論文 / 著書情報
Article / Book Information

Title	Hardware-Centric Analysis of Network Performance for MPI Applications
Author	Kevin Brown, Jens Domke, Satoshi Matsuoka
Journal/Book name	2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS), , , Page 692-699
Issue date	2015, 12
DOI	https://doi.org/10.1109/ICPADS.2015.92
URL	http://www.ieee.org/index.html
Copyright	(c)2015 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other users, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works for resale or redistribution to servers or lists, or reuse of any copyrighted components of this work in other works.
Note	このファイルは著者（最終）版です。 This file is author (final) version.

Hardware-Centric Analysis of Network Performance for MPI Applications

Kevin A. Brown
Dept. of Mathematical and Computing Sciences
Tokyo Institute of Technology
Meguro-ku, Tokyo 152-8550, Japan
Email: brown.k.aa@m.titech.ac.jp

Jens Domke
Faculty of Computer Science
Technische Universität Dresden
Dresden, Germany
Email: jens.domke@tu-dresden.de

Satoshi Matsuoka
GSIC
Tokyo Institute of Technology
Meguro-ku, Tokyo 152-8550, Japan
Email: matsu@is.titech.ac.jp

***Abstract*—As the scale of high-performance computing systems increases, optimizing inter-process communication becomes more challenging while being critical for ensuring good performance. However, the hardware layer abstraction provided by MPI makes it difficult to study application communication performance over the network hardware, especially for collective operations. We present a new approach to network performance analysis based on exposing low-level communication metrics in a flexible manner and conducting hardware-centric analysis of these metrics. We show how low-level network metrics can be revealed using Open MPI's Peruse utility, without interfacing with the hardware layer. A lightweight profiler, ibprof, was developed to aggregate these metrics from message passing events at a cost of <1% runtime overhead for communication in NPB kernel and application benchmarks. We also developed a flexible visualization module for the Boxfish analysis tool to analyze our communication profile over the physical topology of the network. Using case studies, we demonstrate how our approach can identify communication anomalies in network applications and guide performance optimization strategies.**

***Keywords*—Performance analysis, profiling, Open MPI, Peruse, Boxfish**

I. Introduction

High-performance computing (HPC) systems are rapidly growing in physical size and complexity, with staggering increases in node counts over recent years. Based on the TOP500 list of July 2015 [1], each of the five fastest supercomputers has over 15,000 compute nodes interconnected using various topologies. Performing inter-node communication over these topologies is non-trivial and has a significant impact on the overall application throughput. This is especially true for communication-bound applications [2]. Because of this, optimizing communication within these large-scale applications is a standard approach in performance tuning on these massive systems.

Performance tuning efforts, however, are complicated by the use of message passing libraries to simplify inter-process communication. Message passing libraries, such as those that implement the Message Passing Interface (MPI) [3], provide a single abstraction for the hardware layer, its technologies, and its topologies. For all MPI operations, the actual method of data transmission over the network hardware is hidden within the MPI library's implementation and is invisible to the application. Collective operations impose yet another layer of abstraction by concealing the manner in which participating processes exchange data in the MPI layer. This results in the need for communication optimization efforts to span multiple layers of abstractions: the application layer, the logical message passing layer, and the physical data transmission (hardware) layer. Network designers and MPI library developers, especially, require visibility on how communication in the application layer influences network utilization and vice-versa.

This work builds on our previously published poster [4] by defining a comprehensive approach to performance analysis that provides insight into the network performance of complex MPI operations and applications. Our approach involves exposing low-level communication events in a portable way by adding a new event to Open MPI's Peruse utility [5], [6], a performance revealing extension included with Open MPI. These events are then tracked using a lightweight profiler that we developed, named `ibprof`, which incurs <1% runtime overhead for communication in NPB benchmarks. The communication profiles generated by `ibprof` are visualized in a hardware-centric manner, whereby the performance data is mapped onto the physical nodes and links in the network. Visualization is conducted using a flexible visualization module that we developed for Boxfish [7].

The unique contributions of this work are:

1) We describe the design considerations when tracking network traffic per receiver process within MPI.
2) We explain the designs of `ibprof`, a lightweight profiler created to generate communication profiles using Peruse, and our flexible visualization module for Boxfish.
3) We provide an assessment of the memory and runtime overhead of `ibprof`.
4) We show the effectiveness of our analysis approach in identifying communication bottlenecks.

The remainder of this paper is organised as follows. Section II discusses the background and motivation for this work. Sections III and IV describe the technologies that are relevant to our work and the implementation of our solution, respectively. We then provide an evaluation of our solution in Sections V and VI. Finally, we assess related studies in Section VII and present our conclusion in Section VIII.

II. Background and Motivation

A. The Importance of Effective MPI Performance Analysis

The performance of message passing operations in communication-bound application is dependent on, among other things, two main factors: the implementation of the MPI library and the configuration of the network technologies used to undertake the operation [8]–[10]. The MPI library's implementation determines the logical order and semantics used to exchange messages among processes while the network configuration defines the manner in which packets are communicated over the network hardware. To truly understand the performance of communication-bound applications, it is therefore imperative to comprehensively analyze the performance of MPI operations over the network.

