INAM^2: InfiniBand Network Analysis and Monitoring with MPI

H. Subramoni, A. M. Augustine, M. Arnold, J. Perkins, X. Lu, K. Hamidouche, and D. K. Panda

Department of Computer Science and Engineering, The Ohio State University, Columbus, OH

{subramoni.1, augustine.80, arnold.668, perkins.173, lu.932, hamidouche.2, panda.2}@osu.edu

Abstract. Modern high-end computing is being driven by the tight integration of several hardware and software components. On the hardware front, there are the multi-/many-core architectures (including accelerators and co-processors) and high-end interconnects like InfiniBand that are continually pushing the envelope of raw performance. On the software side, there are several high performance implementations of popular parallel programming models that are designed to take advantage of the high-end features offered by the hardware components and deliver multi-petaflop level performance to end applications. Together, these components allow scientists and engineers to tackle grand challenge problems in their respective domains.

Understanding and gaining insights into the performance of end applications on these modern systems is a challenging task. Several researchers and hardware manufacturers have attempted to tackle this by designing tools to inspect the network level or MPI level activities. However, all existing tools perform the inspection in a disjoint fashion and are unable to correlate the data generated by profiling the network and MPI. This results in a loss of valuable information that can provide the insights required for understanding the performance of High-End Computing applications. In this paper, we take up this challenge and design InfiniBand Network Analysis and Monitoring with MPI - *INAM*^{2}. INAM^2 allows users to analyze and visualize the communication happening in the network in conjunction with data obtained from the MPI library. Our experimental analysis shows that the INAM^2 is able to profile and visualize the communication with very low performance overhead at scale.

1 Introduction and Motivation

Across scientific domains, application scientists are constantly looking to push the envelope by running large-scale, parallel jobs on supercomputing systems. Supercomputing systems are currently comprised of thousands of compute nodes based on modern multi-core architectures. Interconnection networks have rapidly evolved to offer low latencies and high bandwidths to meet the communication requirements of parallel applications. InfiniBand (IB) has emerged as a popular high performance network interconnect and is increasingly being used to deploy some of the top supercomputing installations around the world. The Message Passing Interface (MPI) [27] is a very popular parallel programming model for developing parallel scientific applications that run on such high end supercomputing systems.

As IB clusters and the MPI-based applications that use these clusters have become increasingly complex, understanding how an HPC application interacts with the underlying IB network and the impact it can have on the performance of the application becomes ever more challenging. It is critical for the users and administrators of HPC installations as well as developers of high performance MPI middleware that run on these HPC installations to clearly understand this interaction. 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. System administrators, application developers and developers of high performance parallel programming runtimes rely on a plethora of tools to accelerate and simplify the task of analyzing and understanding the various components of an HPC system.

One of the common questions system administrators tend to get from the users of the clusters they manage 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 (e.g.: Nagios [4], Ganglia [1], Mellanox Fabric IT [34], INAM [8], BoxFish [10]). 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 pin point 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, 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 (like TAU [5], HPCToolkit [18], Intel VTune [13], IPM [2], mpiP [3]) give reasonable insights into the MPI communication 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 issues such as increased traffic levels on one link 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. To address this concern, the MPI forum recently proposed the MPI_T [26] interface which allows MPI profiling tools to track the performance of various internal components of the MPI library. Researchers have already begun to take advantage of this interface to provide optimization and tuning hints to the users [14]. However, these tools have no knowledge about the underlying IB fabric 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 IB 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 - ***How can we design a tool that enables in-depth understanding of the communication traffic on the InfiniBand network through tight integration with the MPI runtime?***

2 Contributions

In this paper, we take up this challenge and design *INAM*^{2} - 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 and MPI disjointly. We demonstrate how, through the profiling information provided by *INAM*^{2}, designers 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 show how, through the link analysis capabilities of *INAM*^{2}, system administrators can pin point the cause of network performance issues to a granularity of a process. Our experimental evaluation shows that the *INAM*^{2} is able to profile and visualize the communication with very little performance overhead at scale. To summarize, *INAM*^{2} provides 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

Note that many of the features and capabilities described in this paper are already publically available as part of OSU INAM package for free download at `http://mvapich.cse.ohio-state.edu/tools/osu-inam/`. While we chose MVAPICH2 for implementing our designs, any MPI runtime can be enhanced to perform similar data collection and transmission.

The rest of the paper is organized as follows. Section 3 gives a brief overview of InfiniBand MPI over IB. In Section 4 we present the framework and design of *INAM*^{2}. We evaluate and analyze the correctness and performance of *INAM*^{2} in various scenarios in Section 5. We present the possible use cases for *INAM*^{2} in Section 6. The currently available related tools are described in Section 7. Finally we summarize the conclusions and possible future work in Section 8.

3 Background

In this section, we provide the necessary background information for this paper.

3.1 InfiniBand

InfiniBand is a very popular switched interconnect standard being used by almost 47% of the Top500 Supercomputing systems [33] according to the Nov'15 listing. InfiniBand Architecture (IBA) [16] defines a switched network fabric for interconnecting processing nodes and I/O nodes, using a queue-based model. It supports two communication semantics: Channel Semantics (Send-Receive communication) over Reliable Connected (RC), Extended Reliable Connected (XRC), Dynamic Connected (DC), and Unreliable Datagram (UD); and Memory Semantics (Remote Direct Memory Access communication) over RC, DC and XRC. Both semantics can perform zero-copy transfers from source-to-destination buffers without additional host-level memory copies. RC is connection-oriented and requires dedicated QP for destination processes while the connection-less UD transport uses a single QP for all [24, 22]. XRC optimizes QP allocation by requiring each process to create only one QP per node [23]. DC on the other hand combines the scalability of UD by providing the capability to use just on DC end point to communicate with an peer while providing the high-end RDMA/atomic features available with RC and XRC.

3.2 MPI

Message Passing Interface (MPI) [27], is one of the most popular programming models for writing parallel applications in cluster computing area. 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. In this paper, we use modified MVAPICH2-X [25] based on the 2.2a release for our evaluations. However, our observations in this context are quite general and they should be applicable to other high performance MPI libraries as well.

3.3 MPI_T

The MPI Tools Information Interface (MPI_T) provides a standard mechanism for MPI tool developers to both inspect and tweak 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. Accessing the values of performance variables allows the software to peak under the hood of the MPI library to determine the state 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. Section 14.3 of the MPI 3 standard describes the MPI_T interface in full detail.

