received; 5) inter-node bytes sent / received; 6) total bytes sent / received for collective operations; and 7) total bytes sent for RMA operations. The MPI runtime collects this information and sends updates to the MPI data collection thread via UD Queue Pairs (QP) at user specified intervals (default value: 30 seconds). In addition to this, each packet sent has some meta-data information about the process itself like rank, LID and GUID from which its sending the data, the time stamp when data was sent, job ID etc., which will be used later to retrieve the data from the database. The MPI data collection thread dumps the UD QP and Local Identifier (LID) that it is listening on to a file. This location of this file is passed through environment variables set up by the system administrator to the MPI runtime. The runtime then uses this information while sending data out to the MPI data collection thread. While we chose MVAPICH2 for implementing our designs, any MPI runtime (eg: OpenMPI [16]) can be modified to perform similar data collection and transmission.

3.2 OSU INAM Database

The data storage is a very crucial component of the OSU INAM architecture. A scalable and well-planned database schema is required to guarantee smooth execution of the database queries and manage the high volume of the data inflow in a large scale high-performance environment.

Figure 3.2 shows the database schema for OSU INAM database by the application. Overall, it has nine tables which are able to store all the data, currently required by the application to provide the information required by the user for analyzing the network and
monitoring their the MPI jobs. The tables are organized to store the topology information, port and error counters for network analysis, and MPI job’s data.

In order to store the topology information, fields in the tables “nodes” and “links” are used, which helps to construct the network view of the web application for the underlying IB fabric. The table “route” is used to store information of all the routes between each pair of devices in the fabric. This data provides information to build the job level network view or to check which route is being used by a specific set of nodes etc. In order to store information necessary for network analysis, the tables “port_data_counters” and “port_error” are used to store the performance counters queried from the various fabric devices. For MPI Jobs and Process level information, the schema employs “process_info”, “process_comm_main” and “process_comm_grid” tables which store the MPI communication counters for point-to-point communication, collective communication, and RMA communication. This data facilitates the job and process level analysis along with communication pattern study. It also reports information about several MPI metrics like CPU Utilization, Buffer Usage, I/O activities, etc. for profiling and tuning the MPI jobs.

In this design, there is no referential integrity maintained between any of tables, as its validation and maintenance adds to the data insertion cost. As it is not critical for OSU INAM to have an accurate mapping between every records and since reordering of data and minor loss of data is acceptable, the design does not use any foreign key references.

3.3 Java Web Server Application and Web Based Front End Application

The Java Web Application deployed on the Web Server along with web based front end constitutes the third component of the OSU INAM tool. The purpose of the web interface is
Figure 3.2: Overview of OSU INAM database design
not only to provide a user-friendly interface to the user to work on but also to make the data accessible from any place with network connectivity and device with a browser. Using this web interface, the system administrators, MPI developers, and the users can view all the statistics related to network and MPI jobs parameters. The web application provides several features and functionalities to explore the available data at different granularities. It allows the system administrator to have a high-level network view to quickly look for any point of failure or bottleneck and also lets the developers and users view detailed information of a process for an individual job for profiling and tuning purposes.

While developing the web application along with the user interface, it becomes imperative to have a scalable and modular design. For implementing the web application, the Spring Web model-view-controller (MVC) framework has been considered. For accessing and querying the database resource, the Spring JDBC (Java Database Connectivity) framework is used. The application has two separate data sources, one for the OSU INAM database and other for the resource manager, SLURM, which provides information about
the jobs running on the cluster, etc. This modular design provides the flexibility to provide support for more data sources and other resource managers in the future. The Web application is deployed on a Tomcat server.

On the client side, we use a simple light-weight JQuery based interface for the user to interact and the majority of the calls are AJAX based calls which allow automatic asynchronous updates of information on the client side, providing an uninterrupted seamless current view of the network and all the jobs running on it. The pages are designed to be rendered using least CPU utilization and in less time, to improve the user experience.

The web application adopts the Singleton Pattern for maintaining the network object representing the HPC cluster. The Singleton pattern ensures that the network class has only one instance and provides a global point of access to that instance. This object is updated to represent the latest status of the cluster by querying the database. It holds information regarding the topology, network traffic, current jobs running etc. The benefit of this approach is that to service user’s requests regarding network view can be serviced by this singleton network instance and the web application would not have to hit the database every time, helping to reduce the database load. This singleton instance representing the cluster is generated during the deployment process. The web application is deployed after the daemon has been started and the database has been setup.

The overall processing flow is as follows: 1) Whenever a users click generates an HTTP request, it will be sent to the server side by the web browser or JQuery library with AJAX; 2) Once the Tomcat server receive the request, it is passed to the Spring framework who will dispatch the coming request to the corresponding controller based on the mapping information of URL (in the request) and Controller, which has been configured initially. The dispatcher has information about which controller needs to be invoked; 3) The selected
controller will be invoked and it can query the model for some information, in most cases, about some data in the database; 4) Once processing has been done, the Spring framework will get the response to build the view through JSP, XML etc; 5) Finally, the HTTP response will be sent back to the browser at the client side. Then the Web page will be updated. Note that the whole process is completed very fast since all the data has been stored in the database through the OSU INAM daemon in advance and all the processing steps are configured and indexed in the database.

