T2R2 東京工業大学リサーチリポジトリ
Tokyo Tech Research Repository

論文 / 著書情報
Article / Book Information

題目(和文)	HPCシステムのデータ移送による資源競合
Title(English)	Resource Contention due to Data Movement on HPC Systems
著者(和文)	BROWNKEVIN
Author(English)	Kevin A. Brown
出典(和文)	学位:博士(学術), 学位授与機関:東京工業大学, 報告番号:甲第11009号, 授与年月日:2018年9月20日, 学位の種別:課程博士, 審査員:松岡 聡,増原 英彦,遠藤 敏夫,脇田 建,額田 彰
Citation(English)	Degree:Doctor (Academic), Conferring organization: Tokyo Institute of Technology, Report number:甲第11009号, Conferred date:2018/9/20, Degree Type:Course doctor, Examiner:,,,,
学位種別(和文)	博士論文
Type(English)	Doctoral Thesis

Powered by T2R2 (Tokyo Institute Research Repository)

Resource Contention due to Data Movement on HPC Systems

Kevin A. Brown

A dissertation submitted in partial fulfillment of the requirements for the degree

Doctor of Philosophy
in
Mathematical and Computing Sciences

Tokyo Institute of Technology
2018

Advisor: Professor Satoshi MATSUOKA

Committee Members:

Professor Hidehiko MASUHARA
Professor Toshio ENDO
Associate Professor Ken WAKITA
Associate Professor Akira NUKADA

Acknowledgment

I must first declare my appreciation to my advisor, Prof. Satoshi Matsuoka, for his indelible contribution to my growth as a student and for orchestrating the start of my life as a researcher.

I also gratefully and humbly acknowledge the efforts of my direct collaborators, Dr. Jens Domke of Tokyo Tech, Drs. Abhinav Bhatele and Nikhil Jain of Lawrence Livermore National Laboratory, and Prof. Martin Schulz of Technische Universität München. Their contributions have been instrumental in developing the research topics presented in this document.

To my friends, colleagues, and drinkers who have supported me in this journey: I say "Thanks!" while I continue to search for words that better capture my gratitude. Please know that I could not have gotten here without you.

Portions of this work have been published in Brown et al. [2015] and Brown et al. [2018] with the help of my collaborators throughout my time as a graduate student.

Dedicated to my mom,
who has taught me the magnificence of kindness and selflessness.

Kevin A. Brown
August, 2018

Abstract

Large-scale high-performance systems (HPC), or *supercomputers*, are composed of hundreds/thousands of nodes that are interconnected using advanced network topologies. One such topology is the *fat-tree* topology, which provides high-bandwidth and low-latency communications. However, network communication has become a major source of performance bottleneck, even on fat-tree networks, due to significant growth in the sizes of HPC workloads. Ensuring good communication performance remains challenging because of the complexities of the communication operations, architectures, and workloads, et al.

This work introduces an approach to conducting performance analysis of MPI applications running on large-scale networks by efficiently and portably exposing metrics from within MPI the layer via the Peruse interface. The solution features the creation of the `ibprof` profiler to record InfiniBand traffic data and a generalized performance visualization toolset to accurate combine application-specific information with hardware configuration information of fat-tree networks. `ibprof` incurs an average communication overhead of 2.06% for the `MPI_Alltoall` microbenchmark, 7.63% for the `MPI_Bcast` microbenchmark, and 0.02% for the NPB FT application benchmark. Furthermore, case studies demonstrate that the visualization toolset can identify communication anomalies in applications as well as communication libraries and can also guide performance optimization strategies.

Another important contribution of this work is a characterization of the interference between the MPI and I/O traffic on fat-tree networks. The characterization study was done using the packet-level-accurate network simulations and highlights the effects of varying the message size, communication interval, and job size. The results show that MPI traffic is more sensitive to interference than I/O traffic in all cases, with light I/O traffic exhibiting a 1.9× slowdown due to heavy MPI interference and light MPI traffic exhibiting a 7.6× slowdown due to heavy I/O traffic. Furthermore, a significant feature of I/O traffic patterns was identified and labeled as the *I/O-congestion threshold.* This threshold is the I/O interval (or frequency of sending I/O requests) where the self-congestion from I/O is more detrimental than the contention due to MPI interference. This threshold varies with the I/O request size and the job size. The network topology-specific feature and the characteristics of the interference are used to demonstrate that, independently, specific node allocation and I/O server placement strategies can completely avoid the I/O-MPI interference. Additionally, throttling I/O traffic based on its I/O-congestion threshold can reduce the slowdown of MPI performance by approximately 200% while incurring only an 18% performance penalty.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter briefly describes the current state of large-scale, high-performance computing systems (HPC), their workloads, and the architecture of the infrastructures that support inter-node communication on these systems. The challenges of ensuring good communication performance are discussed as well as the problems with conventional performance analysis approaches. Additionally, the contributions of this work are stated in the context of the previously mentioned performance analysis challenges. Finally, an outline of the rest of this document is provided at the end of this chapter.

1.1 Background

1.1.1 HPC Infrastructure

HPC systems are rapidly growing in physical size, with staggering increases in node counts over recent years. The Sunway TaihuLight supercomputer, which ranks first in the November 2017 Top500 list of fastest supercomputers, comprises of over 40,000 nodes [TOP500.org, 2018]. In fact, each of the top 10 systems have over 1000 computer nodes, and the nodes are interconnected with advance network topologies. These networks include fat-tree [Leiserson, 1985; Petrini and Vanneschi, 1997], 3D-torus [Adiga et al., 2005], 5D-torus [Chen et al., 2011], and 6D-torus [Ajima et al., 2009] topologies. It is common practice for systems to be composed of components from different vendors. For example, the networking elements may be manufactured by Mellanox Technologies Ltd., and the processing element may be produced by Intel Corporation. In these types of situations, each component will have its own performance characteristics and optimizations that are not easily matched to components from other vendors without targeted efforts [Mellanox Technologies Ltd., 2017]. The overall performance of the system is therefore determined by how applications utilize each component and the interactions between all components across the system [Mellanox Technologies Ltd., 2004].

The nodes on HPC systems are assigned different roles based on the systems administrators configurations. These include:

- Management services: authentication nodes, job scheduling nodes

- Compute services: compute nodes where the application code is executed
- Storage services: storage nodes that provide temporary and/or persistent storage of data

1.1.2 HPC Workloads

The phrase *traditional HPC workload* is often used to refer to scientific applications that require large amounts of computing resources beyond what is available in other environments. Supercomputers were most notably used in areas such as fluid dynamics, probabilistic analysis, nuclear test simulations, and other similar classes of complex scientific applications [Cray Research, Inc., 1985]. In recent years, HPC systems have been successfully applied to other classes of problems that can benefit from the large storage capacities and high-bandwidth networks of the systems; data-intensive problems in the areas of Big Data and Deep Learning are now also being solved by supercomputers [Brown et al., 2017]. The types of problems involve moving large amounts of data across the system, and the performance of these problems are more dependent on the speed of data movements instead of the speed of calculations. Hence, modern HPC workloads contain a mix of both compute-intensive and data-intensive applications.

Deploying HPC systems require large amounts of technical and financial resources, therefore, ensuring high overall utilization is one method of justifying the investments made into these systems. The largest supercomputers in Tokyo Institute of Technology (Tokyo Tech), Lawrence Livermore National Laboratory (LLNL), and the Barcelona Supercomputer Center (BSC) are used by researchers from all departments within the respective organizations. This means that the workloads that run on these systems involve a mix of application from different problem domains, communication patterns, and performance characteristics.

1.1.3 Communication Architecture and Libraries

In this work, inter-node communication refers to off-node data movement over the network infrastructure. Data movement of this nature is often necessitated by the data partitioning scheme of the computational algorithms of large-scale simulations and the checkpointing strategies of long-running applications, among other things. Inter-node communication is split into two categories: (1) inter-process communication, where data is sent between application processes that are running on multiple compute nodes; and (2) network I/O traffic, where data is sent between compute nodes and the I/O servers during I/O operations. All references to inter-process communication and I/O traffic will refer to inter-process communication and network I/O traffic, respectively.

Inter-process Communication

Inter-process communication is required for many large-scale applications to broadcast initial parameters, share updated data, combine intermediate results, etc. Performing inter-node communication over advance network topologies is non-trivial and has a significant impact on the overall application throughput. This is especially true for communication-bound applications [Bell et al., 2006], whose performance is more sensitive to communication latency than computation throughput. Because of this, optimizing communication within these large-scale applications is a standard approach in performance tuning on these massive systems. A wide variety of work has been done on designing the network hardware to support communication routines in the application , building communication libraries that take advantage of specialize hardware facilities [Huang et al., 2006], and hiding communication latency in the application [Subramoni et al., 2017].

The Message Passing Interface [MPI Forum, 2018] has become the cornerstone of inter-process communications in many HPC applications. MPI defines a standard, portable interface for performing communication among processes in an HPC application. Communication libraries that implement this interface are referred to as *MPI libraries*. The performance of message passing 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 used to undertake the operation [Peña et al., 2013; Zahavi, 2011; Faraj et al., 2009]. 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. Open MPI [The Open MPI Project, 2014] and MPICH2 [Argonne National Laboratory, 2] are two widely used MPI libraries.

I/O Communication

Parallel file systems are an important part of the HPC machines as they provide highly-available, persistent storage which can be accessed from all compute nodes. To provide parallel access to data from compute nodes, or I/O clients, a single file is often split into *chunks* and spread across multiple individual storage devices of a single file system. High-performance parallel file systems, such as Lustre [lus, 2018], one of the most widely used parallel file systems in HPC, have reached storage capacities of up to 72 PiB and over 1 TiB/s aggregate read throughput [vi4, 2018]. Such storage infrastructures are used for several purposes, e.g. last-level checkpointing, Big Data workflows, and visualization data for scientific applications [Latham et al., 2012; Kurth et al., 2017; Luu et al., 2015; Oral et al., 2014].

The data payload in an I/O operation is often split into multiple requests, which can be distributed, pipelined, and/or processed concurrently. The maximum request size is configurable by the administrator and defaults to 4MB in the latest version of Lustre (version 2.10.4). Notable I/O optimization studies have shown how requests can cause congestion on the storage servers as well as on the network, specifically the links that directly interconnect I/O servers [Liu et al., 2013].