4 Design of INAM^2

The overall architecture of InfiniBand Network Analysis and Monitoring with MPI (INAM^2) is presented in Figure 1. It consists of four major design components: (1) OSU INAM daemon (osuinamd), (2) OSU INAM Database, (3) Java-based Webserver, and (4) Web-based front end for visualization. We will go into the details of each in the following sections.

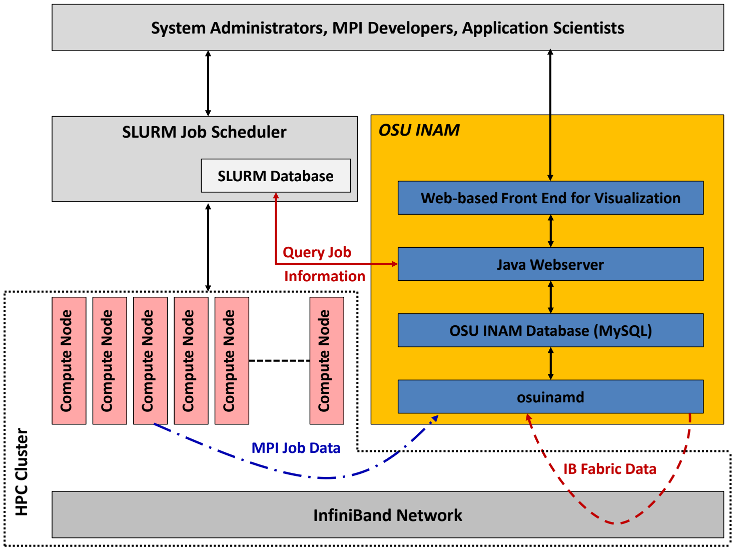


Fig. 1. Overall framework

4.1 Design of OSU INAM Daemon

The OSU INAM daemon is the hub for all data collection related activities in *INAM*^{2}. As we saw in Section 2, one of the major capabilities of *INAM*^{2} is its ability to interact with and extract information from the MPI processes and present data at network level, job level and process level granularities to the end users. Apart from this, the daemon is also responsible for discovering the IB fabric and extracting data from various selected components in the IB fabric. Finally, it is also responsible for pushing all collected data elements into the OSU INAM database (described in Section 4.2). In order to allow these tasks to proceed in parallel and not bottleneck each other, we dedicate a thread for each activity.

MPI Data Collection Thread While existing IB fabric monitoring tools like Nagios, Ganglia, and Mellanox Fabric IT are capable of displaying the overall state of the fabric, they're unable to break down the traffic and classify it at finer granularities (for instance at process level or as point-to-point or collective traffic) which can enable deep understanding. While one can theoretically use per virtual lane level counters and force MPI processes to use different virtual lanes to have process level granularity using existing tools, this method suffers from a fundamental issue — The IB standard only supports 16 virtual lanes. Given the current and emerging dense many-core nodes where the number of processes per node can be as high as 71, it becomes hard to perform a one-to-one mapping between processes and virtual lanes even at the node level. This fundamental bottleneck is further exacerbated by two mundane issues: 1) very few system administrators enable the use of multiple virtual lanes in practice on production supercomputing installations and 2) very few (if any) currently available IB products support per virtual lane level counters. Another advantage of using such an approach is that it frees us from the need to query the HCA on the node as the MPI process itself will send us the necessary details when the node is computing and while the node is not in use, we do not care about it as it is not expected to be contributing network traffic in a significant fashion.

To overcome these limitations, we designed and integrated the MPI data collection thread into the daemon process. The sole responsibility of this thread is to collect data specific to each MPI process running on the system and push it to OSU INAM Database. This allows us to analyze and visualize the data at job level, node level and process level granularities. We designed the thread to be a listener which accepts data from remote MPI processes to avoid the single point bottlenecks that can arise from a design where the thread actively polls each MPI process for data. The thread uses IB based communication to achieve high performance and low latency. The thread further uses the interrupt driven mode in IB to reduce CPU utilization by eliminating the need to continually poll to identify the arrival of new packets.

Design choices for IB transport protocol: As mentioned above, the MPI data collection thread uses IB to enable high performance and low latency communication. It is known that IB supports several transport protocols such as Reliable Connected (RC), Unreliable Datagram (UD), Extended Reliable Connected (XRC), and Dynamic Connected (DC) [23, 24]. Each transport protocol has different cost/performance tradeoffs. Our previous research has shown that using the UD and DC transport protocols over others can have significant benefits in terms of scalability and memory footprint [32, 23]. Thus we eliminate RC and XRC from the pool of possible protocols. The choice of whether to use DC or UD depends on the communication requirements. From the point of view of the *INAM^2*, the communication requirements are similar to what one would expect from a high performance stock market application - typically small messages, high performance, high scalability, low latency and no requirement for absolute reliability. Our previous research has shown that [32], between DC and UD, UD is able to deliver high performance, high scalability, low latency better than DC when reliability is not an issue. Thus, we choose the UD protocol as the IB transport protocol for the MPI data collection thread.

Co-designing the MPI runtime to work with *INAM^2* As we saw in Section 1, 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 [28] 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 buffer 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/GUID from which it's sending the data, 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 (e.g.: OpenMPI [12]) can be modified to perform similar data collection and transmission.

Fabric Discovery Thread The *Fabric Discovery* (FD) thread is responsible for discovering the IB fabric and extracting data from various selected components in the IB fabric. The fabric discovery has multiple phases. In the first phase, the thread uses methods similar to what is used by the "ibnetdiscover" utility to identify the various IB devices present in the network and their current status. The data is stored in an easy to retrieve format in the database. Once all the devices have been identified, it computes the network path between each pair of hosts and pushes this information into the database as well. Once this is done, the fabric thread will monitor the network for any changes at a user specified interval. Then, the *FD* thread switches over to retrieving the performance counter information from the network.

Different design choices were explored to retrieve the performance counter information from the IB fabric. Using an OpenSM plugin in for the *performance manager* module to extract the performance counter information. However, this method will cause data to be extracted from all network devices (including the end nodes themselves). As the MPI data collection thread is already capturing information at a much finer granularity than what can be delivered by the network level counters, it would be prudent to avoid the useless query to retrieve this information. To avoid this, the *FD* thread issues queries to selected components in the network at user specified intervals. In our case, the selected components would be various switches in the network. By doing this, we also reduce the amount of high-priority management traffic that is generated on the network. Although the default value for the query interval is 30 seconds, we recommend that users set it to a lower value as the "Xmit Data" and "Rcv Data" counters are only 32-bit and can easily overflow depending on the volume of data being transferred. On receiving a response, the *FD* thread queues up the message in a FIFO queue to the database thread for eventual insertion into the database.