3.4 Design Enhancements for Scalability For Large Scale HPC Systems

Any tool that aims to visualize and profile large scale HPC systems must address some fundamental challenges: 1) it must be portable, easy to use and understand; 2) should support visualizations of large scale systems and provide scalable storage techniques; and 3) be applicable to the different network interconnects and high-performance MPI libraries that are likely to be used on HPC systems. This section highlights the designs proposed to address these challenges. Figure 3.1 includes the architectural changes due to the proposed design. It consists of four major design components: 1) the Network and MPI load generator, 2) Optimized Network Rendering using PhantomJS and Clustering technique, 3) Optimized Data Storage Techniques, and 4) Network Abstraction Interface to handle multiple networks.

3.4.1 Network and MPI Load Generator

Before deploying the software on a cluster, the user or the system administrator may want to test the software in a contained environment. The software should be able to emulate the behavior of a real cluster and be able to generate the data, both the network
counters and the MPI job counters. This will allow the user to test whether the deployed tool is able to handle the load of the cluster and evaluate the potential impacts. The emulation can be used to stress test the daemon and can be used to tune the database for the expected inflow of data.

In order to achieve this objective, the load generator module was designed and implemented to provide the user the option of running the OSU INAM software in the emulation mode. The installation procedure for this remains the same as for the standard use. While starting the daemon, the user has to pass the emulation flag. The daemon also requires the topology information of the cluster for which it would be generating the load. The cluster’s topology is passed to the daemon as a file which is used by the daemon to extract the node and links information. The topology information can be obtained by running a utility provided along with the software. This utility uses the ibnd_discover_fabric method to get the fabric information of the cluster and write it to a file. This file serves as input to the daemon, where it is loaded by using the method ibnd_load_fabric.

When the daemon is started in the emulation mode, it has to generate the data for network performance counters as well as the MPI job counters. For the network performance counters, the fabric thread would not query the actual ports but instead generate the simulated counters. It also keeps track of the amount of data it is generating which helps to evaluate the incoming traffic because of the performance counters. For the MPI job counters, a separate simulation thread is created, when the daemon is started in the emulation mode to distinguish the MPI part from the daemon. This thread is responsible for simulating MPI jobs creation, sending the MPI job counters to the daemon and stopping a job.

The following properties from the configuration file are used by this thread for the MPI
data generation: 1) Number of processes ran per node, 2) Maximum and minimum number of nodes allocated to a job, and 3) Maximum and Minimum time duration for a job. This thread keeps track of the active jobs that are running and the number of nodes that are available. It creates a job, adds it to the active jobs list, allocates the job a certain number of nodes and assigns a time duration for which the job is supposed to run, based on the parameter mentioned in the configuration. This information is saved in the separate table in the database. As long the job is active, the thread sends the simulated MPI job data. This data is sent via the InfiniBand HCA using the loopback connection. When a job’s duration expires, the job is removed from the list and the nodes are freed.

The web application also needs to be deployed with the simulation flag on in the properties file. The user can view all the network and job functionalities simulated by the daemon. When deployed in the simulation mode, the web application uses the simulated job data from the database and not the resource manager database.

Along with testing all the functionalities provided by the OSU INAM software, the simulator can also be used for load testing. Following features can be tested by the user: 1) whether the network thread is able to consume the MPI data packets generated by the simulator thread and not causing any bottleneck; 2) the database thread is able to receive the data from the fabric discovery and MPI data collection thread, and able to persist in the database; 3) the web application is able to render the network view.

3.4.2 Optimized Network Rendering using PhantomJS and Clustering technique

For the network view, the calculation of the position for all the nodes of the network has a weight associated with it. The web browser, for a small network having around 100-200 nodes, can take few seconds to calculate the positions using the selected physics option and
render the entire network. However, for a mid to large scale network, the time required for rendering increases significantly. For clusters beyond a size of 500 nodes, the time needed for rendering can go up in minutes, sometimes even hours. This not only results in a very poor user experience but also increases the client CPU utilization significantly.

**Designing Optimized Network Rendering using PhantomJS**

The calculation of the positions for the nodes is moved from the web browser on the client to the server side. In order to achieve this objective, PhantomJS is employed. In the new design, PhantomJS is placed along with the web server, as an extension to the visualization component of the OSU INAM software. When the OSU INAM web application is being deployed, during the context initialization, the application invokes a PhantomJS process which calculates the positions of the network nodes, like a web client would, and returns the result to the application. The web application sends this information along with network data to the web client, thereby, eliminating the need for the web client to perform the above task and can directly render the network.

Following are the steps executed during the web application deployment:

1) The web application reads the nodes and links data from the OSU INAM database and constructs the network object.

2) The web application invokes the PhantomJS process and passes the network object to the PhantomJS module.

3) The PhantomJS process calculates the positions of all the nodes based on the configured physics option of VisJS.

4) Once the calculation is complete, the PhantomJS process passes the new information back to the web application and terminates.
5) The web application caches the position data and sends it to the web client along the network whenever there is a request.