1.1.4 Communication Performance Analysis

Communication analysis tools can be placed into two broad categories: *application-* or *process-centric* and *hardware-centric*. Application-centric tools provide performance information specific to application, such as how much data is communicated between two processes of an application. Hardware-centric tools report information in the context of hardware resources, such as how much data traversed a particular network link. All tools fit in one of the two categories, but very few tools span both categories.

VampirTrace/Vampir [Knüpfer et al., 2008], Scalasca [Geimer et al., 2010] and other widely used performance analysis tools rely on the PMPI interface since it allows the capture of timing information without making any changes to the application's source code. Both Vampir and Scalasca provide graphical representations of performance data for parallel applications in a process-centric manner. That is, processes are individually represented and performance metrics are overlaid onto each process. In the case of Vampir, inter-process communication is characterized by lines drawn between the participating processes. This approach to visualization has been helpful in locating computational imbalance, high cache miss rate and opportunities for serial optimizations [TU Dresden ZIH, 2014].

Unlike the previously mentioned analysis tools, Boxfish [Isaacs et al., 2012] supports both process- and hardware-centric approaches for presenting data. It is able to accurately visualize the physical topology of the network and overlay performance data onto the nodes and links within the network. Performance data can be attained from multiple sources: the process (e.g., throughput), the host (e.g., CPU load), the network (eg, port counters), etc.

1.2 Motivation

1.2.1 Network Resource Sharing and Limitations

All resources in a computer system have clearly defined specifications and technological limitations, supercomputer resources included. This is valid for compute resources as well as network resources. For example, even the fastest supercomputers each has a maximum communication bandwidth and a minimum transfer latency. Figure 1.1 shows the performance specifications of the TSUBAME series of supercomputer.

System specifications inform the theoretical upper limit on how much resources can be accessed by applications, and hence, a limit on the application's performance.. Various communication- and I/O-bound operations, such as reading input files, checkpointing, and updating neural network gradients, are always limited by the network capacity. Even the most well-optimized programs such as the FT benchmark from the NAS parallel benchmark suite [Wong et al., 1999] experience these performance limitations. The processes within these application must compete among each other for available bandwidth, and these processes must also compete with other applications that are running on the network. The performance of these applications is further impeded by the reduced available network capacity when other applications share the network links. That is, the application's communication performance is heavily dependent on the *available* network bandwidth and not on the *specified/rated* network bandwidth [Jain et al.,

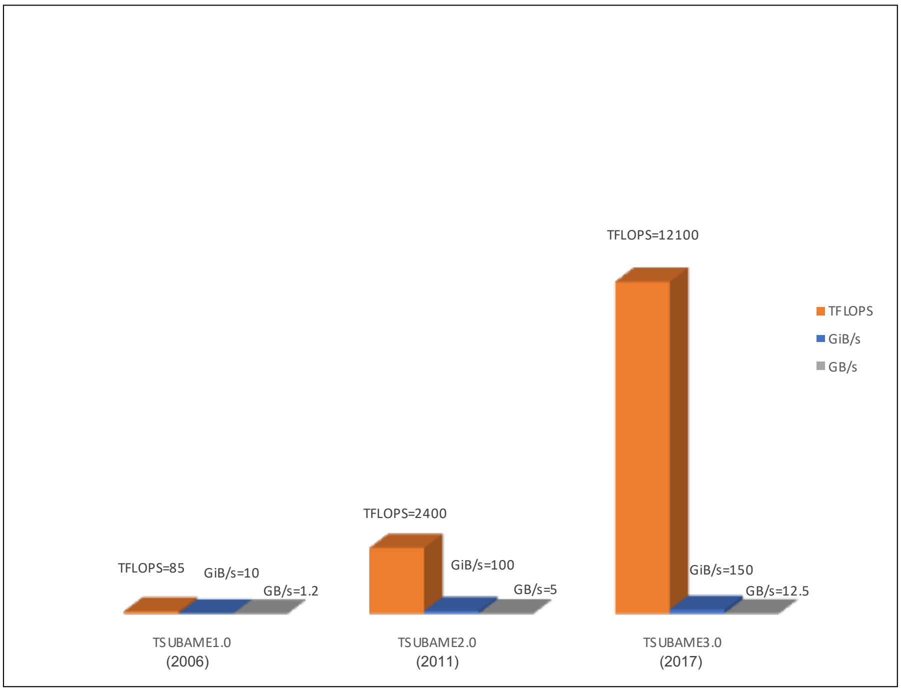


Figure 1.1: The performance trend for the Tokyo Institute of Technology's TSUBAME series of supercomputers – Compute performance in teraflops (TFLOPS), aggregate I/O bandwidth in GiB/s, and network bandwidth in GB/s.

2016]. However, the source of the contention can be difficult to troubleshoot or avoid because of the system architecture and lack of available analysis tools.

1.2.2 Performance Abstractions and Compartmentalized Analysis

The complexity of HPC system components and their interconnectivity requires different levels of abstractions in order to promote programability and usability. However, these abstractions come at the cost of performance visibility. Insightful performance analysis and characterization should capture the salient features that affect the performance of an application, features that may be buried under layers of abstractions and spread across multiple components of the system. The meaningful analysis of complex HPC applications running on advanced network technologies is, therefore, an intricate and challenging task. It requires user-friendly, non-intrusive, low-interference tools that can penetrate multiple layers of software and hardware abstraction in order to accurately represent the application's performance across the network [Kunkel et al., 2018; Lammel et al., 2016; Agelastos et al., 2014]. Otherwise, overly-intrusive analysis efforts will yield diminishing returns, and oversimplification of the environment will stymie optimization efforts.

Interprocess Communication

MPI libraries are guided to minimize communication overhead by ensure that sensitive communication tasks on the application's critical path are not unnecessarily delayed [MPI Forum, 2018]. However, it is common to observe congestion due to MPI traffic within an application which can degrade performance [Bhatele et al., 2014; Vishwanath et al., 2011]. This often occurs in collective operations where multiple processes of the same application are sending traffic simultaneously.

Performance tuning efforts are complicated by the MPI libraries' abstraction of 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 the reverse.

The MPI standard's performance revealing extensions PMPI and MPI_T operate above the MPI layer are are too strict in their management of internal performance variables, respectively. 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. They currently cannot support effective analysis of the MPI library's operations over the network hardware, down to the level of the physical links.

I/O Traffic

Studies of I/O contention and its resulting performance degradation have investigated I/O bottlenecks in both the backend storage devices as well as network interconnects [Boito et al., 2018; Carns et al., 2009; Lang et al., 2009; Luu et al., 2015; Vishwanath et al., 2011; Xie et al., 2012]. However, these studies have done fairly coarse-grained assessments of I/O workloads and have been focused mainly on issues relating to the backend storage. For example, the size of the Lustre I/O requests is guided by the chunk size of the backend storage since the alignment of requests and chunk sizes results in better storage performance. However, with the advent new burst buffer architectures and in-memory storage solutions, the understanding of I/O performance over the network is becoming increasingly important. We need to study network I/O performance in a similar manner as with MPI performance, with careful consideration of request sizes, frequencies, and traffic distributions.

1.2.3 Mixed Workloads Impact on Performance

Communication performance analysis on shared systems involves the added consideration of the performance impact caused by interference from other applications that compete for network resources. Production HPC workloads involve significant amounts of data exchange among compute nodes as well as between compute nodes and parallel file systems. Past research has shown that such data movement on HPC systems is a source of significant performance degradation for many applications [Bhatele et al., 2014; Vishwanath et al., 2011; Solomonik et al., 2011]. Furthermore off-node data movement is particularly susceptible to interference on systems where the network infrastructure is shared by all running jobs [Bhatele et al., 2013; Yang et al., 2016]. However, with the exception of Mubarak et al. [2017a], these studies have focused on network traffic generated by one of the two major sources of off-node data movement: MPI or I/O. For the dragonfly topology [Kim et al., 2009], Mubarak et al. showed that packets generated by MPI communication of one job can experience over 4000× increase in maximum latency due to interference from I/O traffic of another job. This can result in notable performance degradation for the MPI job.

1.3 Problem Statement

The problem statements are detailed here:

MPI is a black box and existing tools fail to show the cause of communication bottlenecks

The hardware abstraction provided by MPI hinders in-depth performance analysis of MPI communication operations. The PMPI interface, which most performance analysis tools are based on, treats the MPI library as a black box. Vampir and similar PMPI-based tool are not able to provide any insight into the internal routines used by MPI or the impact of the network layer on application performance, as shown in Appendix A. The performance of collective operations, which can involve multiple, simultaneously communicating processes, over theses complex networks is even more difficult to analyze. The combination of the network routing algorithm, the placement of processes across the network, and the sequence used to exchange data within collectives can introduce network bottlenecks that are impossible to detect from the application layer.

Analysis needs to be performed for a lower level

PMPI-based tools have confined analysis to the application layer in an effort to remain portable across the different operating environments. However, with the performance of the network layer and the internal routines of collectives having such a big impact on communication performance, there is a need to conduct analysis at an even lower level; both the hardware and MPI layers need to be explored. MPI_T and PMPI can reveal information from within the MPI library, but neither of them supports any method of revealing the network hardware events involved in an MPI operation at a sufficiently fine granularity. Any optimizing strategies relying on these facilities will ignore the impact of the network layer activities on an application's performance.

There is little or no support for analysis on most network topologies The network infrastructure that is used by an application will affect the job's performance. However, the tools available to extracts and analyze network network specific information from the application context are insufficient. For example, even though Boxfish is able to collate and present data from multiple domains, there is no existing facility for easily extracting network information in a generic and portable manner. Existing work with Boxfish relies in using network tools for the IBM Blue Gene/P (BG/P) system to provide information on an application's traffic pattern. Additionally, the only topology that is currently supported by Boxfish is the 3D torus topology of the BG/P systems. 2D mesh, dragonfly, and fat-tree network topologies cannot be analysed in Boxfish.

The degree of application interference on shared systems unknown High system utilization by running multiple concurrent jobs improves overall system throughput but can also degrade the performance of individual jobs. Comprehensive performance analysis covers both understanding the performance feature of an application in isolation as well as understanding its performance in a production environment. For shared environments, the interference characteristics of applications are also important features to be noted. Specifically, how the job's performance is impacted by interference from other applications and the degree of interference caused to other applications by the job in question. The traffic patterns of different workloads can define different interference patterns, and therefore, interference studies for production systems cannot be conducted by considering the workloads in isolation of each other.