Database Thread The *Database* (DB) thread is responsible for receiving information from the MPI data collection thread as well as the *FD* thread. When being run for the first time the *DB* thread will create all the tables in the schema that the given version of the tool expects. If an earlier version of the tool which used a different table scheme exists in the same system, it will automatically update them to avoid conflicts and make life easier for the user.

4.2 Design of OSU INAM Database

The Database design for *INAM*^{2} is critical since all necessary data needs to be stored in and queried from it. A useful and scalable database schema plays a key role for the system to achieve the flexibility and high-performance needed to scale with large clusters. Figure 2 shows the design of the *INAM*^{2} database. From this figure, we can see that through nine tables, *INAM*^{2} is able to cover all the capabilities as mentioned in Section 2 and we believe all these tables and fields are necessary to maintain all the important statistical data for both the InfiniBand network and the MPI processes and their correlations.

For instance, the fields in the tables of “route”, “links”, “nodes”, “port_data_counters”, and “port_errors” can hold all the important data for InfiniBand network infrastructure, like links, nodes, ports and routes. On the other hand, in order to keep track of MPI process communication characteristics, we utilize the tables of “process_info”, “process_comm_main”, and “process_comm_grid” to store MPI library counters and the communication paths over the links. Through these, *INAM*^{2} is able to analyze and profile node-level, job-level and process-level activities for MPI Point-to-point, collectives, and RMA communication. Further, this information can help to profile and report several important parameters/counters of MPI processes at the node-level, job-level and process-level as well as visualize the communication map at process-level and node-level granularities. Another example is analyzing and classifying InfiniBand network traffic flows in a physical link, through tables of “route”, “link_route”, and “links”, we are able to distinguish the traffic into those belonging to different jobs in conjunction with the MPI runtime. More analysis examples and scenarios will be discussed in Section 6.

4.3 Design of Java Webserver and Web-based Front-end Visualization

One of the most user-friendly features our *INAM*^{2} tool provides is the Web-based visualization. Through *INAM*^{2}’s Web front-end, system administrators, MPI developers and end users can easily understand the statistics data of the activities over underlying InfiniBand network and MPI jobs, which are gathered from the OSU INAM daemon and acquired from the Slurm job scheduler as shown in Figure 1. This information is organized and shown through Web pages in a way that can help users to correlate network level and MPI level behavior and identify the root causes of performance issues.

INAM^2 was not only designed for providing functionality, the high-performance design of the Web server and front-end will provide low latency as well as high throughput for users’ queries, so that users can profile the network and MPI job performance during the job execution. As shown in Figure 3, we designed the *INAM*^{2} Web server based on the Spring [15] MVC (Model, View and Controller) architecture which can be integrated easily with a Java Tomcat server. On the client side, we choose to use the light-weight JQuery [21] library to send HTTP requests through AJAX [20]. With the help from JQuery and AJAX, *INAM*^{2} pages can send data to and retrieve responses