The PhantomJS component in the web deployment adds to deployment time. However, this is a one-time cost. In order to avoid the PhantomJS process from being invoked unnecessarily every time the web application is deployed or the server is restarted, the web application writes the network position information on the file system. In case the file is available, the deployment procedure would skip the PhantomJS call and read the positions directly from the file. The web application is also capable of detecting any changes to the network and whether the existing position information in the file requires any changes. If any new nodes are added or if any nodes are removed, the web application would call the PhantomJS to calculate the position taking account the new changes and update the existing file.

**Improving Load Time with Clustering**

Implementing the PhantomJS module significantly improves the network load time on the client side. However, the time required for rendering is directly proportional to the size of cluster i.e the number of nodes and links. For small to mid-size cluster this is not an issue, however for a large scale HPC cluster, there can be some delay introduced because of this. Also, rendering the entire network with all the nodes and links can give a very crowded visualization of the cluster, making the devices cluttered and hard for the user to identify any potential issue or use features provided by the “network view” page.

In order to address these concerns, the option for clustering the network elements was explored. As per the clustering technique, when the network is loaded only the switches and links connecting the switches will be drawn on the Canvas. All the compute nodes connected to a particular switch are represented by a single node which acts as a “blob”.

28
This not only significantly reduces the number of nodes and links that needs to drawn on load thereby, reducing the rendering time, but also, present a much clearer and easy to use view of the network. In order to view the nodes, the user can expand these “blobs” by double clicking them. An expanded set of nodes can be again collapsed back to “blob” connected to the switch. The configuration file of the web application has a parameter to decide the size of the cluster for which the clustering mechanism should be activated. The user can set the value of parameter “osuinam.clustering_threshold” as required. For cluster larger than this size, the leaf nodes will be collapsed by default to improved the rendering time.

Figure 3.4 and Figure 3.5 show the depiction of the full network views of Stampede @ TACC, Comet @ SDSC, Gordon @ SDSC and various HPC at the Ohio Supercomputing Center (OSC) by the proposed tool. Note that the number of nodes and links in parenthesis represents the actual number of nodes that were active when the results were taken and can thus vary slightly from the advertised numbers on the websites of these systems.

3.4.3 Designs to Enhance Data Storage Performance

In order to improve the performance of database operations, several design enhancements for the daemon and database schema were explored. In addition, the database specific optimizations were also considered.

Daemon and Database Changes to Enhance Data Storage Performance

Current implementation the OSU INAM tool supports the MySQL database which is one of the widely used open source SQL-based databases for data persistence. While trying to emulate a large scale size network, like Stampede, several issues were observed with data persistence. Owing to the huge amount of data generated by the load generator, the data
Figure 3.4: Full network view of Stampede@TACC and Comet@SDSC systems as depicted by OSU INAM
Figure 3.5: Full network view of Gordon@SDSC and OSC Clusters as depicted by OSU INAM
storage functionality of the daemon was becoming a major concern, overwhelmed by the data it had to manage. Upon further analyzing this issue, it became evident that the insertion into the database was the bottleneck. The rate of inflowing data was more than the rate at which the database thread was able to consume and insert this data into the database. The data that is being collected by the daemon and stored in the database can be classified into three types, namely: 1) Network fabric and performance counters data, 2) MPI process level information, and 3) MPI process to Node communication grid. In the original design, all the inserts for this three types of data were being serviced by a single database thread with one connection. This was causing the single thread to be overwhelmed by the amount resulting in data getting backed up.

From the database schema 3.2, we can see that there is no dependency between the tables storing these three types of data. This allows the possibility of inserting the different sets of data into the tables in parallel and reduce contention for connection, by adding extra thread. Lack of any foreign dependencies between these tables helps in ensuring the decoupling. As the tables are completely independent and without any references between them, no row/table level locking is caused which could hinder implementing a new thread. With this design, the fabric and port counters can be inserted by one database thread and the MPI related data i.e the MPI process information and the communication grid can be inserted by the second database thread. This not only makes the insertion process faster due the additional parallelism introduced, but it also makes sure that any delay due either fabric or MPI information insertion does not impact the other one.

While the addition of an extra database thread improved the performance to a certain extent by reducing contention and bringing more parallelism, it was observed that the MPI data is still causing an issue with regard to its insertion rate. On further investigation, it
was noticed that it was the insertion for MPI process to node communication grid data which is taking more time than expected and causing delay. This issue arises due to the existing database schema design, where “process_comm_main” and “process_comm_grid” tables are used for storing the grid information. While the ‘process_comm_main’ stored high-level information about the process of a job in a single row, the “process_comm_grid” table store the communication grid information where single row store information on one process to node mapping. This leads to a multiplicative effect on the number of records inserted into “process_comm_grid” depending upon the number of processes. This design was to support read intensive tasks.