Interference mitigation strategies do not account for cross-workload interference Various optimizations that are implemented to both MPI and I/O jobs to improve performance by avoiding interference. These optimizations include job placement, I/O server placement, and throttling. Unfortunately, these optimizations tuned for interference from a similar class of traffic. For example, MPI optimizations are designed to avoid interference from other MPI application and not I/O traffic. The interference mitigation strategies need to be designed with emphasis placed on multiple types of interfering workloads because this represents the actual environment in which the application will be executed.

1.3.1 Research Questions

1. How can fine-grained metrics be exposed across different abstraction layers to support combined process-centric and hardware-centric analysis of communication performance?

2. How can low-level communication metrics be efficiently analyzed at a high (application) level to provide new insights into communication performance?

3. What are the most important aspects of an application's behavior that affect performance and cross-workload contention on fat-tree networks?

4. How should interference mitigation strategies be tuned to address cross-workload interference on fat-tree networks?

1.4 Contributions

In addressing the problems outlined in the previous subsection, the contributions of this work are:

1. **A novel method of efficiently exposing and capturing low-level performance metrics in Open MPI**
   To overcome the limitation of PMPI-based analysis strategies, which treat MPI as a black box, the PERUSE utility in Open MPI is extended in order to reveal InfiniBand network events from within MPI operations. A lightweight, non-intrusive profiling tool is created to record, aggregate, and report the sending of InfiniBand traffic as exposed by this new PERUSE event. Exposing the low level behavior within the MPI library allows for performance analysis that is more comprehensive and more effective at identifying the cause of network communication bottlenecks.

2. **Creating a generalized visualization solution for fat-tree topologies**
   A process of automatically detecting the topology of fat-tree networks and combining performance data with network elements is presented. This process is supported by a newly created Boxfish visualization module that can flexibly and interactively illustrate the performance over the network. This solution supports accurate visualization of any 2-dimensional network topology, including the multi-rail fat-tree topology used by TSUBAME2.5.

3. **Characterize the effects of interference between the MPI traffic and the I/O traffic on fat-tree networks**
   The slowdown due interference between various MPI traffic patterns and I/O traffic patterns is quantified and qualified, detailing how interference is affected by message sizes, communication intervals, and system allocations. This characterization shows that MPI traffic is more sensitive to interference than I/O traffic in all cases shown in this study. Furthermore, a congestion threshold is identified for I/O traffic where the self-congestion from I/O is more detrimental than the contention due to MPI interference.

4. **Guide cross-workload interference mitigation strategies**
   The efficacy of several communication optimization strategies on reducing the impact of interference between MPI and I/O traffic are evaluated. These strategies are targeted at mitigating this interference on fat-tree networks by taking advantage of the topology-specific feature and the characteristics of the interference. Placement strategies are shown to completely avoid I/O-MPI interference, and the throttling of I/O traffic significantly reduces MPI slowdown while incurring only an 18% performance penalty for I/O.

1.5 Outline of Thesis

The rest of the document is organized as follows:

Chapter 2 reviews related research and explain how this work differs from the others.

Chapter 3 presents the tools and technique for exposing network-level communication events using the PERUSE interface in Open MPI on InfiniBand-based networks. The implementation and evaluation details of the `ibprof` profiler are also provided in this chapter.

Chapter 4 outlines the process of visualizing application traffic on fat-tree networks using low-level performance data, such as those reported in `ibprof` profiles. Case studies are also presented that demonstrate the usability of the visualization solution and validate this performance analysis approach.

Chapter 5 provides a characterization of the interference between I/O and MPI traffic on fat-tree networks. This characterization considers the impact of various aspects on interference, including the frequency and size of messages/requests as well as the size of the jobs.

Chapter 6 discusses how different performance optimization strategies that can leverage architectural features of the fat-tree topology to mitigate cross-workload interference.

Chapter 7 provides a summary of the entire work, discusses the implications and limitations of the findings, and briefly mentions future directions of the research.

Chapter 2

Literature Review

This chapter contains an overview of other studies that are closely related to this work. Firstly, the state-of-the-practice of performance analysis is discussed, with attention being given to performance tools that are designed for the measurement and visualization of performance data for network communication. Since performance analysis component of this work deals with MPI communication over HPC networks, only tools in this area are considered and only the the features that relate to the scope of this work are discussed in any detail.

Secondly, a review of performance characterization efforts for MPI and I/O traffic on HPC systems is presented. This review highlights characterization of traffic across production workloads on large-scale supercomputers. These results were attained from real-world application, proxy-applications, or representative benchmarks and kernels.

Thirdly, studies related to the analysis of cross-workload network interference are surveyed and their findings presented. Interference from other jobs running on shared systems has been studied broadly, and a summary of these works are reported along with their limitations.

Finally, methods of optimizing traffic performance by mitigating interference are discussed. Consideration is given to both interference between jobs of a similar workload, such as interference between two I/O jobs, as well as interference between jobs of different workloads, such as and MPI job and I/O job sharing the same network.

2.1 Performance Analysis Tools

2.1.1 Performance Measurement

Using the MPI Standard's Facilities

PMPI and MPI Tools (MPI_T) are the two facilities defined by the MPI standard for users to assess the performance of MPI operations [MPI Forum, 2018]. With PMPI, MPI functions that are prefixed by `MPI_` can also be accessed by changing the prefix to `PMPI_`. Users can then intercept calls to `MPI_` prefixed functions by defining their own custom `MPI_` functions and then calling the corresponding `PMPI_` function within the their custom function to execute the MPI operation. This will provide access to the parameters sent to the MPI library and allows the user to generate timing information for the MPI operation. All parameters that can be accessed by using PMPI already exist in the application space, therefore nothing from within the MPI library is exposed. Popular tools such as VampirTrace [1], Score-P [2], and Extrae [3] use the PMPI interface as the primary mechanism to trace and profile MPI performance. While these tools employ other mechanisms to gather performance data, the PMPI interference is main utility for observing message passing operations.

MPI_T was designed to expose configuration and performance variables from inside the MPI library to the application space. Each implementation decides which variables they chose to support while the standard specifies the API for querying and accessing the supported variables. MPI_T was incorporated into version 3.0 of the MPI standard, which was released in September 2012. While the MPI standard doesn't specify what performance variable an MPI library should expose, it dictates strict management procedures for internal performance variables. This restricts the flexibility for the facility and, at the point of writing, no MPI_T implementation provided the resources necessary to track point-to-point MPI traffic across links in the network.

Using Network Tools

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 traffic data, etc [Bhatele et al., 2012a; Landge et al., 2012; Bhatia et al., 2018]. For example, Mellanox Technologies provides the `ibutils2` package [Mellanox Technologies, Inc., 2018] with utilities for querying performance metrics on their InfiniBand networks.

In their article on the performance of simulations on BG/P systems, Landge et al. [2012] analyzed the network traffic generated during application execution. In addition to the network performance characteristics of MPI collectives and point-to-point operations, they presented several case studies in which they investigate the performance characteristics of a layer and plasma interaction simulator running on different BG/P systems. They measured the network performance of MPI applications by using BG/P system tools to capture port counters before and after executing the MPI operation. The change in counter values reflected the application-generated traffic since there were no other applications running when the experiment was conducted. The network metrics retrieved by these toolsets are application-agnostic and are unable to distinguish between the traffic of different applications. This method of analysis is very inflexible since it requires that no other jobs run on the network while the analysis is being done in order to accurately correlate Using this approach is restrictive and results in sub-optimal system utilization when other applications are prevented from using available resources.

[1] https://tu-dresden.de/zih/forschung/projekte/vampirtrace?set_language=en

[2] https://www.vi-hps.org/projects/score-p/

[3] https://tools.bsc.es/extrae

Using Manual Instrumentation

Other attempts to expose low-level communication performance have focused on the MPI libraries themselves. Miguel-Alonso et al. [2009] described a process of using MPI application traces to conduct simulation-based performance analysis. In order to track the internal point-to-point communication of collectives with their traces, they modified the source code of the MPICH MPI library to expose point-to-point function calls that are used within the collectives. They were then able to use the PMPI interface to trace these internal calls. In a similar manner, Kunkel et al. [2009] also modified the source code of MPICH to gain access to function calls within collectives via the PMPI interface. However, neither efforts were able to identify which network links were being used by the MPI operations since they did not record any hardware information.

This work uses the PERUSE interface [per, 2006], which was developed for the explicit purpose of exposing the operations inside the MPI layer. PERUSE allows for internal performance metrics to be easily managed and safely exposed on production systems. By design, this interface can be extended to identify the ports and networks links used for each communication operation. The solution described in Chapter 3 will show how this interface is used in conjunction with a specially designed profiler to record point-to-point information for all network ports being used by the application.

2.1.2 Performance Visualization

Debugging and tuning MPI applications can be very challenging because of, among other things, the complexity introduced by inter-process communication and inter-process dependency. There are several visual analysis tools and frameworks that are commonly used to assist application developers. Vampir [Knüpfer et al., 2008], Scalasca [Geimer et al., 2010], Intel Trace Analyser [Intel Corportation, b] and PerfExplorer [Huck and Malony, 2005] are a few of these utilities that provide graphical representations of performance data. Metrics are presented relative to individual processes and the relationships among processes are typically determined by their intercommunication patterns.

Aside from the aforementioned tools, a wide range of work has been done on visualizing the performance of distributed applications on large-scale systems. Muelder et al. [2009] discussed the drawback of many visualization approaches. For example, they assert that the Gantt chart approach of plotting process rank versus time for distributed applications is ineffective when the number of processes exceed the number of available pixels on screen. They propose visualizing distributed communication at a very high level and then allowing the user to "drill down" into regions of interest.

Other efforts have attempted to capture a spatial relationship between communicating processes in an effort to improve communication performance. For example, Bhatia et al. [2005] showed how the visualization of the virtual topology of processes in a distributed application can provide insight into the cause of communication bottlenecks. They define the virtual topology based on the communication patterns between neighboring processes.