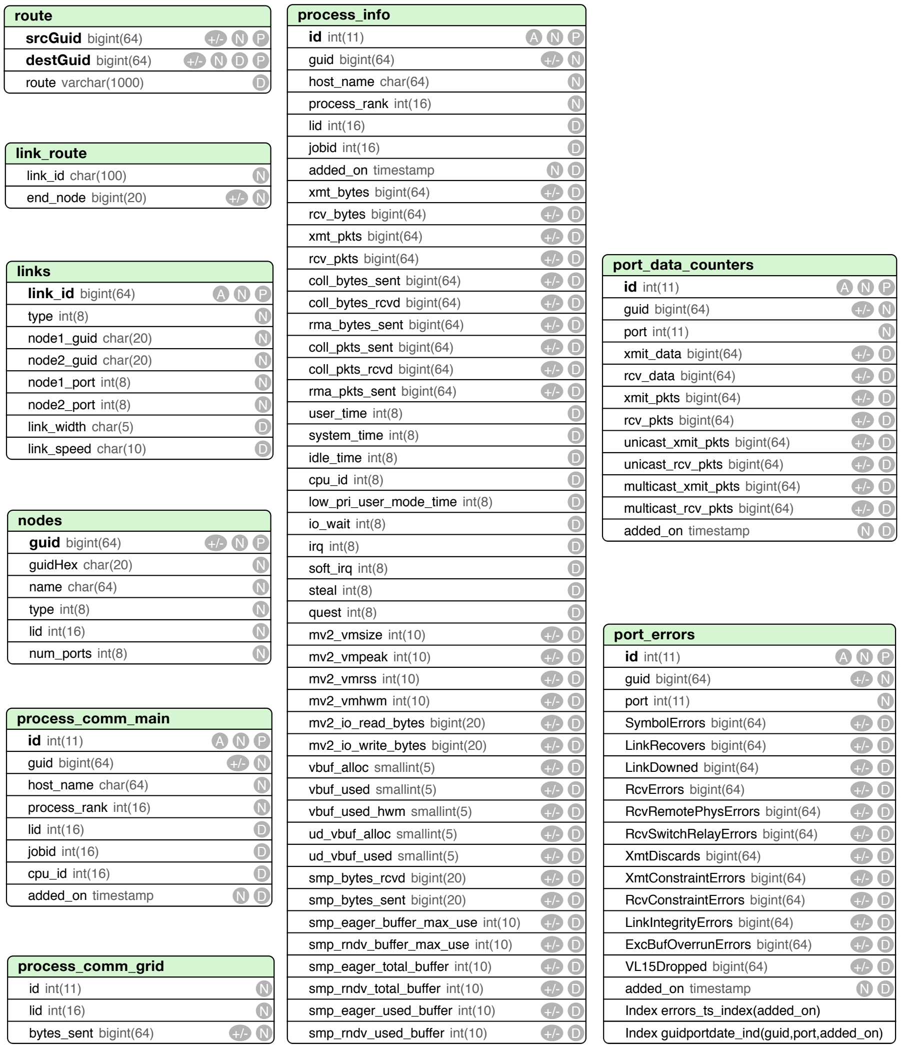

route
srcGuid bigint(64) +/- N P
destGuid bigint(64) +/- N D P
route varchar(1000) D
link_route
link_id char(100) N
end_node bigint(20) +/- N
links
link_id bigint(64) A N P
type int(8) N
node1_guid char(20) N
node2_guid char(20) N
node1_port int(8) N
node2_port int(8) N
link_width char(5) D
link_speed char(10) D
nodes
guid bigint(64) +/- N P
guidHex char(20) N
name char(64) N
type int(8) N
lid int(16) N
num_ports int(8) N
process_comm_main
id int(11) A N P
guid bigint(64) +/- N
host_name char(64) N
process_rank int(16) N
lid int(16) D
jobid int(16) D
cpu_id int(16) D
added_on timestamp N D
process_comm_grid
id int(11) N
lid int(16) N
bytes_sent bigint(64) +/- N
process_info
id int(11) A N P
guid bigint(64) +/- N
host_name char(64) N
process_rank int(16) N
lid int(16) D
jobid int(16) D
added_on timestamp N D
xmt_bytes bigint(64) +/- D
rcv_bytes bigint(64) +/- D
xmt_pkts bigint(64) +/- D
rcv_pkts bigint(64) +/- D
coll_bytes_sent bigint(64) +/- D
coll_bytes_rcvd bigint(64) +/- D
rma_bytes_sent bigint(64) +/- D
coll_pkts_sent bigint(64) +/- D
coll_pkts_rcvd bigint(64) +/- D
rma_pkts_sent bigint(64) +/- D
user_time int(8) D
system_time int(8) D
idle_time int(8) D
cpu_id int(8) D
low_pri_user_mode_time int(8) D
io_wait int(8) D
irq int(8) D
soft_irq int(8) D
steal int(8) D
quest int(8) D
mv2_vmsize int(10) +/- D
mv2_vmpeak int(10) +/- D
mv2_vmrss int(10) +/- D
mv2_vmhwm int(10) +/- D
mv2_io_read_bytes bigint(20) +/- D
mv2_io_write_bytes bigint(20) +/- D
vbuf_alloc smallint(5) +/- D
vbuf_used smallint(5) +/- D
vbuf_used_hwm smallint(5) +/- D
ud_vbuf_alloc smallint(5) +/- D
ud_vbuf_used smallint(5) +/- D
smp_bytes_rcvd bigint(20) +/- D
smp_bytes_sent bigint(20) +/- D
smp_eager_buffer_max_use int(10) +/- D
smp_rndv_buffer_max_use int(10) +/- D
smp_eager_total_buffer int(10) +/- D
smp_rndv_total_buffer int(10) +/- D
smp_eager_used_buffer int(10) +/- D
smp_rndv_used_buffer int(10) +/- D
port_data_counters
id int(11) A N P
guid bigint(64) +/- N
port int(11) N
xmit_data bigint(64) +/- D
rcv_data bigint(64) +/- D
xmit_pkts bigint(64) +/- D
rcv_pkts bigint(64) +/- D
unicast_xmit_pkts bigint(64) +/- D
unicast_rcv_pkts bigint(64) +/- D
multicast_xmit_pkts bigint(64) +/- D
multicast_rcv_pkts bigint(64) +/- D
added_on timestamp N D
port_errors
id int(11) A N P
guid bigint(64) +/- N
port int(11) N
SymbolErrors bigint(64) +/- D
LinkRecovers bigint(64) +/- D
LinkDowned bigint(64) +/- D
RcvErrors bigint(64) +/- D
RcvRemotePhysErrors bigint(64) +/- D
RcvSwitchRelayErrors bigint(64) +/- D
XmtDiscards bigint(64) +/- D
XmtConstraintErrors bigint(64) +/- D
RcvConstraintErrors bigint(64) +/- D
LinkIntegrityErrors bigint(64) +/- D
ExcBufOverrunErrors bigint(64) +/- D
VL15Dropped bigint(64) +/- D
added_on timestamp N D
Index errors_ts_index(added_on)
Index guidportdate_ind(guid,port,added_on)

Fig. 2. Overview of OSU INAM Database Design

from the server asynchronously without interfering with the display and behavior of the existing page. Such a solution will dramatically improve the user experience because it hides a lot of the data processing and page rendering in the background.

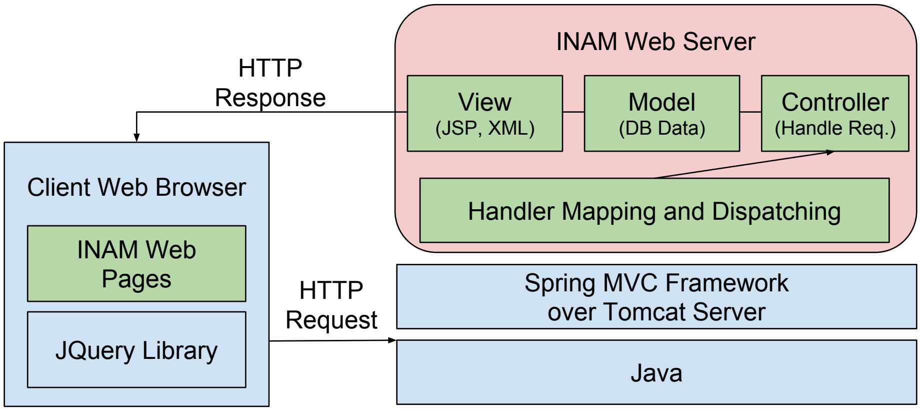


Fig. 3. Overall of INAM Web Server and Front-end

The overall processing flow is as follows: 1) Whenever a user's action generates an HTTP request, it will be sent to the server side by Web browser or JQuery library with AJAX; 2) Once the Tomcat server receives the request, it is passed to the 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. 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 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 get updated. Note that the whole process is completed very fast since all the data has been stored in database through the OSU INAM daemon in advance and all the processing steps are configured and indexed in the database. As indicated earlier, many users' actions are handled through AJAX which alleviates the need to reload the page for fresh data.

5 Experimental Results

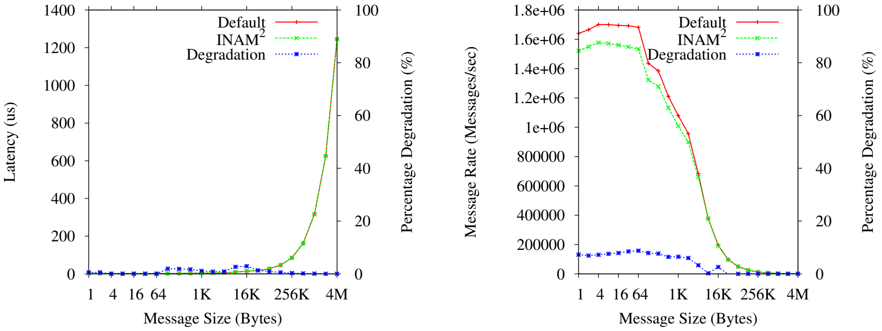
We describe the results of the various experiments carried out for this paper in this section.