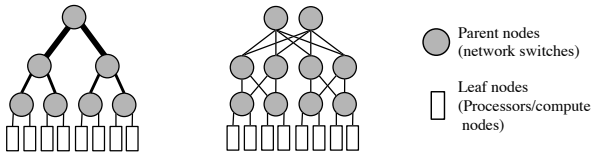
B. Revealing Performance in MPI Operations

Guaranteeing optimal performance of communication-bound applications requires an assurance that the MPI libraries and networking technologies are performing optimally. This assurance can only be attained by analysing the MPI library's operations over the network hardware, down to the level of the physical links. The MPI standard's [3] performance revealing tools, PMPI and MPI_T, either operate above the MPI layer or are too strict in their management of internal performance variables. Hence, they cannot precisely correlate application communication performance with network performance. For these reasons, PMPI and MPI_T are currently unsuitable for tracking network traffic with the granularity required to map MPI messages to network link usage.

The meaningful analysis of complex HPC applications running on advanced network technologies is a daunting task. It requires user-friendly, non-intrusive, low-interference tools that can penetrate multiple layers of software and hardware abstraction to accurately represent the application's performance across the network. Otherwise, overly-intrusive analysis efforts will yield diminishing returns, and oversimplification of the environment will stymie optimization efforts.

C. The Disadvantage of the Network-based Approach

Some communication analysis approaches directly target the network layer. Network management toolsets can be leveraged in these cases to measure network performance at a coarse granularity by sampling port counters for data traffic, etc. However, the network metrics retrieved by these toolsets are typically application-agnostic and are unable to distinguish between the traffic of different applications. This results in the network-based approach being restrictive and can result in sub-optimal system utilization. Assume, for example, an HPC system being used by two users, each running a different application. If user 1 attempts to measure the communication performance of his application by sampling network-wide port counters, his measurements will be inaccurate since the counter values include data for applications being run by both users. The only way to get accurate measurements in this case is to prevent user 2 from using the system during the experiment. This is very inefficient system utilization since the computation and communication capacities can accommodate the simultaneous execution of both applications.



(a) Binary fat-tree (b) Extended generalized fat-tree

Fig. 1. **Fat-tree topologies** Line thickness indicate the link's relative capacity.

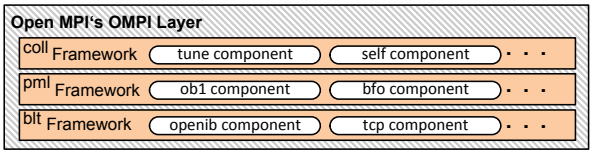


Fig. 2. **Open MPI MCA frameworks** — A graphical representation of selected frameworks in the OMPI layer and their related components

III. Technology

This section provides a technical introduction to the areas relevant to our work, setting the context in which to explain our design and implementation in the subsequent section.

A. InfiniBand Fat-tree Network

One of the most widely used network technologies for HPC systems is InfiniBand (IB) [11], a low latency, high throughput, switched networking architecture. IB hardware provides full-duplex connectivity along with kernel-bypass and remote direct memory access (RDMA) facilities. The set of interconnected nodes (switches, compute nodes, storage nodes, etc.) in an IB network are referred to as a subnet. The `ibdiagnet` utility can be used to query the configuration of subnets, including port configuration information and the port forwarding tables of switches. Active ports on network adapters are assigned local identifiers (LIDs), which are unique to their subnet.

A fat-tree [12], [13] is a hardware-efficient, hierarchical network topology that preserves full bisection bandwidth across the entire network as illustrated in Fig. 1. Because of its benefits, fat-tree networks are used by some of the top HPC systems, most notably Tianhe-2 [14] – the TOP500's fastest supercomputer since June 2013.

B. Open MPI

Open MPI is one of the most widely used MPI libraries and has been chosen for this research because it implements the Peruse interface (see Section III-C). Internal services within Open MPI are defined by frameworks and implemented as components in its Modular Component Architecture (MCA). Each component of a given framework contains a unique implementation of the services handled by that framework. Modules are the runtime instantiations of components. Figure 2 shows a visual description of Open MPI's OMPI code base.

Our focus in this work is on IB network traffic, hence we are most concerned with the `openib` component in the `btl` framework, which manages inter-process data transfer via IB channel adapters. The `openib` component uses the `ibverbs` API to interface with IB adapters and all MPI operations involving the IB channel adapters use this component.

C. Peruse

The Peruse utility was proposed as a performance revealing extension to the MPI standard that allows the tracking of internal events within an MPI library [6]. Peruse accomplishes this by registering a user-defined callback function to each event of interest within the MPI library. Two examples of these events in an `MPI_Send` operation are (1) the point when the MPI library begins processing the send request and (2) the point when the actual data transmission begins. The Peruse standard defines the interface for registering callback functions, the function prototype for callback functions, and the methods for enabling and disabling events.

Keller et. al [15] describes the details of implementing Peruse in Open MPI. Their research reported a 1.7% increase in communication latency when using Open MPI with Peruse versus the native Open MPI on an InfiniBand network. By design, Peruse in Open MPI is very extensible and provides the flexibility we need to easily track network traffic generated by an application. The existing implementation of Peruse in Open MPI is entirely contained within the OMPI code section (see Fig. 2), and all events are tracked within `pml` framework components.

D. Boxfish

BoxFish is a python-based performance analysis tool that is capable of visually representing the physical nodes and links in a network [7]. It uses visualization modules to present performance data in different forms: (1) a `Table` module presents data in tabular form, (2) a `3D Torus` module constructs a 3D torus network topology and presents performance data as the colors of the network elements, etc. The same performance metrics may be presented simultaneously in multiple modules and filters can be applied to modules individually and in groups. Furthermore, these metrics may span multiple domains: application, hardware, and communication domains.

The core of BoxFish handles the reading of performance data from input files and stores the information in a generic, module-independent form. Modules request and receive data from the Boxfish core, which also manages the linking of performance data across multiple modules. The current version of Boxfish has network visualization support for only torus topologies.