None of the aforementioned works in this section propose any tool that incorporates both network and application performance metrics in performance analysis. Boxfish [Isaacs et al., 2012] is one of the few tools that accomplishes this. BoxFish is a python-based performance analysis tool that is capable of visually representing the physical nodes and links in a network. It allows the simultaneous reporting of performance from the application, hardware, and communication domains. Bhatele et al. [2012a] and Isaacs et al. [2012] showed how Boxfish can be used to effectively explore performance data on large-scale systems by including the hardware domain. By using BoxFish in addition to other analysis tools, Bhatele et. al gained a 22% performance improvement for an adaptive mesh refinement library. Their work, however, dealt solely with the 3D torus network of the IBM Blue Gene/P (BG/P) system [IBM Blue Gene Team, 2008]. Bhatele et al. [2016] and Bhatia et al. [2018] presented DragonView and TreeScope for visualizing performance on dragonfly and fat-tree networks, respectively. Both are web-based visualization solutions with analysis features that are similar those of the Boxfish tool and limitation specific to how they implement their solutions.

INTAP-MPI [Subramoni et al., 2013] was created as a network topology-aware performance analysis tool for MPI applications on InfiniBand networks. This work involved the collection of network-related metrics at a low level via the MPI library and analyzing these metrics with respect to the network configuration. However, INTAP-MPI does not offer any visualization of the physical network. Its visualizations go only as far as to report the number of hops taken by MPI messages and the volume of messages transferred among nodes/processes.

In summary, not many visualization tools are able to accurately capture the physical topology of the system's network. For those tools that do, such as Boxfish, DragonView and TreeScope, they are specific to a single topology and have limitations due to how they visualize their networks.

2.2 Performance Characterization

2.2.1 MPI

León et al. [2016] and Jain et al. [2017] studied the performance of different fat-tree topologies to determine the trade-off between the performance and cost of different topology configurations. Between both works, performance characterization was done for the traffic patterns of 10 application and proxy-applications that are representative of production HPC workloads. They detailed the type of MPI operations and their message sizes for certain operations. They also categorized the operations frequencies, communicator size, and message sizes. The works demonstrated that typical MPI workloads consisted of a wide variety and MPI operations and a myriad of traffic patterns with small messages below 512 B, medium messages below 64 KB, and large messages above 64 KB. The MPI operation type usually determines how traffic is distributed across the network, whether it is a point-to-point operation involving two nodes or a type of collective

Table 2.1: Common HPC communication patterns [Yuan et al., 2013]

Pattern	Description
all-to-all	throughput for all-to-all pattern, correlate to bisection bandwidth
bisect	average throughput all bisect patterns,same as effective bisection bandwidth
2DNN	average throughput for all 2-dimensional nearest neighbor patterns with random 2D grid sizes
2DNNDIAG	average throughput for all 2-dimensional nearest neighbor with diagonals patterns with random 2D grid sizes
3DNN	average throughput for all 3-dimension nearest neighbor patterns with random 3D grid sizes
3DNNDIAG	average throughput for all 3-dimension nearest neighbor patterns with random 3D grid sizes
RANDOM50	average throughput for random patterns where the number of SD pairs is uniformly distributed between 1 to nprocs × 50
RANDN50	average throughput for random patterns with the same number (uniformly distributed from 1 to 50) of random destinations for each node
permutation	average throughput for all permutation patterns
shift	average throughput for all possible shift patterns

operation involving multiple nodes.

A characterization of common HPC communication patterns is described by Yuan et al. [2013] in their evaluation of a new network topology. These patterns are shown in Table 2.1. The *bisect* pattern was noted for being useful in assessing the effective bisection bandwidth of a topology.

2.2.2 I/O

Darshan [Carns et al., 2011] is a popular I/O characterization tool across HPC centers. It supports profiling of all application specific profiling and well as system-wide profiling [Carns et al., 2009]. Luu et al. [2015] used Darshan logs to survey the I/O activities on three supercomputers at two of the United States Department of Energy (DOE) national research laboratories. Years of application and system logs were analyzed to understand the overall behavior of I/O activity on the petascale systems: Mira, Edison, and Intrepid. While their study did not capture the activities of all I/O jobs on the system, the findings were quite insightful. The following are some of their findings for I/O activities on DOE systems, in no order of significance:

- Few applications ever use more than 10% of the system's processors or transfer more than a few gigabytes of data
- 75% of the applications never exceeded 1 GB/s, or approximately 1% of the

peak aggregate bandwidth

- The resource utilization is dominated by a small number of science applications

- The majority of jobs send less than 1 GB of data, but few jobs exceed 1 TB

- POSIX I/O is more widely used than parallel I/O libraries

- Metadata often exceed I/O cost due to small jobs, small requests, and bad I/O practices

- The majority of request sizes in major applications can be as small as 8 B

2.3 Interference Analysis

Communication performance degradation due to interference has been investigated from the aspect of intra-application interference as well as inter-application interference.

Bhatele et al. [2013] and Yang et al. [2016], among others, have demonstrated that inter-process communication is particularly susceptible to interference on systems where the network infrastructure is shared by all running jobs. Liu et al. [2013] has shown that heavy streams of I/O requests can cause congestion on the storage servers as well as on the network, specifically the links that directly interconnect I/O servers, and Xie et al. [2012] used sampling and statistical modeling to analyze the contention for I/O bandwidth. Each of these studies focused on either I/O vs. I/O interference or MPI vs. MPI interference.

Mubarak et al. [2017a] evaluated the interference between MPI and I/O traffic on dragonfly networks with burst buffer. The work used the CODES simulation toolkit to simulate a dragonfly network with 9,600 nodes, 25 groups, 96 routers/group. Of the 9,600 nodes, 300 are burst buffer nodes with R/W bandwidth of 5.7 GiB/s per burst buffer. Their results showed that packets generated by MPI communication of one job can experience over 4000× increase in maximum latency due to interference from I/O traffic of another job. This can result in notable performance degradation for the MPI job, especially collective operations. They also evaluated different burst buffer placements, job placements, and job sizes with the conclusion that: (i) confining I/O traffic to a chassis reduces the MPI interference but increases I/O time and (ii) sending I/O traffic to a random burst buffer has the best I/O performance but causes more interference for MPI traffic. Their work tested a single I/O traffic pattern and a single MPI traffic pattern. Additionally, the conclusions are only valid for the dragonfly topology and cannot be applied to I/O nor MPI traffic on fat-tree networks. The interference study in this dissertation considers a variety of traffic patterns and focuses on the fat-tree network topology.

2.4 Optimizations and Interference Mitigation

2.4.1 MPI Optimizations

Collective communication over complex networks is very sensitive to network performance due to the simultaneous cross-communication by multiple processes. Work has been done to match the semantics of MPI libraries to these complex networks, as well as to design network technologies to meet the needs of MPI operations. The designers of the IBM Blue Gene/P (BG/P) architecture described how both the MPI library and the network technology used in BG/P have been engineered to work together in [Faraj et al., 2009]. Peña et al. [2013] showed that a network's physical topology can affect the performance of MPI applications by mapping MPI processes to nodes based on the performance characteristics of the Blue Waters supercomputer's 3D torus Gemini interconnect. Bhatele et al. [2012b] went further by optimizing applications with collectives over sub-communicators using a tool called Rubik, which assists the user in mapping the virtual topology of MPI processes to processors in the physical topology for BG/P and Blue Gene/Q systems.

Zahavi [2011] used models and simulations to achieve contention-free routing for a single collective running on a fat-tree network by taking into consideration the process-to-node mapping and the communication pattern of the target collective. His work decomposed the collectives into the communication patterns they employ and grouped the collective based on these patterns. This information was then used to design a routing algorithm to match the communication pattern of the target collective.

While all of the aforementioned optimizations yielded improvements in MPI performance, the impact of the optimized MPI traffic on the performance of I/O traffic was not assessed nor discussed.

2.4.2 I/O Optimizations

Parallel I/O libraries like MPIO, ADIO, and HDF5 are used to improve I/O performance on large-scale systems [Latham et al., 2017; Lofstead et al., 2010]. The libraries often transparently re-orchestrate the application's I/O activities to better match the underlying storage system's capabilities. MPIO two-phase I/O is one such orchestration. The first phase, called the aggregation phase, is a set of aggregator processes collecting the I/O requests from the other processes. In the second phase, the aggregators perform the I/O operation on the storage devices. Two-phase I/O reduces contention by reducing the number of processes simultaneous accesses the same I/O servers.

Other techniques to avoid congestion include I/O prefetching, buffering, scheduling, and throttling [Lofstead et al., 2010; Liu et al., 2013; Qian et al., 2017]. Many studies have been conducted on implementing these techniques for real workloads. However, none of these studies have evaluated the effect of I/O congestion and mitigation techniques on non-I/O jobs that share the network.

Chapter 3

Measuring MPI Traffic

This chapter describes how analysis of low-level communication metrics can be conducted with Open MPI to show performance contentions in the network layer. It first highlights how the hardware layer abstraction that is provided by MPI makes it difficult to study application communication performance over the network hardware, especially for collective operations. Then it covers how low-level network metrics can be revealed using Open MPI's PERUSE utility. The development and evaluation of a lightweight profiler, called `ibprof`, to aggregate these metrics are also covered in this chapter.

3.1 Overview of the Problem

HPC communication libraries improve the programmability of HPC systems by providing a composite abstraction layer between the application and the network infrastructure. Communication libraries that implement the Message Passing Interface (MPI) are one such group of libraries. These libraries are based on programming models, in this case the MPI standard, that are designed for HPC workloads and therefore contain support and optimizations for the common HPC communication tasks, such as collective broadcast and reduction operations.

The abstractions created by these libraries will limit the amount of information that the application can access in the lower levels of the network stack [Buntinas and Gropp, 2005a,b]. For analysis purposes, this is not ideal since the network hardware is a potential location for significant communication bottlenecks. Performance metrics from the network hardware have been shown to provide insightful information when analyzing performance of large-scale applications [Bhatele et al., 2012a; Isaacs et al., 2012; Landge et al., 2012]. However, the lack of access to relevant information from the application layer has necessitated probing the network hardware for system-wide traffic data in order to conduct this analysis as described in Section 2.1.1.