5.1 Experimental Setup

Each node of our 184 node testbed has eight Intel Xeon cores running at 2.53 Ghz with 12 MB L3 cache. The cores are organized as two sockets with four cores per socket. Each node also has 12 GB of memory and Gen2 PCI-Express bus. They are equipped with MT26428 QDR ConnectX-2 HCAs with PCI-Express interfaces. We used a Mellanox MTS3610 QDR switch, with 11 leafs, each having 16 ports. Each node is connected to the switch using one QDR link. The HCA, as well as the switches, use the latest firmware. The operating system used is Red Hat Enterprise Linux Server release 6.5 (Santiago), with the 2.6.32-431.el6.x86_64 kernel version. Mellanox OFED version 2.2-1.0.1 is used on all machines.

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

In this section we study the impact the co-design of the MPI runtime with the MPI data collection thread of *INAM^2* has on basic communication performance of different point-to-point as well as collective microbenchmarks and popular application kernels like the NAS parallel benchmarks [9]. Figure 4a compares the basic point-to-point inter-node latency obtained with and without the data collection happening in the MPI runtime. As we can see, the data collection adds less than 1% degradation when compared to the native performance. In Figure 4b, we depict the inter-node message rate obtained with the osu_mbw_mr microbenchmarks using a pair of processes. We see that the data collection and transmission adds about 6% to 8% overhead for messages less than 4,096 bytes. However, for larger messages, we see no significant impact at all (less than 1%).



(a) Latency

(b) Message Rate

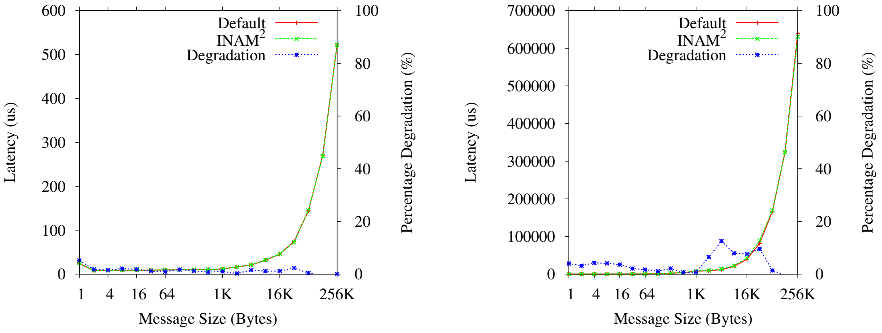
Fig. 4. Microbenchmark-level point-to-point performance

Figures 5a and 5b depict the performance impact the data collection and transmission has on some common collective communication patterns such as Broadcast and Alltoall, respectively. The evaluations were done at a scale of 512 processes. As we can see, the tool adds less than 5% overhead for Broadcast. For Alltoall, we observe less than a 5% degradation for messages up to 1,024 bytes. For larger messages the degradation is mostly around 7% with only 4,096 byte message showing up to 12% degradation.

Figure 6 compares the performance of the version of the MPI runtime with support for MPI level data collection with one which does not have the support. As we can see, at the application level, there is little to no impact on the performance due to the addition of the data collection and reporting. These are encouraging trends which positively advocate the use of such tools for end applications on modern supercomputing systems.

6 Discussion on Features of INAM^2 and its Impact

In this section, we highlight some of the many features of INAM^2 and describe some of the potential impact it can have on the understanding and performance of applications.



(a) Performance of Broadcast

(b) Performance of Alltoall

Fig. 5. Microbenchmark-level collective performance at 512 processes

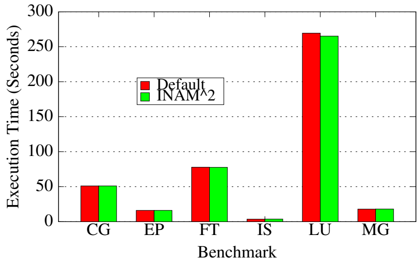


Fig. 6. Performance of class D NAS parallel benchmarks at 512 processes

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

Several high performance implementations of the the MPI programming model allocate a set of internal communication buffers that have been pre-registered with the IB HCA to enable fast small message communication. MVAPICH2, for instance, has been extensively tuned to ensure that the number and size of communication buffers is large enough to maintain good communication performance without significantly increasing the amount of memory consumed for these buffers. The memory footprint is even more important for these as they are always "pinned" to the physical memory and cannot be swapped out in case the application requires more memory to do its computation. However, at the level of end applications, one cannot assure that all the internal communication buffers that have been pre-allocated and pinned are being used by the application for communication. We use the INAM^2 tool to profile and understand the communication behavior of the NAS benchmarks and visualize how they use the internal communication buffers that MVAPICH2 allocates. Once the job has completed execution, we monitor the high water marker for the internal communication buffer usage over the lifetime of the job using the "historical job view" that INAM^2 offers. Table 1 shows the results of our analysis. It 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 inter-node 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 INAM^2. 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.

Table 1. Comparison of communication buffer utilization for default and tuned scenarios for 512-process class D NAS parallel benchmarks

Benchmark	Default-HWM (Max Value)	Default-Alloc (Max Value)	Default-Communication Buffer-Memory (Sum) (MB)	Tuned-HWM (Max Value)	Tuned-Alloc (Max Value)	Tuned-Communication Buffer-Memory (Sum) (MB)
CG	1	240	1570.20	2	48	409.33
EP	1	240	1570.20	3	48	348.22
FT	356	544	1735.49	295	320	647.24
LU	161	352	1584.74	152	192	503.76
MG	30	240	1570.20	32	80	561.33

6.2 Identifying and Analyzing Sources of Link Congestion

Existing IB fabric monitoring tools are capable of identifying congested links in the fabric. However, identifying the network “hot spots” alone is not good enough for system administrators. What they are looking for is the source of the congestion. Unfortunately, no tool offers the kind of automatic “reverse-lookup” feature that allows one to identify the various sources (end compute nodes) that could possibly have routes through the link in question. On a typical network with dynamic routing, doing this would prove to be a near insurmountable challenge. However, as IB networks are typically statically routed, it becomes a challenge that can be solved. We tackle and solve this challenge in INAM^2 using the various tables described in Figure 2. Figure 7 depicts how one can identify the various routes going through the link. As we can see, the different paths that go through a given link gets highlighted in yellow. We actually go one step further and provide the capability to analyze and classify the traffic flowing in a physical link into those belonging to different jobs in conjunction with the MPI runtime allowing system administrators to identify exactly which job was contributing to the traffic going over a particular link. Users can view the link utilization by the jobs sending/receiving data through it in both directions in an absolute (in terms of number of bytes)or relative sense (as a percentage of total link capacity). Figure 8 depcits how, by selecting a job id, INAM^2 can process level link utilization for the selected job. Sections 4.2 and 4.3 describe how data is fetched from various tables to construct and display this novel feature.