On considering the use cases which require the communication grid, we saw that this data is mainly used for: 1) Displaying the Process-to-Node communication grid at Job and Node level for understanding communication patterns and 2) Link Utilization at Job and Process level. However, these operations are generally not performed heavily and may require a subset of the data present in the table. With this in mind, the possibility of changing the schema design to make it support write-intensive operation for MPI communication grid was explored. MySQL provides a datatype “TEXT” which allows storing a string of size 2000 characters. Rather than storing the grid in the process_comm_grid table, a new column of type “TEXT” is used in process_comm_main for storing the grid information in a text format. This eliminates the need to insert the grid array with process-to-node mapping in process_comm_grid table, thereby removing the multiplicative effect. This change in table design improves the inserts time significantly. With the new change, the web application has to process the text column to extract the grid value.
Optimizing MySQL for Enhanced Performance

As the cluster size increases, the database size also keeps growing. Thus, it becomes necessary to optimize the various database parameters in order to get the best performance for read and write operations. To this end, one first needs to identify the most critical parameters that affect the performance of database operations and choose an optimal value for it that delivers peak performance. One of the first optimizations was to change the MySQL storage engine from MyISAM to InnoDB. Unlike MyISAM which gives table level locking write mechanism, InnoDB provides row level locking write operations, thereby allowing faster concurrent read and write operations on the same table. This change enhanced the performance when read (by the UI) and write (by the daemon) operations were happening concurrently from the database. The second major change was to use the technique of inserting records in a “bulk” fashion instead of individually to the database. Each database insertion operation has a constant overhead associated with it for performing various “meta” operations in the database (e.g., obtaining locks, updating indices, etc.). By performing bulk inserts, one can reduce the impact of the meta operations by number of inserts for a given set of operations.

One of the major differences between MySQL and other in-memory databases and memory backed key-value stores is that MySQL writes everything to disk immediately assuring full persistence. On the other hand, the schemes where data is staged in memory perform a delayed flush tactic and perform flushes in the background thereby allowing for better performance with insert operations. MySQL InnoDB also has a specific tuning parameter `innodb_flush_log_at_trx_commit` which when set to a value of 2 will amortize the insert operations happening over one second and flush it to disk after expiry of that time interval. Almost 10X improvements in the speed of insert operations after setting
this parameter. The con of this approach, however, is that, in the case of an operating system failure or power failure, there is a potential that, at most, one second of data will be lost. Note that the failure of a MySQL daemon process will not result in any loss of data [6]. Such a situation can easily be mitigated using redundant, backup MySQL processes running on a separate node. Thus, a design that favors performance and chooses to persist the last seconds worth of data in memory instead of persisting it immediately to disk is chosen.

In the same vein, there are certain other parameters that will have an impact on performance when staging some amount of data in memory. These values are further optimized for the MySQL InnoDB engine to achieve the best performance for large clusters. Table 3.1 summarizes the database parameters selected for optimization and the final optimized values used for experiments in. Note that these parameters were chosen after careful analysis of the many MySQL related optimization parameters and selected due to their potential to impact insertion and query performance.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default Value</th>
<th>Optimized Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>innodb_buffer_pool_size</td>
<td>Size of memory area where InnoDB caches table and index data</td>
<td>8 MB</td>
<td>4,096 MB</td>
</tr>
<tr>
<td>innodb_log_buffer_size</td>
<td>Size of memory area which InnoDB uses to write to the log files on disk</td>
<td>1 MB</td>
<td>1 MB</td>
</tr>
<tr>
<td>innodb_log_file_size</td>
<td>Size of each log file in a log group</td>
<td>1 MB</td>
<td>256 MB</td>
</tr>
<tr>
<td>innodb_flush_log_at_trx_commit</td>
<td>Controls the balance between strict ACID compliance for commit</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Batch size</td>
<td>Number of records inserted with one operation</td>
<td>1 records</td>
<td>100 records</td>
</tr>
</tbody>
</table>

### 3.5 Re-designing the Data Collection Daemon to Handle Multiple Networks

Initial implementation of the OSU INAM daemon was based on the Mellanox OpenFabrics Enterprise Distribution (OFED) software for the InfiniBand networks. It makes use
of the utilities provided by the OFED library to extract the fabric and counters information. However, in order to provide flexibility and portability to support other architectures as well, a general set of abstractions and interfaces have been proposed. Their main objective is to provide an interface to provide network-based functions which fetch the fabric data. The architecture specific information like the fabric’s topology information, route information, and the counters data are all retrieved by daemon’s fabric thread by querying the underlying fabric. For this purpose, we have the below functions declared with their implementations being underlying fabric specific.

```c
int get_fabric_data(app_data_t *ad);
int get_route_data(app_data_t *ad);
int get_perf_counters(app_data_t *ad);

typedef struct app_data {
    struct inam_node *node_head;
    struct inam_link *link_head;
    struct inam_route *route_head;
    struct perf_counter *perf_count_head;
    ...
} ...
```

Figure 3.6: Proposed network abstraction API

The Omni-Path Host Fabric Interface provides several primitives for querying and extracting information from the OPA fabric. The open source OpenFabrics software stack allows the user to obtain the meta information of the network devices like the topology information and performance counters of the ports. The opareport utility present in the OFA FastFabric package provides with the topology information. The daemon uses functions similar to the opareport to query the underlying network using the “Sweep” method to obtain the fabric object “FabricData_t”. The fabric thread, then, parses this object to
extract the nodes (STL_NODE_SW and STL_NODE_FI) information from the Fabric object. It then proceeds to extract the routes information between all the compute nodes (STL_NODE_FI).