IV. Design and Implementation

The general design idea is to visualize the communication traffic of MPI applications over the physical links of the network. Moreover, the performance data used for this hardware-centric visualization should be exposed to the application layer while expressing low-level data movement in the network layer. This results in analysis being conducted at a high level while giving meaningful insights into low-level performance.

A. InfiniBand Fat-Tree Visualization

We developed a Boxfish visualization module, named `Fat Tree` module, to provide flexible hardware-centric visualizations. Our module is based on the structure of the `3D Torus` module that is distributed with the Boxfish tool. However, our module differs in the way performance data is referenced and visualized. With the `3D Torus` module, a link is referenced by a combination of a 3-point Cartesian coordinate ($[x, y, z]$) and its direction along a Cartesian axis ($\pm x$, $\pm y$, or $\pm z$) in 3-dimensional (3D) space. Our module references each link using an explicit pair of source node location and target node location ($[x_1, y_1][x_2, y_2]$) in 2-dimensional (2D) space. This generalized method of representing links gives our module the flexibility to visualize any 2D network topologies, for example: fat-tree, 2D mesh, and dragonfly networks. To ensure that we accurately represent the bidirectional flow of traffic, we designed the module so that each end of a link is colored independently, i.e., based on the traffic sent over the link from the node connected at that end.

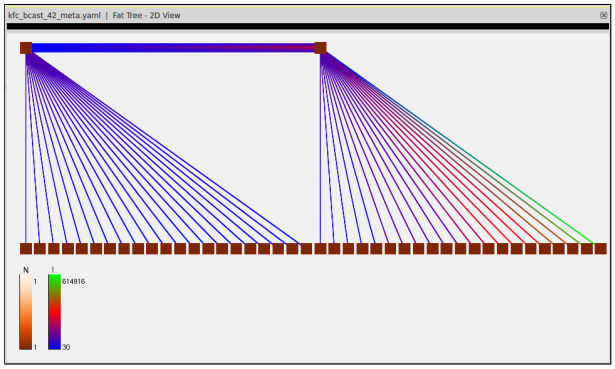


Fig. 3. **Boxfish `Fat Tree` module** — sample visualization

Fig. 3 shows the visualization of the communication profile for a `MPI_Bcast` running on a testbed at Tokyo Tech. The two multicolored bars in the bottom-left corner of the image indicate the color-value map that is used for nodes and links: "N" for nodes and "L" for links. The two brown squares at the top of the image represent the two switches and the 42 squares that are arranged horizontally below them are the management and compute nodes. Each line drawn between two squares represents a physical network link between those two nodes. The two switches at the top of the network image are interconnected with 15 individual links. Node colors are of no significance in this sample visualization while link colors reflect the amount of application traffic that was sent over the link. In the case of the right-most link, the end that connects to the server is green while the opposite end is blue. This means that the node connected at the bottom (a compute node) injects a relatively large amount of data onto that link while the node connected at the top (a switch) sends only a small amount in the opposite direction.

B. Exposing Low-level Traffic via Peruse

We exposed the low-level application traffic metrics required for our visualization by creating a new event, named `PERUSE_OPENIB_SEND`, in the Peruse code base of Open MPI. To make this event meaningful in relating inter-process communication with network link traffic, we required information on which network ports are involved in the communication and the type of traffic being communicated, i.e, user data or control messages. The `peruse_comm_spec_t` structure was extended to include variables that store the source and destination port LIDs as well as the type of data being sent.

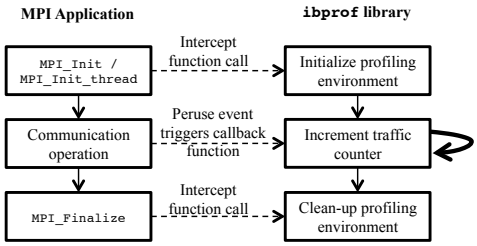


Fig. 4. **ibprof overview** — For each of the listed operations performed by the application, we show the corresponding action performed by our profiler.

We added the PERUSE_OPENIB_SEND event at a lower level to other Peruse events, in the openib btl component (see Fig. 2), where the port information is available. Our event is triggered immediately after each call to ibv_post_send, which is the ibverbs command for sending data via IB adapters. This allows for overlapping the event processing with data communication, since ibv_post_send returns immediately after offloading the send request to the IB adapter.

C. ibprof Profiler

The profiling tool that we built to record information from the PERUSE_OPENIB_SEND events, named ibprof, was implemented as a shared library that can be preloaded to an application's unmodified binary at runtime. The profiles generated by ibprof are written using the Open Trace Format (OTF) [16]. ibprof's operations may be grouped into three phases: (1) initializing the profiling environment, (2) accumulating communication statistics, and (3) cleaning up the profiling environment. We use MPI's PMPI interface to intercept the relevant MPI calls in order to trigger phases (1) and (3) in our profiler as shown in Fig. 4. Phase (2) is performed within our Peruse callback function that is defined in the profiler.

Communication statistics are accumulated in dynamically allocated traffic counter arrays. Two arrays of traffic counters are maintained in ibprof for each active port on the system: one array for bytes sent and one for bytes received. The index of each array element corresponds to a target LID and the value of the element corresponds to the amount data sent to or received from the respective target LID, depending on the array. Separate arrays are needed for send and receive counters because the amount of data transmitted during an RDMA operation is not always recorded at both peers.

By default, all communication traffic will be recorded by the profiler. However, environment variables can be used to limit the measurement to specific MPI operations, and manual instrumentation can be done to limit measurements to specific regions in the application source code. The user can also manually instrument the application source code to dump counters on demand, with each dump representing a "code region" in the profile.