6.3 Monitoring Jobs Based on Various Metrics

While typical job schedulers list what nodes are being used by which jobs, they do not list what each individual job is currently doing and how that impacts the different components of the HPC system. For instance, if a job is dumping a lot of data to the file

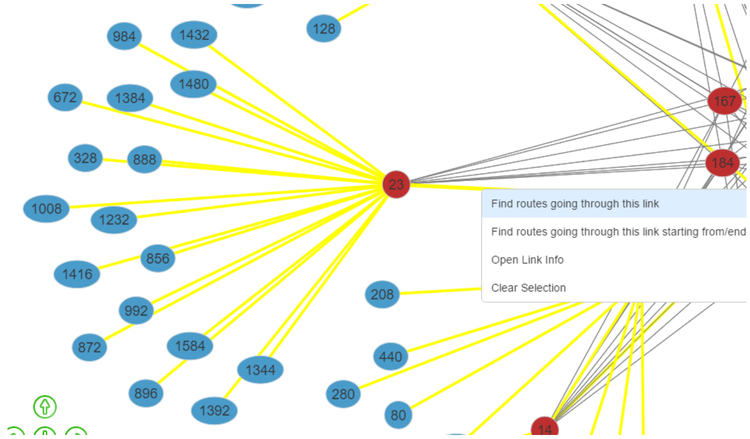
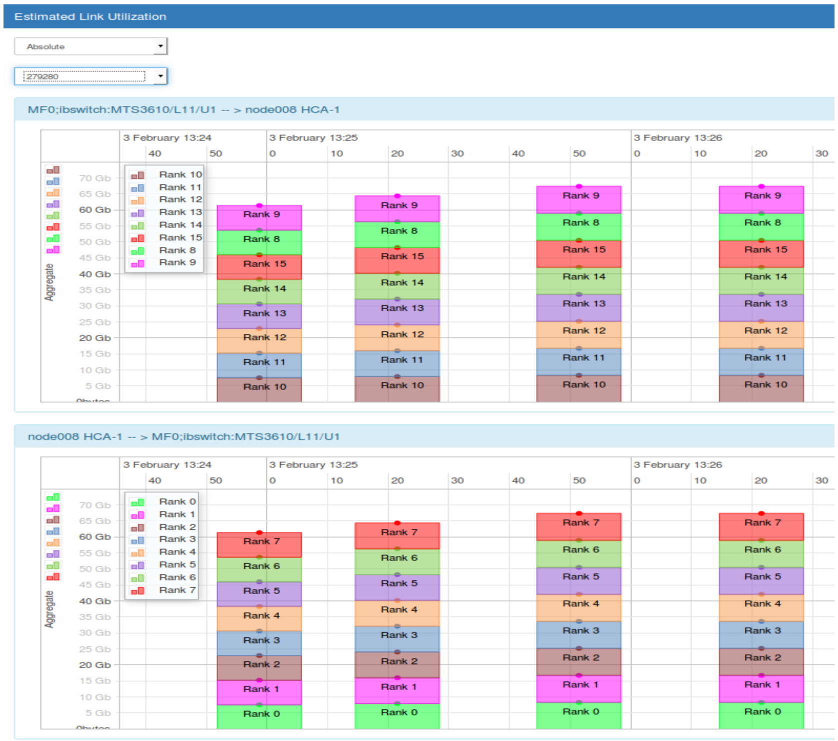


Fig. 7. Identifying communication routes going through a given link


Estimated Link Utilization

Absolute

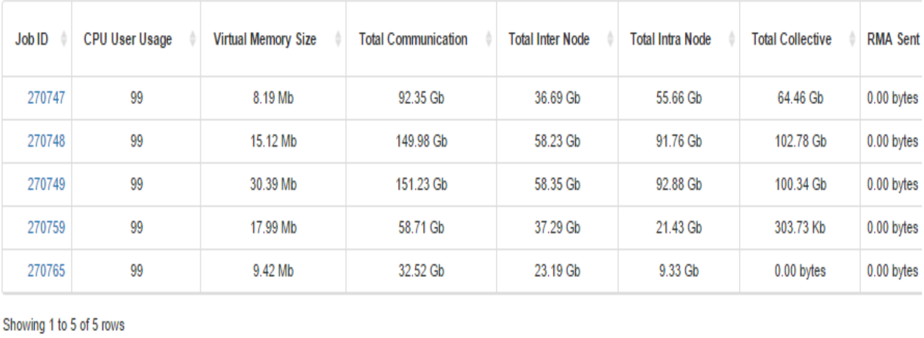
279280

MF0;ibswitch:MTS3610/L11/U1 -- > node008 HCA-1

node008 HCA-1 -- > MF0;ibswitch:MTS3610/L11/U1

Fig. 8. Process level link utilization for a user specified job

system due to a checkpoint operation or because it has encountered a segmentation fault and is currently in the process of dumping cores, it is going to negatively affect all other processes in the system. Similarly, if a job is performing a network intensive communication operation, like an Alltoall, all jobs may get affected. Thus, it is in the best interest of all concerned that such ”high-value” jobs be closely monitored by the system administrator. To address this concern, we introduce a “Live Job” page in INAM^2 which lists all MPI jobs that are sending data to it through the MPI data collection framework. The page allows sorting the various jobs in ascending/descending order of the various metrics listed in the “process_info” table depicted in Figure 2. Figure 9 shows an example of how this page would look like on a real cluster scenario with various jobs running. As we can see, each job ID is a hyperlink which takes the user to the “job page” for the corresponding job so that the user can get more details of what exactly is going on in the job.