For the performance counters, the daemon iterates over every port of each switch to read the performance counters. The performance counters are 64-bit counters which allow them to hold more data before overflowing. This allows the user to reduce the query frequency compared to an MOFED InfiniBand implementation.

Both the topology and port counters information are mapped to the high-level data structures defined. Based on the flag passed during compilation, the daemon can be build to support either the MOFED InfiniBand implementation or the Omni-Path Architecture.
In this Chapter, we first go over various features provided by the OSU INAM tool. We then analyze the impact of running MPI jobs along with OSU INAM on the job’s performance and evaluate the results of changes implemented for scalability.

4.1 Discussion on Features of OSU INAM and its Impact

The features provided by the OSU INAM tool, on a high level, can be categorized as network level features for monitoring network metrics and as job level monitoring features for profiling and analyzing the MPI jobs.

4.1.1 Network Level Monitoring

The network page, as shown in Figure 4.1 renders the network topology allowing the user to have a high-level view of the cluster. It displays all the nodes [HCAs and switches] and the links, provides basic information about the devices [GUIDs, LIDs], and the MPI jobs running on the nodes. The user can view the current traffic flowing through the cluster based on the link usage displayed on the cluster. This view keeps updating asynchronously at a user specified interval providing a real-time view of the cluster activities. The user can view either the entire network or has the option of viewing only certain nodes selected.
by the user or a set of nodes allocated to the jobs entered by the user. Along with the link usage which is derived from the Xmit/Rcv data, the network page allows the user to view different network metrics like Collective data, RMA, Point to Point flowing through the links. It also provides the error information helping the admin to identify any link or node failure from the network view. By default, the network view would display live data. The page also offers the capability to view historical information. The user can select the historical view option, enter a start and an end time, and view the network activities that transpired during that duration.

The network page provides an interactive interface for the user to select nodes and links to view additional information. For example, the user can view routes passing through a selected link, all the available routes between a set of selected nodes, etc. The user can also view detailed information about a particular switch or a node by double clicking on the nodes. For the switch, the OSU INAM tool provides information about all the active ports being used and the live values of all the data and error counters. For the nodes, the
tool displays information about the job using that node and detailed information about all the processes running on it.

### 4.1.2 Job Level Monitoring

The OSU INAM tool provides features to the MPI developers and users to monitor MPI jobs on a real time basis and to profile their applications at different granularities. Based on the MPI data and network metrics collected by the daemon and the information from the resource manager, the tool is able to provide a holistic picture of how the jobs are running, their resource utilization, the communication pattern and how they might be getting impacted by other network activities. Just like for the network components, the tool also has an option to view historical information for a job. The user can enter job ID of a completed MPI job and can analyze the metrics during its execution. Below are certain specific features which help the users in profiling their application and monitoring any discrepancies.

**Analyzing and Understanding Inter-node Communication Buffer Allocation and Use**

Several implementations of MPI Runtime would have a set of buffers reserved to facilitate the data communication occurring during the application execution. This set of buffers are preregistered during the job initialization with the IB HCA for small message transfer. MVAPICH2 uses this mechanism extensively and even provides facilities to tune this parameter for a specific application. It is very critical to tune and manage these buffers as they are pinned down to the memory and cannot be swapped during the execution of the application, thereby making them unavailable for the MPI job to use it. Hence, it becomes pertinent that the user is able to tune these parameters as required. OSU INAM is able to provide the information about this buffer usage which can help the user to tune the buffer
Table 4.1: Comparison of communication buffer utilization for default and tuned scenarios for 512-process class D NAS parallel benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Default-HWM (Max Value)</th>
<th>Default-Alloc (Max Value)</th>
<th>Default-Communication Buffer-Memory (Sum) (MB)</th>
<th>Tuned-HWM (Max Value)</th>
<th>Tuned-Alloc (Max Value)</th>
<th>Tuned-Communication Buffer-Memory (Sum) (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CG</td>
<td>1</td>
<td>240</td>
<td>1570.20</td>
<td>2</td>
<td>48</td>
<td>409.33</td>
</tr>
<tr>
<td>EP</td>
<td>1</td>
<td>240</td>
<td>1570.20</td>
<td>3</td>
<td>48</td>
<td>348.22</td>
</tr>
<tr>
<td>FT</td>
<td>356</td>
<td>544</td>
<td>1735.49</td>
<td>295</td>
<td>320</td>
<td>647.24</td>
</tr>
<tr>
<td>LU</td>
<td>164</td>
<td>352</td>
<td>1584.74</td>
<td>152</td>
<td>192</td>
<td>503.76</td>
</tr>
<tr>
<td>MG</td>
<td>30</td>
<td>240</td>
<td>1570.20</td>
<td>32</td>
<td>80</td>
<td>561.33</td>
</tr>
</tbody>
</table>

parameter. OSU INAM displays the number of buffers allocated by MVAPICH2 and the number of buffers used by the application. Table 4.1 shows the results on how the internal buffers were used for communication while running the NAS benchmarks. After the job completion, we can check for the high water marker for the internal buffer usage over the job’s execution time.