D. Post-processing Profile Conversion

In a post-processing step, we combine the information from our communication profiles with network configuration information to write network visualization information for Boxfish. Network configuration information is obtained using the ibdiagnet utility on each subnet used by the application.

After parsing the profile and ibdiagnet output files, a connected graph is created to represent the nodes (switches, compute nodes, etc.) and links in the network. Afterwards, a series of breadth-first searches is performed, with the network adapters as the root for the BFS, to determine the vertical position of each switch in the fat-tree topology. Nodes are positioned horizontally in a manner that reduces the number of intersecting links. Weights are then added to the link ends by tracing the path of application traffic across the network using the forwarding tables provided by ibdiagnet. The weight at each link end represents the number of bytes transferred on the link by the node connected to that end.

V. Evaluation

The overhead of our profiling is evaluated in two parts: first is the memory consumption of our profiler and second is the runtime cost of profiling communication.

A. Memory Consumption

The memory consumption of ibprof is fixed for each process on a given system regardless of the number of processes in the application. Traffic counter arrays are the only dynamically allocated memory and are determined by the number of active ports in the system. Other variables are allocated on the stack and have insignificant memory footprints. For a given system, the memory consumed by the traffic counters is given by the following formula:

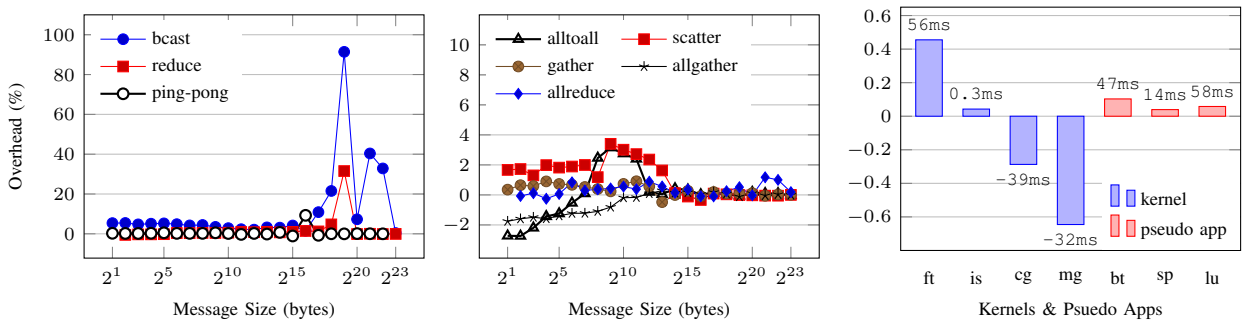
$$mem\_usage = num\_active\_ports \times counter\_size \times 2 \quad (1)$$

where *num_active_ports* is the number of active ports across the system and *counter_size* is the size of a single counter element (i.e., 8 bytes). The product of these two values are doubled since separate counter arrays are used for sent and recieved data. Therefore, on a system with 1024 nodes and one active port per node, ibprof would consume $1024 \times 8 \times 2 = 16384$ bytes of memory for storing traffic counters.

B. Profiling Overhead

1) Experiment Setup: We conducted experiments on an 44-node, 2-switch InfiniBand-based cluster to measure runtime overhead. The MPI library used was Open MPI v1.6.5, which was compiled with the --enable-peruse flag and included our Peruse extension described in Section IV-B. We had exclusive access to the system for these experiments.

The runtime impact of our profiling library was assessed using benchmarks from the Intel MPI Benchmark (IMB) suite [17] and the NAS Parallel Benchmark (NPB) suite [18]. Runs were conducted on 32 nodes, with the exception of IMB ping-ping and NPB pseudo apps, which occupied 2 and 36 nodes respectively. A one-to-one process-to-node mapping was used with each process bound to core 0 on its respective node. IMB benchmarks used 100,000 iterations for small message sizes and automatically decreases this value for larger messages in order to attain meaningful results in a timely manner. All other parameters were kept at their default values. We ran 100 profiled trials and 100 un-profiled trials for each benchmark. Message sizes for IMB experiments ranged from 2 bytes to 8 MB and NPB experiments used the class C problem size.



(a) **Intel MPI Bencharmark** (b) **NAS Parallel Benchmarks**

Fig. 5. **Runtime overhead of communication profiling.** Subfigure (5a) shows the percentage increase in communication latency for the various IMB benchmarks. These results are separated into two charts for increased readability. Subfigure (5b) shows the increase in runtime of NPB kernels and pseudo applications. The height of each bar represents the overhead as a percentage of the communication runtime and the bars annotation states the actual change in communication runtime.

2) Increase in Communication Latency: Figure 5 shows the results of our experiments. These graphs represent the increase in communication latency caused by our profiler and does not reflect the time for dumping profiles. Results were averaged across all 100 pairs of runs with standard errors $<1\%$ in all cases. The average overhead was 11.6%, 3.4%, and 1.3% for `MPI_Bcast`, `MPI_Reduce`, and `MPI_Scatter`, respectively, over all message sizes while other IMB benchmarks averaged below 1% overhead. Similarly, all NPB benchmarks averaged below 1%, with the communication-bound FT benchmark reporting the highest value of 0.46%.