3.2 Review of the Relevant Architectures

3.2.1 InfiniBand Fat-tree Network

One of the most widely used network technologies for HPC systems is InfiniBand (IB) [InfiniBand Trade Association, 2014], a low-latency, high-throughput, switched networking architecture. InfiniBand hardware provides full-duplex connectivity along with kernel-bypass and remote direct memory access (RDMA) facilities. The set of interconnected nodes (switches, compute nodes, I/O servers, etc.) in an InfiniBand 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.

Theoretically, fat-tree networks can provide congestion- and interference-free routing for pair-wise communication across the system when it provides full bisection bandwidth [Leiserson, 1985; Öhring et al., 1995; Petrini and Vanneschi, 1997]. However, the routing algorithm and communicating processes must be coordinated to prevent overlap of network path between all active communication pairs [Zahavi, 2011]. Such a case is impractical and generally not applicable to production workloads. In practice, congestion does occur on fully-provisioned, high-throughput fat-tree networks that service production workloads [Jain et al., 2017].

3.2.2 MPI Libraries

The MPI Standard has been undergoing developed by the MPI Forum for over 30 years since the first version of the standard was published in 1994. The MPI Forum, a consortium of groups from both academia and industry, stated that MPI "should establish a practical, portable, efficient, and flexible standard for message-passing" [MPI Forum, 2018, pg. 1]. Popular libraries that implement the MPI standard, such as Open MPI and MPICH, use modular or layered architectures that serve the abstract the internal components of the operations. The benefits of these approaches include improved code reuse while flexibly supporting a wide variety of HPC environments and technologies [Buntinas and Gropp, 2005a].

The cost of the modular and layered architectural designs in reduced access to performance metrics across layers. Since performance measurement tools require access to these metrics, reduced visibility of metrics across layers prevents tools from reporting a comprehensive view of communication behavior without significant effort and unwieldy procedures.

Open MPI

Open MPI [Gabriel et al., 2004] is one of the most widely used MPI libraries. The code of Open MPI is separated into three sections: OMPI, which provides the MPI standard's API and supporting logic for the API; the Open Run-Time Environment (ORTE), which provides support for various operating environments; and the Open Portable Access Layer (OPAL), which provides utility support for

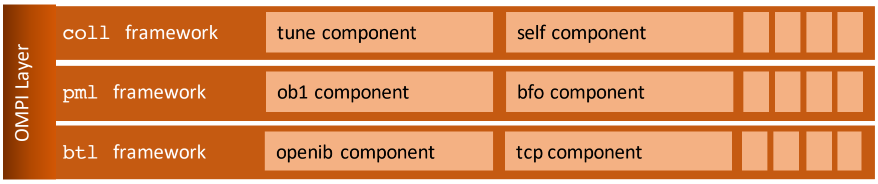


Figure 3.1: **Open MPI MCA frameworks** | A graphical representation of selected frameworks in the OMPI layer and their related components.

```c
typedef int (peruse_comm_callback_f)(peruse_event_handle event_h, MPI_Aint uid, peruse_comm_spec_t *spec, void *param);
```

Figure 3.2: **Peruse callback function prototype.**

OMPI and ORTE [The Open MPI Project, 2014]. 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 3.1 shows a visual description of Open MPI's OMPI code base.

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

3.2.3 The Peruse Extension for MPI

The Peruse interface was proposed as a performance revealing extension to the MPI standard that allows the tracking of internal events within an MPI library [per, 2006]. 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.

Figure 3.2 shows the prototype for Peruse callback functions. The `event_h` parameter holds the event handle while `uid` indicates the type of event that triggered the callback function. `spec` is a pointer to a `peruse_comm_spec_t` structure, which holds information related to the MPI operation that caused the invocation of the callback function. The information stored by the structure include a pointer to the message buffer, the message size, the peer process' rank, and other MPI operation-specific data. The callback function also takes a pointer to a variable in the application space, `param`, which the user specifies when registering the callback function.

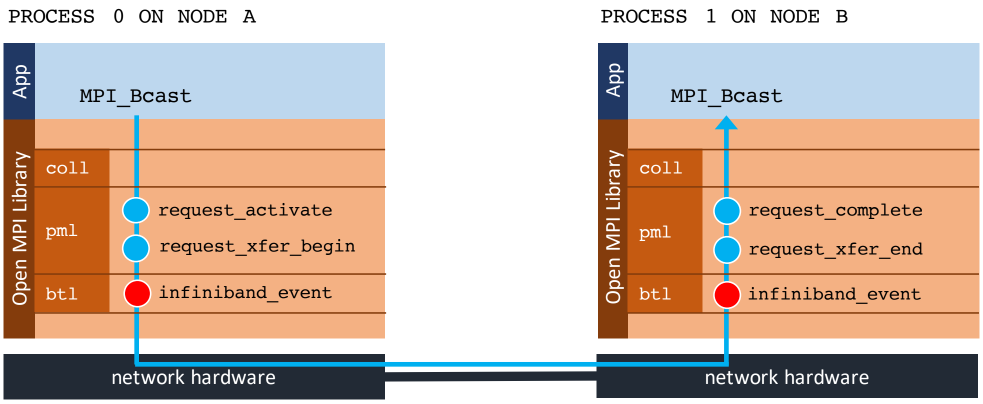


Figure 3.3: Peruse implementation in Open MPI. The blue line represents the communication path between MPI processes, and the blue dots show where Peruse events are placed along the communication path. The red-colored event indicates the location of the newly added PERUSE_OPENIB_SEND, which is described in Section 3.3.

Keller et. al [Keller et al., 2006] 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 Figure 3.1), and all events are tracked within pml framework components. Peruse was integrated in Open MPI through the use of C macros, which are inserted at points of interest within Open MPI's source code and triggers the callback function of the event it represents.

3.3 Exposing Low-level Traffic via PERUSE

Low-level application traffic metrics are required for showing application performance over network links. Since multiple application processes may share the same node, and therefore use the same network links, a distinction must be made between which network packets are sent by which processes in order to relate inter-process communication with its associated network link traffic. This association is made by recording the network ports that are involved in the communication and the type of traffic being communicated, i.e, user data or control messages. These low-level, application-specific metrics are exposed by creating a new event, named PERUSE_OPENIB_SEND, in the Peruse code base of Open MPI. This new event is labeled as infiniband_event in Figure 3.3.

The hardware-specific port information required by the PERUSE_OPENIB_SEND event is not available in the pml framework components where all other events were being tracked. The PERUSE_OPENIB_SEND event was added at a lower level in Open MPI's structure compared to other Peruse events, in the openib btl component (see Figure 3.1), where the port information is available. This event

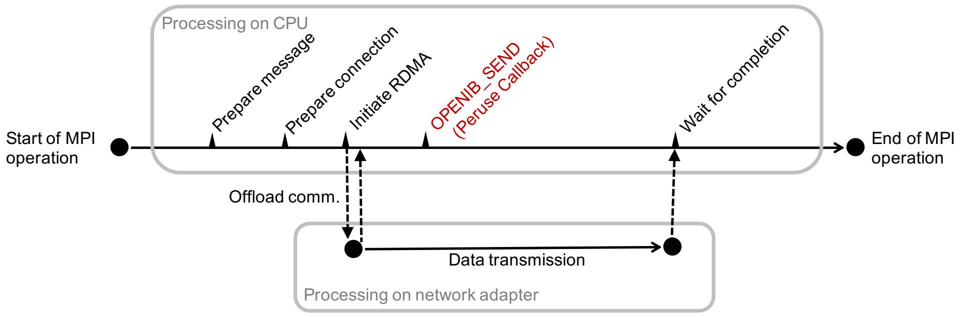


Figure 3.4: Overlapping Peruse InfiniBand callback with data transmission.

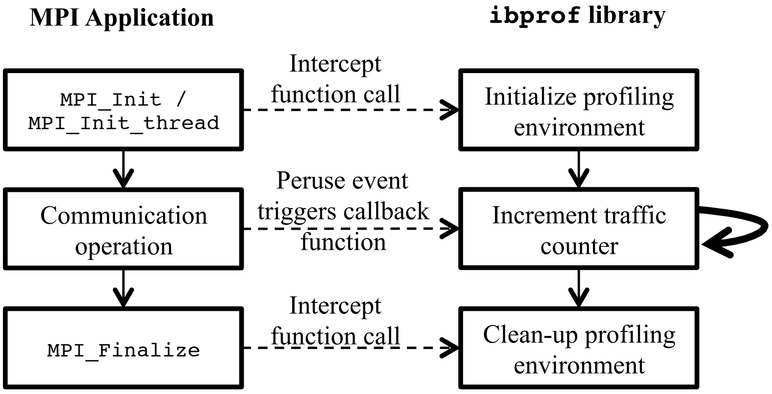


Figure 3.5: `ibprof` **overview** | For each of the listed operations performed by the application, we show the corresponding action performed by our profiler.

is triggered immediately after each call to `ibv_post_send`, which is the `ibverbs` command for offloading data transmission to the InfiniBand adapters and initiating the RDMA operation. This allows for overlapping the event processing with data communication, as illustrated from Figure 3.4, since `ibv_post_send` returns immediately after offloading the send request to the InfiniBand adapter. 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.

3.4 ibprof: InfiniBand Profiler

Since no existing tools are capable of utilizing the new `PERUSE_OPENIB_SEND` event, the `ibprof` profiling tool[1] was created to record InfiniBand traffic information reported by this new event.

3.4.1 Design and Implementation

Architecture

The `ibprof` profiler was implemented as a shared library that can be preloaded to an application's unmodified binary at runtime. The profiles generated by `ibprof`

[1] `ibprof` source code can be found at https://bitbucket.org/kevinabrown/ibprof.

are written using the Open Trace Format (OTF) [Knüpfer et al., 2006]. OTF was chosen because it is an established tool for collecting performance metrics on large-scale platforms and can easily support future extensions to our data collection methods. `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. MPI's PMPI interface is used to intercept the relevant MPI calls in order to trigger phases (1) and (3) in our profiler as shown in Figure 3.5. Phase (2) is performed within the PERUSE callback function that was defined in the profiler.