Table 4.1 highlights the number of internal inter-node communication buffers taken for a 512 process run of class D NAS parallel benchmarks. The column “Default-Alloc” highlights the number of communication buffers pre-allocated with the default communication buffer tuning done for MVAPICH2. The “Default-HWM” column highlights the maximum number of communication buffers actually used by the application kernel in the default scenario. As we can see, there is a significant waste of communication buffers for several application kernels. With this insight, we perform application specific tuning and reduce the number of internode communication buffers pre-allocated at initialization time. “Tuned-Alloc” indicates the number of buffers allocated after we tuned the number of communication buffers with the insights gained from OSU INAM. As we can see by comparing the memory taken for the default and tuned, we are able to save significant amounts of memory without any impact on the communication performance. Another observation is that the “Tuned-HWM” value is higher than “Default-HWM” in several cases even when
the “Tuned- Alloc” is much less than “Default-Alloc” indicating better utilization of available communication buffer resources.

**Identifying and Analyzing Sources of Link Congestion**

Using the network metrics obtained from querying the fabric, it is possible to locate any link congestion happening in the cluster. Many network-based utilities are able to provide this feature. However, many times just locating the point of congestion is not enough, the user may also want to know what is the source of the congestion. In a network with dynamic routing, this can be very challenging. However in an IB network managed using OpenSM, the routes between the nodes are predefined. This knowledge about routes, which is present in the OSU INAM database, in conjecture with the information from resource manager about the nodes allocation and the communication metrics from the MPI job data allows OSU INAM to provide a picture of a link’s utilization by the jobs passing data through it. The Links Information can display data at the job level, showing which job is using what percentage of the bandwidth, whether one job’s communication is having an impact on any other jobs using the same links. It also able to display the percentage of bandwidth utilized within a job at process level as shown in Figure 4.2.

**Monitoring Jobs Based on Various Metrics**

For a user, it can be important to know the behavior of the MPI jobs which generally can be interpreted from the various metric related to CPU utilization, Memory usage, I/O activities and communication characteristics. A general job scheduler would have information about the nodes in the network allocated to a job along with certain other information like its start and end time etc. However, it does not list the various metrics which an user would need to profile their application that might have an impact on the HPC environment.
For example, if a job is performing heavy I/O action due to a check-pointing operation or it has encountered a segmentation fault and is in the process of dumping its cores, causing some negative impact on the system’s performance. Similarly, if there is heavy communication task being carried out by the job like an “AlltoAll” that could possibly have some effect. In order to address this and to allow the user to focus on these “high value” jobs, OSU INAM displays a list of live jobs in the HPC cluster along with the important metrics in a tabular format in the “Live Jobs” page. This table shows real-time data based on the information saved in the “process_info” table. The user can sort this list of live jobs based on any of this metric as per the user’s requirement. If the user is interested in viewing the jobs contributing most to the I/O load, he can just sort it based on the I/O read or write metrics. The job id displayed in the first column are hyperlinks which will open individual jobs page to provide detailed information about the jobs.
Capability to Profile and Report Several Metrics of MPI Processes at Different Granularities

One of the risks of providing a lot of information directly to the user is that it may get overwhelming for the user to study the data and he may miss out on any existing issue or bottleneck. To improve user experience and to get the best out of the data, OSU INAM strives to display the collected data in a granular manner. For the MPI, though all the data collected by the data is at a process level, the web application is able to aggregate and categorize this data at much coarser granularities. The web interface has been designed to display high-level view of the job [Figure 4.4], which is the network view along with job level details. The user can then dig his way to view detailed information, at node level [Figure 4.5] followed by process level.

Many times, it is after the execution, that an admin/user may want to understand the behavior of a job or the impact a job it may have had. OSU INAM is able to provide such “post-mortem” analysis as it stores all the data for later retrieval, which many current IB fabric tools lack. If an administrator has access to a tool like OSU INAM which has the ability to “play back” events that occurred at a time in the past, it allows them the flexibility to inspect events at a later point in time and identify the culprit(s) that caused the issue.
Figure 4.4: Job level view

Figure 4.5: Node level view
4.2 Experimental Results

In Section 4.2.1, we have the experimental setup used for running the all our experiments. In Section 4.2.2, we see the impact on the MPI applications when they are used along with OSU INAM. The results after the scalability enhancements are presented in Section 4.2.3.

4.2.1 Experimental Setup

Each node in the experimental cluster equipped is with Intel Westmere series of processors using Xeon dual socket, quad-core processors operating at 2.67 GHz with 12 GB RAM, MT26428 QDR ConnectX-2 HCAs (32 Gbps data rate) with PCI-Ex Gen2 interfaces. The operating system used is Red Hat Enterprise Linux Server release 6.7 (Santiago), with kernel version 2.6.32-431.el6 and Mellanox OFED version 2.2-1.0.1. We have used MySQL v5.1.73 and MemSQL v5.0.3 for the evaluations. The version of PhantomJS used was 2.0.0. Table 4.2 highlights the salient points of the different HPC systems the authors are using as testbeds for their tool.