Job ID	CPU User Usage	Virtual Memory Size	Total Communication	Total Inter Node	Total Intra Node	Total Collective	RMA Sent
270747	99	8.19 Mb	92.35 Gb	36.69 Gb	55.66 Gb	64.46 Gb	0.00 bytes
270748	99	15.12 Mb	149.98 Gb	58.23 Gb	91.76 Gb	102.78 Gb	0.00 bytes
270749	99	30.39 Mb	151.23 Gb	58.35 Gb	92.88 Gb	100.34 Gb	0.00 bytes
270759	99	17.99 Mb	58.71 Gb	37.29 Gb	21.43 Gb	303.73 Kb	0.00 bytes
270765	99	9.42 Mb	32.52 Gb	23.19 Gb	9.33 Gb	0.00 bytes	0.00 bytes

Showing 1 to 5 of 5 rows

Fig. 9. Live job page to display jobs in ascending/descending order of various performance metrics in conjunction with the MPI runtime

6.4 Capability to Profile and Report Several Metrics of MPI Processes at Different Granularities

One of the dangers of providing users with too much data is the possibility of inundating them with so much information that the high value data items get lost in the deluge of less relevant details. Thus, it is always helpful if one can aggregate and display the information to users so that they are first presented with a high level view first (e.g.: a cluster level or job level) and then allowed to slowly dig their way into more details (e.g.: node level or process level views). We provide this exact capability in INAM^2. Although the data from the MPI processes arrive at a process level granularity, once it has been entered into the database, things can be easily manipulated so that we can aggregate and display the details at much coarser granularities (e.g.: node level or job level). Figures 10 and 11 depict examples of the live job-level view of a given job and node level view of different processes that belong to a job respectively as rendered by INAM^2. Further, INAM^2 allows such analysis to be done in a “live” or a “historical” manner. This capability of INAM^2 to display historical information can prove very useful to system administrators. For instance, it is quite possible that the system or network administrator is made aware of an issue “post-mortem”. In such scenarios, the current IB fabric monitoring tools, which do not have support to store information in databases for later retrieval, more or less leave the administrators helpless. However, if an administrator has access to a tool like INAM^2 which has the ability to “play back” events that occurred at a specified time in the past, it provides administrators the flexibility to inspect events “post-mortem” and identify the culprit(s) that caused the issue.

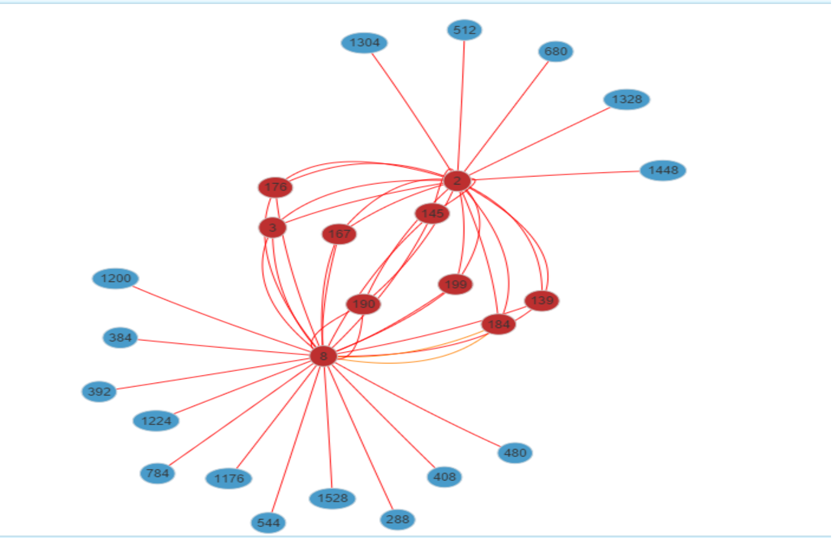


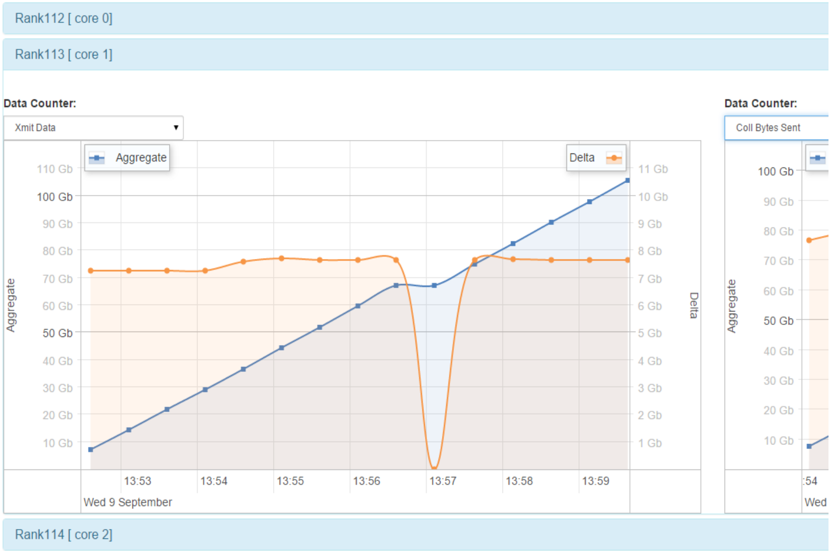
Fig. 10. Live job level view of a particular job

7 Related Tools

We believe that *INAM^2* fills a void in the tools space as many do not monitor and correlate the impact of particular MPI jobs on the system. In principle, the pre-existing tool most closely related in design is *Lightweight Distributed Metric Service (LDMS)* [6] by Sandia. It strives to be a low overhead system monitoring tool which also correlates jobs to the impact on the system. LDMS does not monitor the InfiniBand network directly as *INAM^2* but does a good job monitoring other resources such as memory or filesystem I/O.

Another tool suite available is *HOlistic Performance System Analysis (HOPSA)* [7]. This suite is more focused on application on MPI but is designed in a way where each application or job may have different metrics monitored which may not allow for a full system view of how the set of jobs are interacting on the system.

The group at the Texas Advanced Computing Center have also developed a tool *TACC STATS* [31] to help them to explore job and system level reports. These reports help to identify jobs or system components that may need attention based on policies that they’ve set forth ahead of time. The main difference between this tool and *INAM^2*



Rank112 [core 0]

Rank113 [core 1]

Data Counter: Xmit Data

Data Counter: Coll Bytes Sent

Rank114 [core 2]

Fig. 11. Live node level view of different processes that are part of a particular job

is that *TACC STATS* does their analysis post-mortem whereas we try to make this information available in real time.