The averaged runtime differences were in the order of microseconds for the IMB benchmarks and milliseconds for the NPB benchmark. Because such small differences could be attributed to jitters in the system, we ran similar experiments at different times over several days for verification. Similar trends were observed in the results with some runs occasionally reporting negative overheads and all overheads remaining negligible except for spikes in the `MPI_Bcast` and `MPI_Reduce` results for the message sizes shown. We confirmed that the spike in the `MPI_Bcast` can be attributed to Open MPI switching from the send/receive semantics to RDMA pipeline protocol when the message size surpasses 256 KB. We ran a set of `MPI_Bcast` trials with the RDMA pipeline size limit changed from 256 KB to 1 MB and the pipeline send length changed from 1 MB to 4 MB. As expected, we observed additional spikes for messages between 1 MB and 4 MB in size. Further research is being planned to ascertain the cause of this phenomenon.

3) Increase in Application Runtime: The total increase in application runtime when `ibprof` is used is equal to the increase in communication latency plus the time taken to write profiles. On our system, the time taken for a complete profile dump was less than 1 seconds, irrespective of the application or communication pattern. This time is dependent on the IO subsystem's performance, which is beyond the scope of this work.

VI. Case Study

In this Section, we showcase the usability of our profiling approach and analysis toolchain. We analyze the execution of samplesort, a popular sorting algorithm for parallel systems, and we also compare the performance of different MPI library versions. Experiments were conducted on TSUBAME2.5, which utilizes two independent IB subnets and each compute node has a link to each subnet.

A. Visualizing Traffic Patterns and Contention in Samplesort

Samplesort, as described in [19], is a sorting algorithm for distributed memory environments. The main idea behind the algorithm is to find a set splitters to partition the input keys into p buckets corresponding to p processes in such a way that every element in the i^{th} bucket is less than or equal to each of the elements in the $(i+1)^{th}$ bucket. Because splitters are selected randomly, the resulting bucket sizes may be uneven. This could result in communication and computation imbalances when keys are shuffled and sorted, respectively.

For our experiment, we used the samplesort code presented in [19] [1]. We executed samplesort with 128 MPI processes, using a 1:1 process-to-node mapping. Each process started with 1 GB of unsorted integers, randomly generated with a uniform distribution. The same random number seed was used in all cases. Fig. 6 shows a typical process-centric visualization of samplesort's main communication routines over 128 nodes using Paraver. We are unable to extract any network performance insights from this and other similar visualizations that are generated using PMPI-based instrumentation tools.

1) Performance Analysis using our `ibprof` Profiler and our Boxfish Module: We profiled an execution of samplesort using `ibprof` and visualized the network traffic in our Boxfish fat tree module. Segments of the code were manually instrumented to enable the identification of the code block where the all-to-all key exchange is conducted in order to perform a meaningful analysis. Fig. 7 shows the network traffic generated by the main communication routines of samplesort. This section of the profile reflects the traffic generated by the segment of the program highlighted in Fig. 6.

The red links that are visible in area C of Fig. 7 represent links that were carrying the most traffic during the communication block of the code. By exploring the visualization

[1] Source code: http://users.ices.utexas.edu/~hari/talks/hyksort.html

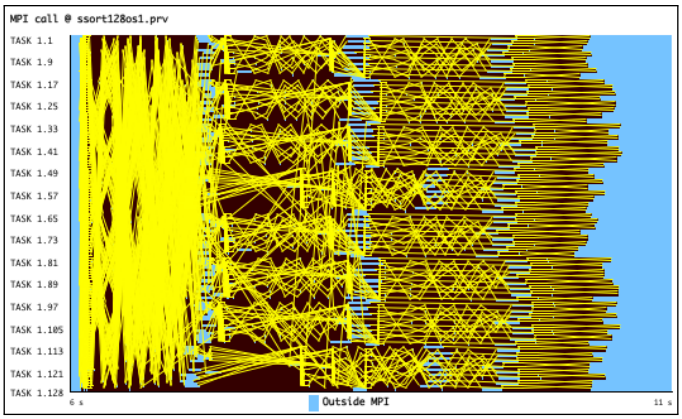


Fig. 6. **Paraver visualization** — Horizontal multi-coloured rows represent the execution of different samplesort processes and the yellow lines drawn diagonally across the rows indicate communications among the processes.

and applying various filters, we were able to clearly identify highly-used links. Such links are potential points of network bottlenecks. Link contention becomes even more likely when these links are simultaneously being used by traffic from other applications, a situation that is very probably for large, multi-user HPC-systems such as TSUBAME2.5.

2) Optimization Efforts: Our first effort to reduce this contention was to use the data in our profile to identify which of our processes were sending/receiving the most traffic over these links. We moved these process to other nodes with the intention of having its traffic sent over a different path through the network. However, this failed to achieve any performance gains because moving these processes created new hotspots in other areas of the network. The issue was further complicated by the fact that the routing between nodes is not identical on both subnets. Certain nodes (storage, management, etc.) are connected to a single subnet, which results in non-identical port forwarding tables on each subnets. We are able to confirm these routing differences visually with our Boxfish module by noting the difference in traffic patterns across both subnets, i.e., comparing the upper and lower halves of Fig. 7. Our visualization with Boxfish was also able to show that the elimination of hotspots in one subnet by re-mapping processes sometimes caused the creation of even more hotspots in the other subnet.

We then attempted to use a different set of nodes for the experiment. This is shown in Fig. 8. The traffic visualization of this new run reveals that the nodes are more tightly clustered and the peak application traffic per link has been reduced by over 20%. Additionally, the traffic in this run does not use level 3 switches in either of the subnets, thereby reducing the communication latency. We compared the runtimes over both node sets and found that the all-to-all performs better on the new node set 94 times out of 155 trials. The average performance gain was 5.08%. Due to time restraints and resource constraints, additional runs could not be performed and the network was shared by other users during our experiments.