<table>
<thead>
<tr>
<th>Cluster</th>
<th>Number of Nodes</th>
<th>Number of Switches</th>
<th>Number of Ports</th>
<th>Number of Links</th>
<th>Network Topology</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gordon</td>
<td>1,081</td>
<td>64</td>
<td>2,304</td>
<td>1,657</td>
<td>3D Torus</td>
</tr>
<tr>
<td>OSC</td>
<td>1,400</td>
<td>184</td>
<td>5,628</td>
<td>3,027</td>
<td>Composite Fat-Tree</td>
</tr>
<tr>
<td>Comet</td>
<td>1,879</td>
<td>212</td>
<td>7,636</td>
<td>4,377</td>
<td>Hybrid Fat-Tree</td>
</tr>
<tr>
<td>Stampede</td>
<td>6,541</td>
<td>764</td>
<td>27,504</td>
<td>16,893</td>
<td>Partial Fat-Tree</td>
</tr>
</tbody>
</table>
4.2.2 Impact of Co-design of MPI Runtime with OSU INAM on the MPI application

In this section, we study the impact of co-design of MPI runtime with the MPI data collection thread of the OSU INAM daemon, the overhead involved in sending data for the daemon’s thread. The experiments are carried on basic communication pattern such as point-to-point and collective operation, and popular application kernel like the NAS Parallel Benchmarks.

Impact of Profiling on Performance of Basic Microbenchmarks and NAS Parallel Benchmarks

![Figure 4.6: Microbenchmark-level point-to-point (Latency) performance](image)

The below Figure 4.6 shows the comparison between the basic point-to-point internode latency with data collection on for the daemon and without it. We observe that the degradation with data collection on is less than 1% when compared against native execution. Figure 4.7 plots results obtained from running the osu_mbw_mr microbenchmarks using a pair of nodes with MPI data collection running and without it. From the result, we can see that the collective and transmission adds to around 6% to 8% overhead for messages less
Figure 4.7: Microbenchmark-level point-to-point (Message Rate) performance
than 4,096 bytes. For larger message, there is hardly any impact with the overhead less
than 1%.

Figure 4.8: Microbenchmark-level collective performance at 512 processes (Performance
of Broadcast)

Figure 4.9 and Figure 4.8 show results for benchmarks for collective communication
patterns to study the overhead due to data collection and transmission. We ran osu_broadcast
and osu_alltoall with MPI data collection and without it at a scale of 512 processes. For the
AlltoAll benchmark, we see a degradation of 5% for messages less than 1,024 bytes. For
Figure 4.9: Microbenchmark-level collective performance at 512 processes (Performance of Alltoall)

Figure 4.10: Performance of class D NAS parallel benchmarks at 512 processes

larger, the degradation is around 7%, with only 4,096 showing around 12%. For the broadcast benchmarks, the tool adds to around 5% overhead compared to the version without the tool.

Figure 4.10 depicts results for application with MPI runtime with OSU INAM co-design and with it. The graph shows outcome for the different kernels/applications of the NAS Parallel Benchmarks. We can observe that support for data collection and transmission for OSU INAM causes negligible impact of the performance of these applications. These are
encouraging trends which positively advocate the use of such tools for end applications on modern supercomputing systems.

4.2.3 Impact on OSU INAM post Scalability Enhancement Design

The web client UI load time is heavily dependent on the machine’s CPU power and memory on which rendering in performed. In the case of MySQL or other data storage, the performance will be dependent on the underlying storage system used (such as SSDs, HDDs), CPU processing speed, and the operating system version. Trying to evaluate and compare results between different setups would be too exhaustive and it would be unfair to compare to results between configurations. Thus, in the interest of evaluating accurately and fairly, all the experiments were performed on the setup describe in Section 4.2.1. The network and load generator have been to used to simulate large clusters and evaluate results for them. In addition, this tool has been deployed on the HPC systems at OSC and on Comet at SDSC.

Impact of PhantomJS on UI Load Time

In this section, we are going to study the impact of implementing PhantomJS in the web application layer on the rendering time for a network. As mentioned in Section 3.4.2, PhantomJS pre computes the positions required for network devices to be drawn. We will also analyze the affect of clustering on the rendering time. As the web server and the web client are on two different machine, the data transfer also needs to be accounted for. In the experiment setup, the two different systems were connected via regular 100Mbps Ethernet connection. Figure 4.11 shows the results for the four different HPC clusters. In this experiment, results have been evaluated for three modes: A) With both PhantomJS and
Clustering, B) With PhantomJS and without Clustering, and C) With Only Clustering and Without PhantomJS

Figure 4.11: Performance breakup of UI load time

Figure 4.11 shows both the data transfer time and the rendering time. Majority of the time is involved in the rendering and it is dependent on the size of the cluster. Gordon, smallest among the four, takes the least, while, Stampede takes the most. For Stampede, without the PhantomJS based support, the tool takes almost 1,250 seconds to render the entire network. However, with PhantomJS based design the time of rendering is reduced by a factor of 5.5X to 225 seconds. If we add the clustering enhancement on top of PhantomJS, the rendering time can be further brought down by a factor of 8X to 28 seconds. Thus, cumulatively, the “PhantomJS + Clustering” design is able to bring down the cost of rendering by a factor of 44X.
Impact of MySQL Optimizations