As mentioned earlier, several tools exist which allow system administrators to analyze and inspect the IB fabric such as Nagios, Ganglia, Mellanox Fabric IT, INAM, and BoxFish. However, due to lack of in-depth knowledge about the MPI library, no existing IB fabric monitoring tool can correlate the network level and MPI level behavior to classify traffic as being generated by particular MPI primitives. Furthermore, they cannot classify network traffic as belonging to a particular job due to the lack of interaction with the job scheduler.

Also mentioned earlier, existing MPI level profiling tools like TAU, HPCToolkit, Intel VTune, IPM, and mpiP give reasonable insights into the MPI communication 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. With *INAM*^{2} we strive to bridge the gap between the system level and MPI level profilers and monitors.

8 Conclusions and Future Work

In this paper, we presented the design of *INAM*^{2} - 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. We demonstrated how, through the profiling information provided by *INAM*^{2}, designers 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 *INAM*^{2}, system administrators can pin point the cause of network performance issues to a granularity of a process. Several features of *INAM*^{2} presented in this paper 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 the *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 part of future work, we plan to incorporate support for additional MPI_T counters in conjunction with the MPI library. We would also like to extend INAM^2 to be capable of profiling and analyzing communication taking place to and from GPGPUs. Further, we would like to add the capability to profile various PGAS programing languages such as OpenSHMEM [29], UPC [35] and CAF [11] as well as different BigData frameworks like Apache Hadoop [17], MapReduce [19] and Spark [30].

9 Acknowledgements

We would like to thank Michael Knox from Cray and John Hanks from KAUST for their feedback on the OSU INAM package and thus enabling us to fix several bugs and performance issues.

References

1. Ganglia Cluster Management System. http://ganglia.sourceforge.net/.
2. Integrated Performance Monitoring (IPM). http://ipm-hpc.sourceforge.net/.
3. mpiP: Lightweight, Scalable MPI Profiling. http://www.llnl.gov/CASC/mpip/.
4. Nagios. http://www.nagios.org/.
5. A. D. Malony and S. Shende. Performance Technology for Complex Parallel and Distributed Systems. In *Proc. DAPSYS 2000, G. Kotsis and P. Kacsuk (Eds)*, pages 37–46, 2000.
6. A. Agelastos, B. Allan, J. Brandt, P. Cassella, J. Enos, J. Fullop, A. Gentile, S. Monk, N. Naksinehaboon, J. Ogden, M. Rajan, M. Showerman, J. Stevenson, N. Taerat, and T. Tucker. The Lightweight Distributed Metric Service: A Scalable Infrastructure for Continuous Monitoring of Large Scale Computing Systems and Applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 154–165, Piscataway, NJ, USA, 2014. IEEE Press.
7. HOPSA – Holistic Performance System Analysis. http://www.vi-hps.org/projects/hopsa/overview.
8. OSU InfiniBand Network Analysis and Monitoring. http://mvapich.cse.ohio-state.edu/tools/osu-inam/.
9. D. H. Bailey, E. Barszcz, L. Dagum, and H.D. Simon. NAS Parallel Benchmark Results. Technical Report 94-006, RNR, 1994.
10. PAVE Software Boxfish. https://computation.llnl.gov/project/performance-analysis-through-visualization/software.php.
11. Coarray Fortran (CAF). http://caf.rice.edu/.
12. Open MPI : Open Source High Performance Computing. http://www.open-mpi.org.
13. Intel Corporation. Intel VTune Amplifier. https://software.intel.com/en-us/intel-vtune-amplifier-xe.
14. E. Gallardo, J. Vienne, L. Fialho, P. Teller and J. Browne. MPI Advisor: A Minimal Overhead MPI Performance Tuning Tool. In *EuroMPI 2015*, 2015.
15. Spring Framework. http://projects.spring.io/spring-framework/.
16. G. Pfister. Aspects of the InfiniBand Architecture. In *2001 IEEE International Conference on Cluster Computing (CLUSTER)*, page 369. IEEE Computer Society, 2001.
17. Apache Hadoop. https://hadoop.apache.org/.

18. HPCToolkit. http://hpctoolkit.org/.
19. J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.
20. Asynchronous JavaScript and XML. http://www.w3schools.com/Ajax/ajax_intro.asp.
21. Jquery. https://jquery.com/.
22. M. Koop, T. Jones, and D. K Panda. MVAPICH-Aptus: Scalable High-performance Multi-transport MPI over InfiniBand. In *IPDPS'08*, pages 1–12, 2008.
23. M. Koop, J. Sridhar, and D. K. Panda. Scalable MPI Design over InfiniBand using eXtended Reliable Connection. *IEEE Int'l Conference on Cluster Computing (Cluster 2008)*, September 2008.
24. M. Koop, S. Sur, Q. Gao, and D. K. Panda. High Performance MPI Design using Unreliable Datagram for Ultra-scale InfiniBand Clusters. In *ICS '07: Proceedings of the 21st annual international conference on Supercomputing*, pages 180–189, New York, NY, USA, 2007. ACM.
25. J. Liu, W. Jiang, P. Wyckoff, D. K. Panda, D. Ashton, D. Buntinas, W. Gropp, and B. Toonen. Design and Implementation of MPICH2 over InfiniBand with RDMA Support. In *Proceedings of Int'l Parallel and Distributed Processing Symposium (IPDPS '04)*, April 2004.
26. M. Schulz. MPIT: A New Interface for Performance Tools in MPI 3. http://cscads.rice.edu/workshops/summer-2010/slides/performance-tools/2010-08-cscads-mpit.pdf.
27. Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, Mar 1994.
28. MVAPICH2-X: Unified MPI+PGAS Communication Runtime over OpenFabrics/Gen2 for Exascale Systems. http://mvapich.cse.ohio-state.edu/.
29. OpenSHMEM. http://openshmem.org/site/.
30. Apache Spark. http://spark.apache.org/.
31. TACC STATS. https://www.tacc.utexas.edu/research-development/tacc-projects/tacc-stats.
32. H. Subramoni, K. Hamidouche, A. Venkatesh, S. Chakraborty, and D. K Panda. Designing MPI Library with Dynamic Connected Transport (DCT) of InfiniBand: Early Experiences. In *Supercomputing*, volume 8488 of *Lecture Notes in Computer Science*, pages 278–295. Springer International Publishing, 2014.
33. Top 500 Supercomputers. http://www.top500.org/statistics/list/.
34. Mellanox Technologies. Mellanox Integrated Switch Management Solution. http://www.mellanox.com/page/ib_fabricit_efm_management.
35. Unified Parallel C (UPC). http://upc.lbl.gov/.