B. Visualizing Traffic Patterns Inside the MPI Library

A comparison of benchmark runtimes using different versions of Open MPI on TSUBAME2.5 uncovered the fact the

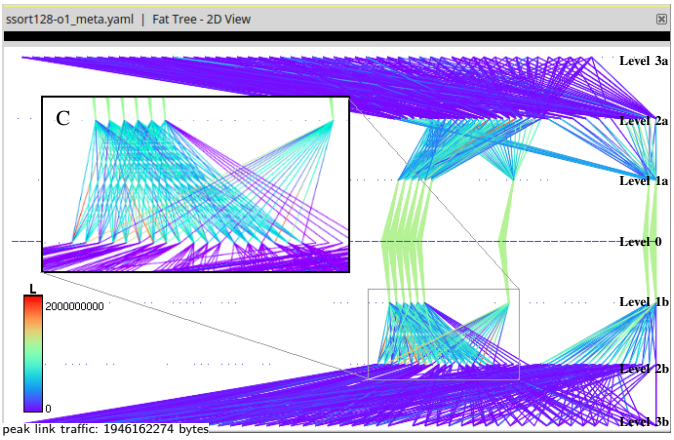


Fig. 7. **Boxfish visualization of samplesort's communication traffic on TSUBAME 2.5's network** — For increased readability, network links that were not used by our application traffic are not shown. Level 0 represents all the non-switch nodes in the network, which includes all compute, management, and storage nodes. Levels 1a, 2a, and 3a contain all the switches of the first subnet and levels 1b, 2b, and 3b contain the switches of the second subnet. Lines drawn between levels represent a subset of the network links in the system. Segment C highlights one area of high application traffic.

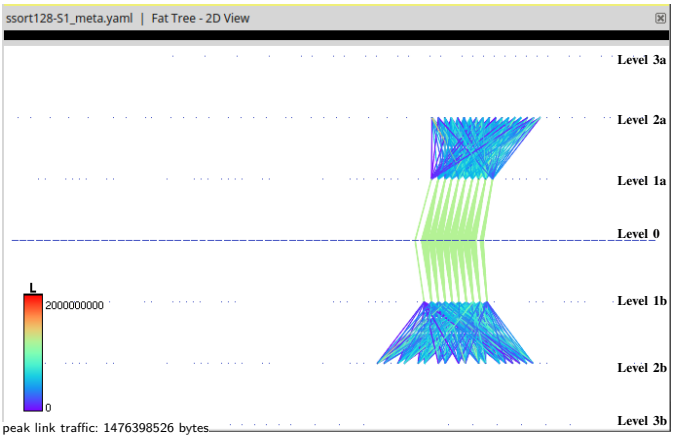
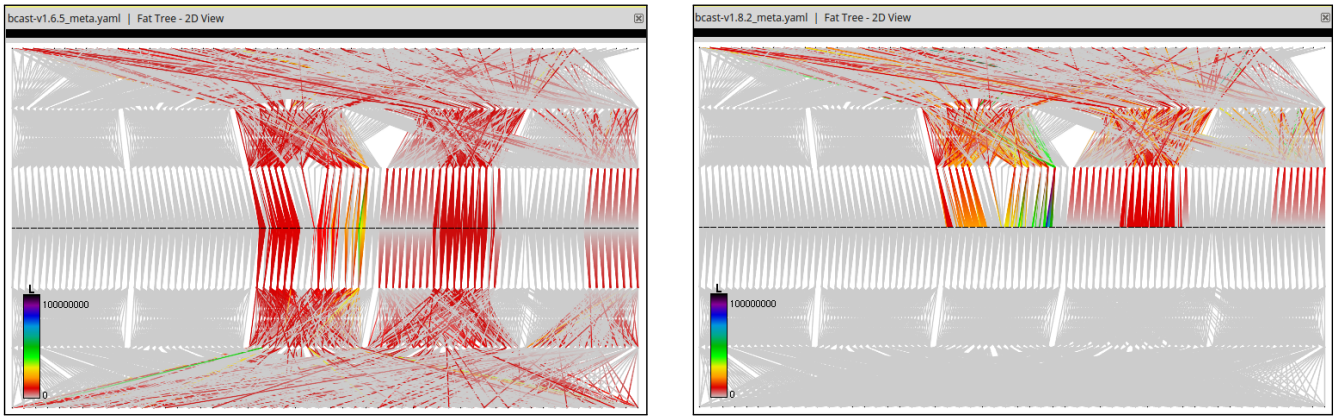


Fig. 8. **Boxfish visualization of samplesort's communication traffic after using a different node set** — The nodes in the new set are in close physical proximity to each other. Notice that no traffic is going to level 3 switches, and hence those links are not visible.

latest version of Open MPI, v1.8.2, was performing significantly slower than v1.6.5. Runtime traces indicated that the slowdowns were due to increased communication time, but the traces could not identify the root cause. IMB PingPong test reported a 40-50% reduction in communication throughput for message sizes over 12 kB when using Open MPI v1.8.2. To view the network communication pattern, we used `ibprof` to profile runs of an `MPI_Bcast` microbenchmark using the different versions of Open MPI with the default system parameters. As illustrated in Fig. 9, v1.6.5 was distributing traffic evenly across both subnets while v1.8.2 used only a single subnet. This accounted for the approximately 50% drop in throughput reported by the IMB microbenchmark.