As described in Section 3.4.3, optimizing the values of critical parameters in MySQL have a significant impact on performance delivered. Figure 4.12 highlights the results of the study done to optimize the various MySQL parameters. 55,008 Port Counters records being inserted once every 30 seconds, 104,656 MPI Process Info and 104,656 MPI Comm Grid Info records being inserted once every 45 seconds (over quarter million records) were used for all experiments depicted in Figure 4.12. These numbers are similar to what a cluster of Stampede’s size running at nearly full load will generate. Note that for each run measuring the impact of a parameter on performance, innodb.flush_log_at_trx_commit was set to “2” and all other parameters (except the one being measured) were set to the default values indicated in Table 3.1.

As can be seen, batch size has the most impact on performance. It is observed that, by increasing the number of records inserted in one go, one is able to significantly enhance the performance. For instance, using a value of 100 for batch size reduce the total cost of inserts from 78,398.61 ms to 55,346.39 ms — a 29% improvement. Although, increasing the batch size to 200 leads to 33% improvement, it is not chosen as it would double the memory requirement at the daemon end for relatively small benefits. It is observed that, with the default values of buffer pool size and log file size as being used by InnoDB, the database was not able to handle the rate of inserts described above and was leading to terrible performance. Thus, these are not shown here. The search for an optimized value is instead started from a larger base value for each of these parameters (512 MB for buffer pool size and 256 MB for log file size). For buffer pool size, a value of 4,096 MB (4 GB) delivers the best performance. For log file size on the other hand, although a value of 2,048 MB (2 GB) delivers the best performance, the 256 MB is chosen due to two reasons —
1) the difference in performance obtained at 256 MB and 2,048 MB is only 2% and 2) it would significantly reduce the memory footprint of the MySQL daemon process. Finally, for the log buffer size parameter, it is found that the default value was already providing the best trade-off between performance and memory footprint. Thus the default value itself is selected for this parameter in subsequent testing.

Figure 4.12: Impact of MySQL optimization on performance of insert operations
Chapter 5: CONCLUSIONS AND FUTURE WORK

5.1 Summary of Contributions

In this thesis, we presented the design of OSU INAM - a low-overhead profiling and visualization tool that is capable of presenting the profiling information obtained from the network and the MPI library in conjunction.

Following sections provide a more detailed summary of the research contributions:

5.1.1 Design and Features of OSU INAM

In Chapter 3, we described the basic modular design of OSU INAM along with the detailed description of each component. In Chapter 4, we demonstrated how, through the profiling information provided by OSU INAM, developers as well as users of high-performance middleware can gain more insights into the communication characteristics of their runtimes allowing them to further fine tune the performance on a per application or per run basis. We showed how, through the link analysis capabilities of OSU INAM, system administrators can pin point the cause of network performance issues to a granularity of a process.
5.1.2 Enhancing OSU INAM for Large Scale Clusters

The design of a high performance and scalable network-based performance analysis tool for MPI that is able to visualize and profile networks of large scale supercomputing systems with high performance and scalability was presented in Chapter 3. A modular redesign of the data collection daemon was done to support analysis and monitoring of Omni-Path and InfiniBand networks. A simple load generator that allowed to stress test various components of the proposed tool was also developed. The experimental results in Chapter 4 demonstrated that, while loading up the user interface depicting a cluster with 6,541 nodes, 764 switches and, 16,893 network links, the PhantomJS based design that is capable of pre-rendering and caching a network, was able to give a speed up of 44X over the over non pre-rendered/caching enabled solution. The proposed lock-free designs, new database schema, and optimizations enabled the tool to handle the data generated by a) querying 27,504 switch ports at a frequency of once every 30 seconds and b) 104,656 MPI processes at a frequency of once every 45 seconds — over quarter million inserts into the database every 45 seconds.

5.2 Software Release and its Impact

Several features of OSU INAM presented in this thesis are already publically available in the released versions of the OSU INAM package which can be downloaded for free from http://mvapich.cse.ohio-state.edu/tools/osu-inam/. We plan to release the remaining features in upcoming releases of OSU INAM. While the MPI data collection was designed and implemented using MVAPICH2-X, note that the same techniques are equally applicable to other MPI stacks. As of July ’16, OSU INAM has been downloaded more than 360 times directly from the OSU site.
5.3 Future Work

We plan to add support to profile and analyze the energy consumed by a MPI application. We also would like to add the capability to profile various PGAS programming languages such as OpenSHMEM [30], UPC [36] and CAF [15] as well as different Big-Data frameworks like Apache Hadoop [20], MapReduce [24] and Spark [31]. In addition, we plan to investigate the benefits of using RDMA technology to enhance the performance of memcached based data staging and to explore the use of high performance interconnects and protocols to enhance the data transfer time between the Java webserver and front end UI.
Bibliography


58