Recording Traffic Counters

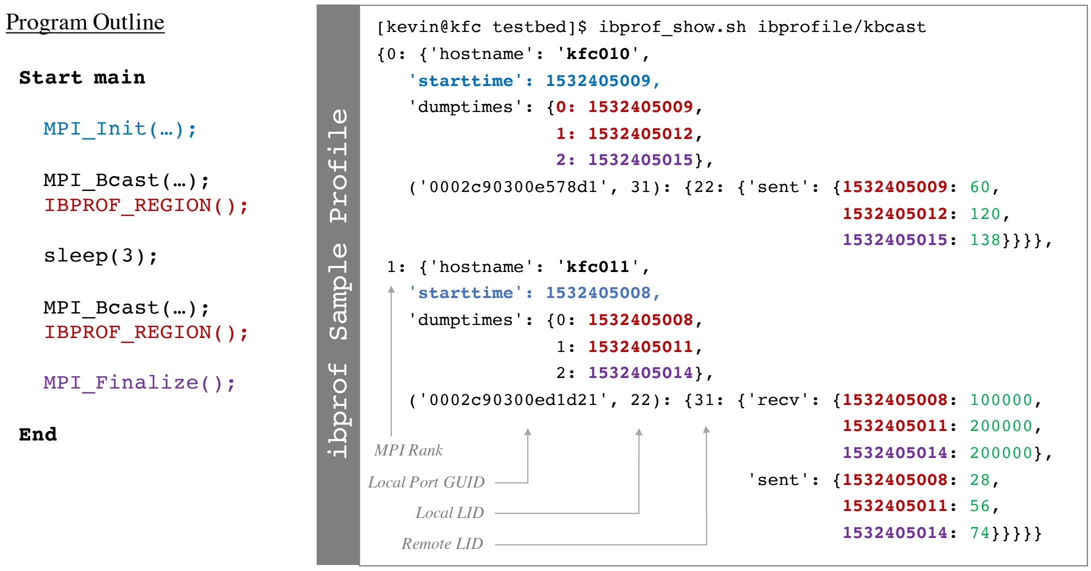
Communication statistics are accumulated in dynamically allocated traffic counter arrays, which are referenced in the callback function by the `param` argument (see Figure 3.2). Traffic counters a recorded for each active port used by the MPI library. During `ibprof`'s initialization phase, the callback function and counter arrays are registered with the new PERUSE `PERUSE_OPENIB_SEND` event. Host and port configuration information are also written to OTF definition files during the initialization phase. Non-zero traffic counters are written to OTF event files during the library's clean-up phase. 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 used for sent and received counter because the amount of data transmitted during an RDMA operation is not always recorded at both the sender and the receiver. For example, in a receiver-initiated RDMA operation the sender notifies the receiver when the data is ready to be transmitted. The receiver will then connect to the sender and pull the data remotely without the involvement of the sender. In this case, only the receiver will record the amount of user data that is sent during the operation since the sender does not track all the messages.

Managing the Profiler's Scope

By default, profiling is done for all communication. However, environment variables can be used to limit profiling to specific collectives. If profiling is limited to a specific collective, the `PERUSE_OPENIB_SEND` event is not activated when the profiler is initialized. Calls to the collectives are intercepted using the PMPI interface and the event is activated only during the execution of the specified collectives. Otherwise, the event is activated during the initialization phase and remains active for all operations until it is deactivated using manual instrumentation in the application source code or when `MPI_Finalize` is called. The technical details of the implementation is illustrated in Appendix B.2.

Manual instrumentation, available only when the library is joined to the application at link time, allows the user to specify the region(s) within the application that should be profiled. The user can also use this form of instrumentation to in-



```
Start main

  MPI_Init(…);

  MPI_Bcast(…);
  IBPROF_REGION();

  sleep(3);

  MPI_Bcast(…);
  IBPROF_REGION();

  MPI_Finalize();

End
```

```
[kevin@kfc testbed]$ ibprof_show.sh ibprofile/kbcast
{0: {'hostname': 'kfc010',
   'starttime': 1532405009,
   'dumptimes': {0: 1532405009,
                 1: 1532405012,
                 2: 1532405015},
   ('0002c90300e578d1', 31): {22: {'sent': {1532405009: 60,
                                            1532405012: 120,
                                            1532405015: 138}}}},
 1: {'hostname': 'kfc011',
   'starttime': 1532405008,
   'dumptimes': {0: 1532405008,
                 1: 1532405011,
                 2: 1532405014},
   ('0002c90300ed1d21', 22): {31: {'recv': {1532405008: 100000,
                                            1532405011: 200000,
                                            1532405014: 200000},
                                   'sent': {1532405008: 28,
                                            1532405011: 56,
                                            1532405014: 74}}}}}
```

Figure 3.6: `ibprof` **Sample Profile** | The blue, red, and purple lines in the program produce the corresponding timestamps in the profile. Traffic counter values are shown in green.

struct the library to write-out counters on demand, with each write representing a "code region" in the profile. A more meaningful analysis of the application's performance can be achieved by inspecting the profiles of individual code regions instead of viewing a single profile of the entire application run. Additional usage examples and information about the `ibprof` profiler can be found in Appendix B.

Figure 3.6 shows the outline of a simple program that makes two calls to `MPI_Bcast` as well the resulting `ibprof` profile generated by running the program on two nodes. Each `MPI_Bcast` call sends 100,000 bytes from rank 1 to rank 2. `IBPROF_REGION()` calls define the boundary between code region, and each call initiates an output of the total traffic sent/received since the program started (not since the last output was written). Only non-zero counters are outputted. The call to `MPI_Finalize` also indicates the end of a code region, hence, a third output of each non-zero counter is written. The profile show the start time and the times when counters are written on each rank as `starttime` and `dumptimes`, respectively. Timestamps are formatted using the Unix epoch time format. For each `dumptime`, the number of bytes sent/received for each port is also shown in the profile.

3.4.2 Evaluation

The overhead of profiling using **ibprof** is evaluated in two parts: the memory consumption of the profiler and the runtime cost of profiling communication. The usability of the profiles will be demonstrated in Chapter 4.

Experiment Methodology

The runtime overhead 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 3.3. No other jobs were running on the system during these experiments.

The runtime impact of `ibprof` was assessed using benchmarks from the Intel MPI Benchmark (IMB) suite [Intel Corportation, a] and the NAS Parallel Benchmark (NPB) suite [NASA Ames Research Center, 2012]. 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. One hundred (100) profiled trials and 100 un-profiled trials for each benchmark were ran. Message sizes for IMB experiments ranged from 2 bytes to 8 MB and NPB experiments used the class C problem size.

For all experiments described in this section, the minimum of the (average) time for each trial was taken as the resulting value since this would be the most reproducible result [Gropp and Lusk, 1999]. Preliminary tests showed that the chosen number of iterations and trials used in these experiments yield reasonably accurate and valid results within reasonable time.

Memory Consumption

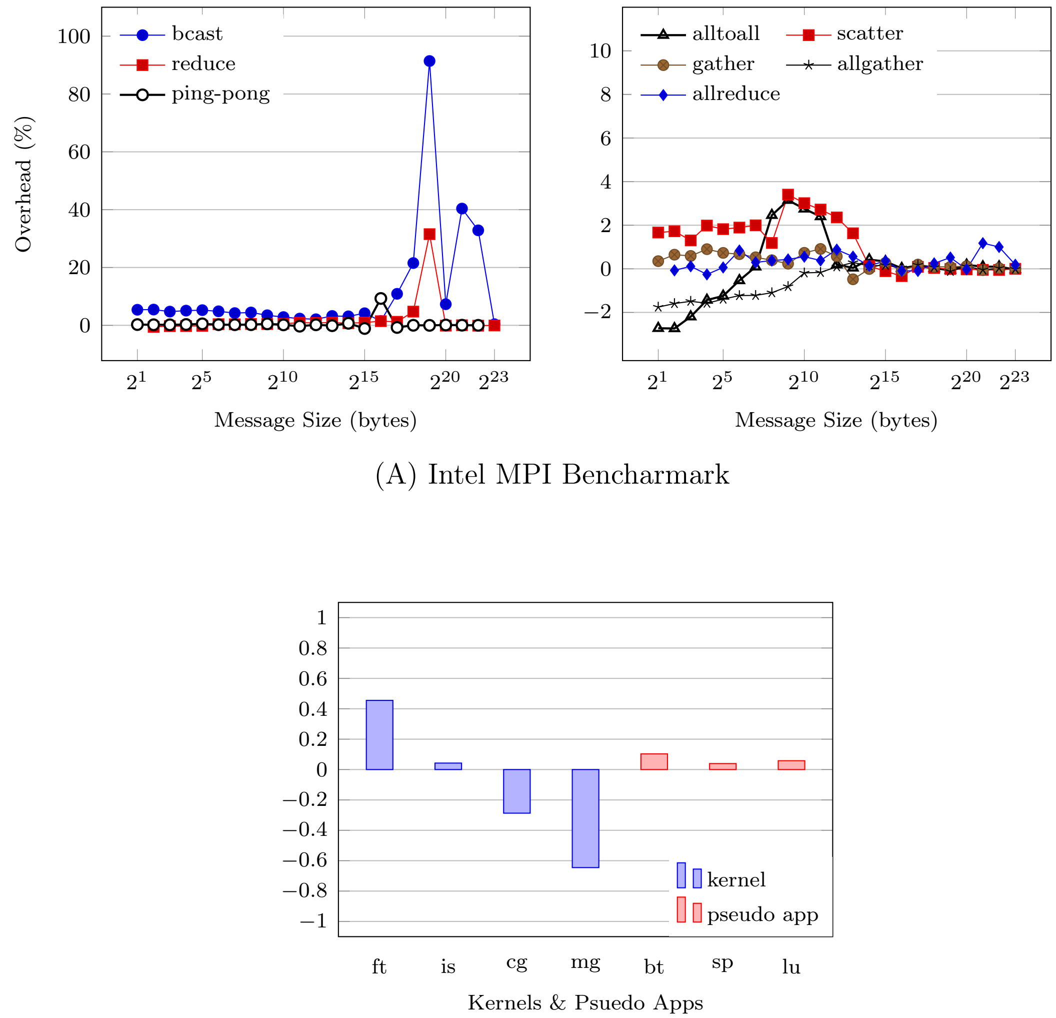
The memory consumption of `ibprof` is fixed for each process on a given system regardless of the number of processes in the application. Traffic counter arrays are the only dynamically allocated memory and are determined by the number of active ports in the system. Other variables are allocated on the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

$$mem\_usage = num\_active\_ports \times counter\_size \times 2 \tag{3.1}$$

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 received data. Therefore, on a system with 32 nodes and one active port per node, `ibprof` would consume $32 \times 8 \times 2 = 512$ bytes of memory for storing traffic counters for a single profile.