Investigations revealed that v1.8.2 introduces a new MCA paramter **`btl_openib_ignore_locality`** whose default value of "0" caused the library to not use all interfaces. While setting the value to "1" corrected this, more fine-grained profiling using `ibprof` reported that the library was using interfaces



(a) **Using Open MPI v1.6.5**

(b) **Using Open MPI 1.8.2**

Fig. 9. **Comparison of using different Open MPI versions on TSUBAME2.5** — The traffic pattern of MPI_Bcast on 512 nodes is shown. All network links are shown except for those connected to management or storage nodes. The same link-value range is used for both vizualizations.

in a round-robin manner, i.e., each successive MPI operation uses a different interface from its preceding operation. This is less efficient at load balancing traffic than v1.6.5, which splits messages across both interfaces within MPI operations instead of between operations.

C. Discussion

The use of performance visualization tools like Paraver can aid in the identification of various process-based bottlenecks, e.g. late senders, however, they do not reveal any of the library's internal routines nor do they expose activity over network links when used in isolation. We've demonstrated that by conducting performance analysis at a lower level, our profiling utility and hardware-centric visualization give us the ability to locate potential bottlenecks in a portable and non-intrusive way.

Our toolchain can be used in to isolate application-specific traffic in a shared environment and gives application users greater visibility into the communication components of their codes and how they are affected by system configurations. Moreover, the use of our toolchain extends the analysis capabilities of MPI library developers, network designers, and systems adminsitrators as they are presented with new insights into the performance within their environments.

VII. Related Work

Analyzing the performance inside MPI routines has been the subject of numerous research studies. Kunkel et al. [20] and Miguel-Alonso et al. [21] did work on tracing the MPI point-to-point operations that make up collective communication but could not show the communication performance in the network layer. Studies done on MPI_T [22], [23] were also unable to identify any method of tracking network link traffic for MPI messages like we do in this work.

In their article on the performance analysis of simulations on IBM Blue Gene/P (BG/P) systems, Landge et al. [24] demonstrated how BoxFish can be used to visualize network traffic generated during an application's execution. They acquired communication traffic measurements by recording the changes in network port counters during each MPI operation by using BG/P system tools. This method is restrictive since it prevents any other application from running on the system during the experiments. Furthermore, this and other Boxfish related works, such as [25] and [7], dealt with only tori topologies and not fat-tree or any other network topology. Our approach records application-specific performance metrics within the MPI library and can be use on shared nodes and shared networks. Moreover, our extension to Boxfish enables the visualization of any 2D network topologies, not just fat-trees.

Prominent performance visualizing tools such as Vampir [26], Paraver [27], Scalasca [28], and PerfExplorer [29] present data in a process-centric manner, which means that metrics are presented relative to individual processes. None of these tools consider the physical network links when graphically representing process locations, resulting in network bottlenecks at the link level going undetected.

INTAP-MPI [30] was created as a network topology-aware performance analysis tool for MPI applications on InfiniBand networks. They use a similar concept to ours in that network-related metrics are collected at a low level within the MPI library and analyzed with respect to the network configuration. However, INTAP-MPI does not offer any visualization of the physical network. It's visualizations go only as far to report the number of hops taken by MPI messages and the volume of messages transferred among nodes/processes.

Unlike INTAP-MPI, our hardware-centric visualization shows how network links and paths can become overloaded by process placement, choice of routing algorithms, and changes in the message passing semantics, etc. Our approach also excels by tracking traffic per process and per node-port, allowing us to accurately and precisely show how an application using multiple subnets can exhibit a different communication pattern on each subnet. We are able to identify the cumulative impact this has on the application, as shown in Section VI-A2. INTAP-MPI and all other process-centric tools would have combined the measurements of both subnets and ultimately obscured the true nature of the situation.

VIII. Conclusion

MPI libraries, by design, prevent the user from easily seeing the correlation between communication events in the application and data transmission over network links. Our profiler, `ibprof`, successfully penetrates this abstraction while using the Peruse interface and lightweight enhancements to Open MPI. `ibprof` is non-intrusive and incurs, on average, negligible increase in communication latency with NPB benchmarks, 11.6% for the `MPI_Bcast` collective, and less than 5% for other MPI collectives. Our Boxfish `Fat Tree` module enables the hardware-centric visualization of our profiles, exposing the network-level performance of MPI applications running on any 2D network topology such as fat-trees.

Our case studies have proven that the hardware-centric, low-level manner in which we analyze performance offers insight into the performance of MPI operations in ways that PMPI-based, process-centric tools cannot. Our approach can also be used to support other areas of network research that require an efficient way to track and visualize the performance of applications over large-scale networks. Such research areas include the improvement of MPI collectives and the optimization of routing algorithms to increase the efficiency of current and future HPC systems.

References

[1] "Top500 List," http://www.top500.org/, July 2015.

[2] C. Bell, D. Bonachea, R. Nishtala, and K. Yelick, "Optimizing Bandwidth Limited Problems Using One-Sided Communication and Overlap," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, 2006.

[3] MPI Forum, "`MPI`: A Message-Passing Interface Standard," http://www.mpi-forum.org/, Mar 2014.

[4] K. A. Brown, J. Domke, and S. Matsuoka, "Tracing Data Movements Within MPI Collectives," ser. EuroMPI/ASIA '14, 2014, Poster presented at the 21st European MPI Users' Group Meeting.

[5] "Open MPI: Open Source High Performance Computing," http://www.open-mpi.org/, accessed: 2014-06-29.

[6] "MPI PERUSE: An MPI Extension for Revealing Unexposed Implementation Information version 2.0," http://hcl.ucd.ie/wiki/images/e/ea/Current_peruse_spec.pdf, Mar 2006, accessed: 2014-06-29.

[7] K. E. Isaacs, A. G. Landge, T. Gamblin, P.-T. Bremer, V. Pascucci, and B. Hamann, "Exploring Performance Data with Boxfish," in *Proceedings of SC Companion: High Performance Computing, Networking, Storage and Analysis (SCC), 2012*, 2012.

[8] A. J. Peña, R. G. C. Carvalho, J. Dinan, P. Balaji, R. Thakur, and W. Gropp, "Analysis of Topology-dependent MPI Performance on Gemini Networks," in *Proceedings of the 20th European MPI Users' Group Meeting*, ser. EuroMPI '13, 2013, pp. 61–66.

[9] E. Zahavi, "Fat-Trees Routing and Node Ordering Providing Contention Free Traffic for MPI Global Collectives," in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, May 2011, pp. 761–770.

[10] A. Faraj, S. Kumar, B. Smith, A. Mamidala, J. Gunnels, and P. Heidelberger, "MPI Collective Communications on the Blue Gene/P Supercomputer: Algorithms and Optimizations," in *Proceedings of the 23rd International Conference on Supercomputing*, ser. ICS '09, 2009, pp. 489–490.

[11] InfiniBand Trade Association, http://www.infinibandta.org/, Jun 2014.

[12] C. Leiserson, "Fat-trees: Universal Networks for Hardware-efficient Supercomputing," *IEEE Transactions on Computers*, vol. C-34, pp. 892–901, Oct 1985.

[13] S. Öhring, M. Ibel, S. Das, and M. Kumar, "On Generalized Fat Trees," in *Proceedings of the 9th International Parallel Processing Symposium, 1995*, Apr 1995, pp. 37–44.

[14] J. Dongarra, "Visit to the National University for Defense Technology Changsha, China," http://www.netlib.org/utk/people/JackDongarra/PAPERS/tianhe-2-dongarra-report.pdf, Jun 2013, accessed: 2014-07-05.

[15] R. Keller, G. Bosilca, G. Fagg, M. Resch, and J. J. Dongarra, "Implementation and Usage of the PERUSE-Interface in Open MPI," in *Proceedings of Euro PVM/MPI*, 2006.

[16] A. Knüpfer, R. Brendel, H. Brunst, H. Mix, and W. E. Nagel, "Introducing the Open Trace Format (OTF)," in *Proceedings of the 6th International Conference on Computational Science - Volume Part II*, ser. ICCS'06, 2006, pp. 526–533.

[17] Intel Corportation, "Intel MPI Benchmarks 4.0 Update 2," https://software.intel.com/en-us/articles/intel-mpi-benchmarks, accessed: 2015-01-26.

[18] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga, "The NAS Parallel Benchmarks," NASA Ames Research Center, Tech. Rep. RNR-94-007, Mar 1994.

[19] H. Sundar, D. Malhotra, and G. Biros, "HykSort: A New Variant of Hypercube Quicksort on Distributed Memory Architectures," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ser. ICS '13, 2013.

[20] J. Kunkel, Y. Tsujita, O. Mordvinova, and T. Ludwig, "Tracing Internal Communication in MPI and MPI-I/O," in *Proceedings of International Conference on Parallel and Distributed Computing, Applications and Technologies, 2009*, 2009.

[21] J. Miguel-Alonso, J. Navaridas, and F. Ridruejo, "Interconnection Network Simulation Using Traces of MPI Applications," *International Journal of Parallel Programming*, vol. 37, no. 2, pp. 153–174, 2009.

[22] T. Islam, K. Mohror, and M. Schulz, "Exploring the capabilities of the new mpi_t interface," in *Proceedings of the 21st European MPI Users' Group Meeting*, ser. EuroMPI/ASIA '14, 2014.

[23] R. Rajachandrasekar, J. Perkins, K. Hamidouche, M. Arnold, and D. K. Panda, "Understanding the memory-utilization of mpi libraries: Challenges and designs in implementing the mpi_t interface," in *Proceedings of the 21st European MPI Users' Group Meeting*, ser. EuroMPI/ASIA '14, 2014.

[24] A. Landge, J. Levine, A. Bhatele, K. Isaacs, T. Gamblin, M. Schulz, S. Langer, P.-T. Bremer, and V. Pascucci, "Visualizing Network Traffic to Understand the Performance of Massively Parallel Simulations," *IEEE Transaction on Visualization and Computer Graphics*, pp. 2467–2476, Dec 2012.

[25] A. Bhatele, T. Gamblin, K. Isaacs, B. Gunney, M. Schulz, P. Bremer, and B. Hamann, "Novel Views of Performance Data to Analyze Large-scale Adaptive Applications," in *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC), 2012*, Nov 2012, pp. 1–11.

[26] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. Müller, and W. Nagel, "The Vampir Performance Analysis Tool-Set," in *Tools for High Performance Computing*, M. Resch, R. Keller, V. Himmler, B. Krammer, and A. Schulz, Eds. Springer Berlin Heidelberg, 2008, pp. 139–155.

[27] Barcelona Supercomputing Center, "Paraver: a flexible performance analysis tool," http://www.bsc.es/computer-sciences/performance-tools/paraver/general-overview, 2014, accessed: 2014-06-29.

[28] M. Geimer, F. Wolf, B. J. N. Wylie, E. brahm, D. Becker, and B. Mohr, "The Scalasca performance toolset architecture," *Concurrency and Computation: Practice and Experience*, vol. 22, pp. 702–719, 2010.

[29] K. A. Huck and A. D. Malony, "Perfexplorer: A performance data mining framework for large-scale parallel computing," in *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, ser. SC '05, 2005.

[30] H. Subramoni, J. Vienne, and D. Panda, "A scalable infiniband network topology-aware performance analysis tool for mpi," in *Euro-Par 2012: Parallel Processing Workshops*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, vol. 7640, pp. 439–450.