Increase in Communication Latency

Figure 3.7 shows the results of the communication overhead experiments. These results represent the increase in communication latency caused by `ibprof` profiler and does not reflect the time for dumping profiles. The values presented are the



(A) Intel MPI Bencharmark

(B) NAS Parallel Benchmarks

Figure 3.7: **Runtime overhead of communication profiling.** Subfigure (a) shows the percentage increase in communication latency for the various IMB benchmarks. These results are separated into two charts for increased readability. Subfigure (b) 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 bar's annotation states the actual change in communication runtime.

	MPI_Alltoall	MPI_Bcast
Profiling with output (ms)	97.579	16.759
Profiling only (ms)	84.343	3.506
Output time (ms)	13.236	13.253

Table 3.1: `ibprof` **output time** | This table shows the time taken to profile collective microbenchmarks with and without writing OTF files.

arithmetic mean of all 100 pairs of runs, with standard errors <1% in all cases. The mean 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/noise in the system, the experiments were repeated 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.

It was 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. An additional set of `MPI_Bcast` tests were done with the Open MPI's 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, the spikes were observed for messages between 1 MB and 4 MB in size.

Increase in Application Runtime

Table 3.1 shows the average output time for the collective microbenchmarks. 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 the experimental environment described in the **Experiment Methodology** section, the time taken for a complete profile dump was less than 1 seconds for each application benchmark. This time is dependent on the amount of data being written, which is equivalent to the amount of memory used for storing counters and is relatively small, and the performance of the I/O subsystem. Other studies have demonstrated that I/O workloads that are similar in size to these profiles incur negligible I/O overhead when writing to disk.

3.5 Discussion

ibprof successfully penetrates this abstraction while using the PERUSE interface and lightweight enhancements to Open MPI. This demonstrates that per-port, application-specific traffic information can be collected without detrimental degradation to application performance. While this information is not enough to identify all forms of communication anomalies, it is sufficient to expose certain forms of performance features that no other tool can expose. For example, ibprof measures the traffic sent and received at each active port, distinguishing between how the MPI library uses each port in a system with multiple connections to the network. Chapter 4 provides use cases that prove the usefulness of such performance data in large-scale performance analysis.

The low-overhead of ibprof and the compactness of the profiles make this profiler suitable for always-on system monitoring solutions since it causes minimal performance degradation and has a small storage footprint. Furthermore, the user can easily enable or disable profiling for individual runs by setting the appropriate environment variables.

3.6 Summary

MPI libraries, by design, prevent the user from easily seeing the correlation between communication events in the application and data transmission over network links. This chapter proposes safely exposing InfiniBand network activity in a portable manner by extending the PERUSE utility in Open MPI. A new MPI profiler, ibprof, is implemented to record traffic metrics when the PERUSE reports activity over the InfiniBand interface. The resulting profile reports the amount of data sent between all ports used by the user's processes. 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 blocking MPI collectives. The profiling overhead is minimized by overlapping the reporting done by PERUSE with the actual data transmission event. Therefore, the data collection overhead is can be hidden by the communication latency.

Chapter 4

Visualizing MPI Traffic on Fat-tree Networks

This chapters details the development of a flexible visualization method for application traffic on fat-tree networks in order to identify communication bottlenecks and anomalies. The hierarchical topology of the fat-tree network is automatically generated from point-to-point connectivity information, and the links of the network are used to encode traffic information from application profiles. The network visualization is implemented in a new module for the Boxfish analysis tool, supporting analysis of application traffic and performance data across the physical topology of a fat-tree network. Case studies are used to demonstrate how this approach can identify communication anomalies in network applications and guide performance optimization strategies.

4.1 Overview of the Problem

The presentation of the performance data is an important factor that can affect the ease of performance analysis in a given domain[Gao et al., 2011; Isaacs et al., 2014]. Notably, performance data visualizations have been used to present performance metrics in formats that capture the physical hardware as well as the logical problem domain, as discussed in Section 2.1.2 of the Literature Review chapter. Such studies have motivated the use of more physically and logically representative visualizations in this work.

Prior to this work, none of the published visualizations were able to perform all of the following tasks:

- Automatically visualize the hierarchical topology of a fat-tree network
- Show the location of application processes on the physical nodes of the network
- Highlight the path of individual application traffic across the network
- Map the logical layout of the application's problem to the physical topology

Having these functions within a single solution is necessary to assess how correlation among the problem domain, communication pattern, and hardware configuration affect the application performance. Optimizations applied to a single domain can have an adverse effect on performance in another area, so it is important for all areas to be considered when conducting performance analysis. Based on the related literature that have been reviewed, the Boxfish [Isaacs et al., 2012] performance analysis tool provides the most of the listed function relative to other solutions. The limitation of BoxFish, in this context, is that it does not support fat-tree topologies.

The extent to which any tools can automatically capture the topology of a fat-tree network is also a challenge. While many graphing algorithms can easily create tree representation of a single-rail fat-tree network, no tool has been identified that can support a multi-rail topology, such as TSUBAME2.5.

4.2 Review of the InfiniBand Fat-tree Networks

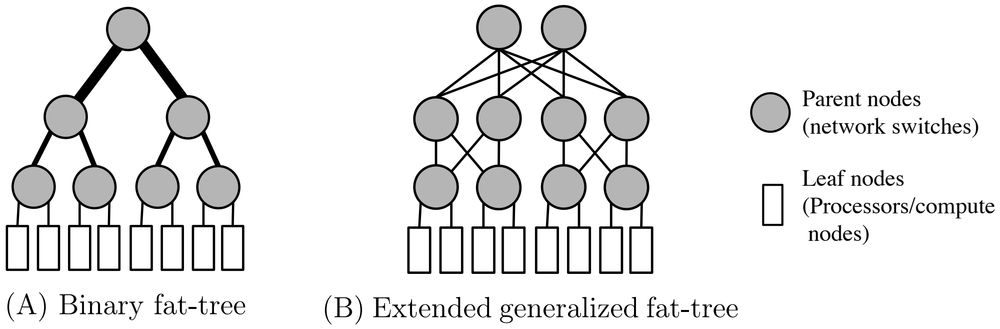
A fat-tree [Leiserson, 1985] is a hardware-efficient, hierarchical network topology that preserves full bisection bandwidth across the entire network. Logically, it can be seen as a complete binary tree with edges increasing in capacity as we move up towards the root of the tree, thereby providing full bisection bandwidth. An extended generalized fat-trees ensures uniform hardware requirements and high-throughput across the network by varying the number of connections at each level as described in [Öhring et al., 1995]. The fat-tree topologies are illustrated in Figure 4.1.

A popular fat-tree configuration is the k-ary tree that is constructed using commodity hardware, with k indicating the number of ports per switch – the *radix* of each switch [Al-Fares et al., 2008]. In this configuration, there exist groups of all-to-all connections between $k/2$ layer-1 switches and $k/2$ layer-2 switches. Each of these groups is called a pod.

This work uses a 3-level, full bisection bandwidth network constructed from 36-port switches with 72 leaf switches at level-1. The number of links in this network is $\frac{36}{2} \times 72 \times 3 = 3,888$. Each InfiniBand network link supports full-duplex communication with independent channels being used for traffic in each direction. Accurate representation of InfiniBand traffic should therefore capture this bi-directional flow of traffic.

4.3 Multi-rail InfiniBand Fat-Tree Visualization

To efficiently support analysis of application performance over fat-tree networks, a process of automatically detecting and visualizing the topology of multi-rail fat-tree networks was developed. The development of this visualization solution was necessitated by the lack of tools available to easily exploit the `ibprof` profile data (see Chapter 3). Hence, the requirements of this solution are inline with the limitations in conducting application-specific performance analysis on fat-tree networks. The description of the design process uses Munzner [2009] four-step


Parent nodes (network switches)
Leaf nodes (Processors/compute nodes)

(A) Binary fat-tree (B) Extended generalized fat-tree

Figure 4.1: **Fat-tree topologies** Line thickness indicate the link's relative capacity.

design and validation model to demonstrate how the solution meets its objectives, and the *why-how-what* topology [Brehmer and Munzner, 2013] is used for describing the analysis tasks in order to demonstrate that the design meets its objectives.

4.3.1 Methodology

Segmentation of Analysis Goals and Tasks

This visualization solution is separated into two segments. The first segment synthesizes and couples performance and visualization information into a form that can be consumed and displayed by appropriate visualization tools. The second segment is the creation of a visualization tool that can enable the identification of communication anomalies in applications running on InfiniBand fat-tree networks.

Based on the limitations described in Section 4.1, this visualization solution should accomplish the following goals by performing the following tasks:

Segment 1 *Hardware-centric performance description*

Goal 1 Capture the topology of fat-tree networks

Task 1 Portably generate topologically-correct view of the network that can be used by multiple tools

Goal 2 Support hardware-centric communication analysis. That is, presenting application performance in the context of the physical elements of the network.

Task 2 Connect application communication performance with links across the system

Segment 2 *Holistic performance visualization*

Goal 3 Exploration of application performance to detect communication anomalies.

Task 3 Flexibly and interactively visualize application traffic flowing across the network

Goal 4 Expose the relationship between the application, communication, and hardware domains.

Task 4 Simultaneously analyze the system performance metrics and application performance metrics in the same temporal and spacial location.

The input of each successive task include the output its preceding task. The additional external information required to carry out the tasks in this solution are:

1. network configuration information (required by **Tasks 1–2**),
2. application traffic information (required by **Task 2**), and
3. system performance information (required by **Tasks 4**).

Application traffic information is captured in `ibprof` profile data, and network configuration information is provided by the `ibdiagnet` utility. TSUBAME2.5 wrote `ibdiagnet` output files to the file system every hour. This frequency is satisfactory since the network configuration doesn't change unless there is a failure, the network is restarted, or the system administrators intentionally change the configuration. The `ibdiagnet.lst` file provides port connection information and the `ibdiagnet.fdbs` file provides port forwarding information, otherwise referred to as packet switching rules. Packets switching rules allow for the paths of packets to be projected across the network based on their destinations. System performance information can be retrieved from environment monitoring tools and can be processed in a similar manner as application traffic information.

Visual Representation of Data and Tasks

The goals (or *"why"*?) of the visualizations tasks have been stated in the previous subsection. A fat-tree topology is essentially a node-link graph with *nodes* in the graph representing switches, compute nodes, and other processing elements in the network. The node-link representation has been successfully used in to depict fat-trees in previous work as it most closely resembles the user's mental model of the network. A 1-dimensional (1D) node-link graph representation is used in this dissertation because the hierarchical node-link topology is easy for users to conceptually process since the levels in the network map directly to the levels in the hierarchical graph, unlike matrix-based representations [Ghoniem et al., 2004, 2005]. Section 4.4 will demonstrated that the degree of information occlusion in this hierarchical node-link representation is not prohibitive, especially when appropriate filters are used.

The manner (or *"how?"*) of achieving achieving the goals are as follow:

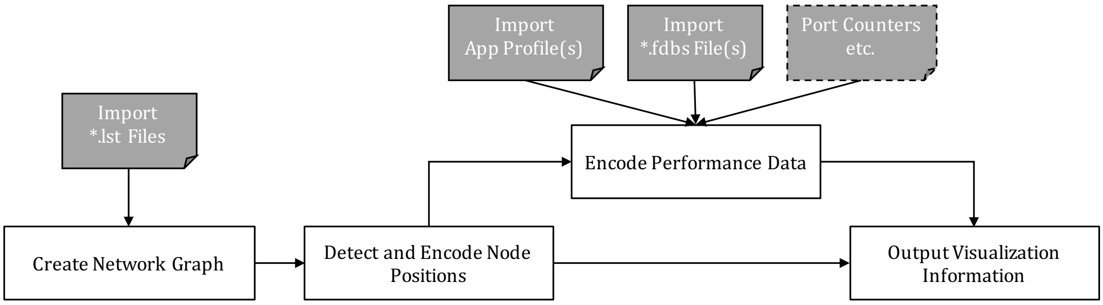


Figure 4.2: Overview of the network layout process.

Task 1 is achieved by encoding horizontal (x) and vertical (y) node position information that is inline with the constraints of a fat-tree network. For example, nodes in the same level cannot be connected.

Task 2 is achieved by encoding application traffic values on the links of the network and recording the network information in a generic, easily parsed format. This way, viewing the network will support viewing the performance over the network.

Task 3 is achieved by arranging the glyphs for nodes and links using only information provided in the output of Task 2. The tool must not infer anything about the topology while supporting navigation, filtering, and selection.

Task 4 is achieved by importing, selecting, and filtering performance information across different classes (or domains: application, communication, hardware) based on user input. Supporting features are already implemented in BoxFish and are not redefined in this dissertation. Isaacs et al. [2012] provides an overview of the features that are relevant to this task.

The items being processed and presented (or *"what?"*) for **Tasks 1–4** are the traffic values, nodes, and links. These are the only things required to investigate application performance over the network.

4.3.2 Visualization Design and Implementation

A network layout tool[1] was designed to encode the network layout (**Task 1**) and application traffic (**Task 2**), while a visualization module[2] was created in the BoxFish analysis tool to visualize (**Task 3**) and analyze traffic performance over the network (**Task 4**). The operations of the network layout tool are illustrated in Figure 4.2 and have been implemented in Python.

Encoding Network Layout

The layout tool first extracts network configuration information from the `ibdiagnet` output files. After parsing all input files, the tool creates a connected graph to

[1] https://bitbucket.org/kevinabrown/ibprof_converter

[2] https://bitbucket.org/kevinabrown/boxfish

represent the nodes[3] and links in the network by using the port-to-port connections listed in the `ibdiagnet.lst` file. This file also specifies whether a node is a switching node or compute node. Starting from a random compute node, a sequence of breadth-first searches is performed to identify each node's nearest neighbor and determine the vertical position of each switch in the fat-tree topology. The vertical position of each node is determined based on the shortest distance to any compute node:

- Level-0 nodes are compute nodes since they are at the bottom of the fat-tree.
- Level-1 nodes are directly connected to compute nodes: nodes that are one hop away from the nearest compute node.
- Level-2 nodes are two hops away from the nearest compute node.
- Level-Y are Y hops away from the nearest compute node.

The horizontal position of nodes in each level is assigned based on another breadth-first traversal of the graph. Once a node is reached during this traversal, it is assigned the next horizontal positions in sequence at its level.

If the network has two subnet and an `ibdiagnet.lst` file is provided for each, the vertical positions nodes in the second subnet are inverted by negating the y value of each node. Hence, level-Y nodes in the first subnet will have positions of (x_i, Y) and level-2 nodes in the second subnet will have values of $(x_i, -Y)$.

Encoding Application Traffic

Application profiles from `ibprof` are also parsed by the layout tool and provide information on (i) which MPI process is running on which node, (ii) the InfiniBand port configuration for nodes with MPI processes and (iii) the size and destination of traffic sent per-port per-code region. The `ibdiagnet.fdbs` file provides port forwarding information, otherwise referred to as packet switching rules.

Weights are added to the link ends by tracing the path of application traffic across the network using the port forwarding tables. The weight at each link end represent the number of bytes transferred on the link by the node connected to that end of the link. The tool also supports reading other form of performance data, such as port counters etc. Finally, output files are written containing position and performance information of the network nodes and links. The tool supports YAML and comma separated value (CSV) output formats.

Implementation of BoxFish Fat-tree Module

The new BoxFish visualization module is similar in structure to the `3D Torus` module that is distributed with the Boxfish tool. However, `Fat Tree` module differs in the way performance data is referenced and visualized. With the `3D Torus`

[3] The term *node* is used in reference to both switches (or switching nodes) and compute nodes.

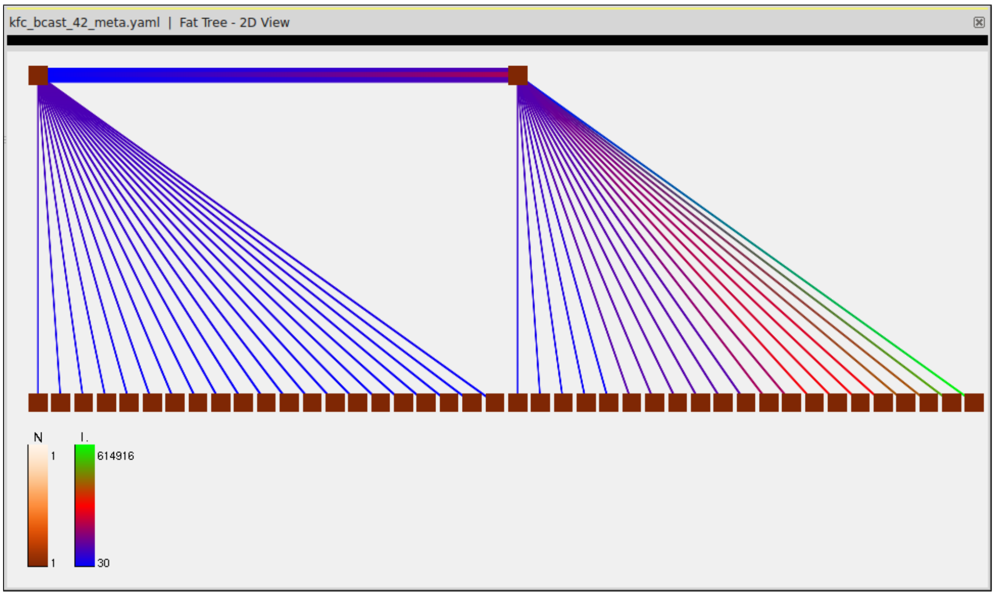


Figure 4.3: **Boxfish `Fat Tree` module** | sample visualization

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. The `Fat Tree` 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 provides the flexibility of visualizing any 2D network topologies, for example: fat-tree, 2D mesh, and dragonfly networks.

To ensure that the bidirectional flow of traffic is accurately represent,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. The 2D visualization of the `Fat Tree` module can be zoomed and panned, focusing on any area of the network specified by the user. Support for standard BoxFish highlighting, filtering, and module linking features are also provided in our module. By implementing 3D rendering and 3D rotation support, we can extend the `Fat Tree` to further support all 3-dimensional topologies.

Figure 4.3 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 maps that are 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.

4.4 Case Studies

In this Section, the usability of the `Fat-Tree` visualization is demonstrated using **ibprof**'s communication profiles. This is achieved via an analysis of the execution of *samplesort*, a popular sorting algorithm for parallel systems, and a comparative analysis of the performances of different Open MPI library versions. Experiments were conducted on TSUBAME2.5, which utilizes two independent InfiniBand subnets and each compute node has a link to each subnet. It is important to note that other users were sharing the network while these experiments were being conducted. The presence of other users on the network means that the communication performance of our application can be affected by the interference from other users. Therefore, instead of using the application runtime as the main performance metric, the intensity and spread of traffic are also considered when assessing performance.

4.4.1 Visualizing Traffic Patterns and Contention in Samplesort

Samplesort, as described in [Sundar et al., 2013], is a sorting algorithm for distributed memory environments. The algorithm is designed to find a set splitters that 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. The algorithm starts with each process possessing a subset of the unsorted keys. Keys are randomly sampled across all processes and these samples are used to identify the splitters. Data is split locally into chunks belonging to each bucket and then all the chunks are sent to their destination processes via global all-to-all. Finally, each process locally merges and sorts its bucket.

This case study uses the samplesort code presented in [Sundar et al., 2013] [4]. The code with 128 MPI processes, using a 1:1 process-to-node mapping. Each process is started with 1 GB of unsorted integers, randomly generated with a uniform distribution. The same random number seed was used in all cases.

Based on the performance data outputted by the samplesort program, the all-to-all exchange of data took 3.74 s. Figure 4.4 shows a typical process-centric visualization of samplesort's main communication routines over 128 nodes using Paraver [Barcelona Supercomputing Center, 2014], a flexible performance analysis tool developed at the Barcelona Supercomputing Center (BSC). Paraver extracts performance information from the MPI library using the PMPI interface, which we consider a process-centric approach since it provides only information that's

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

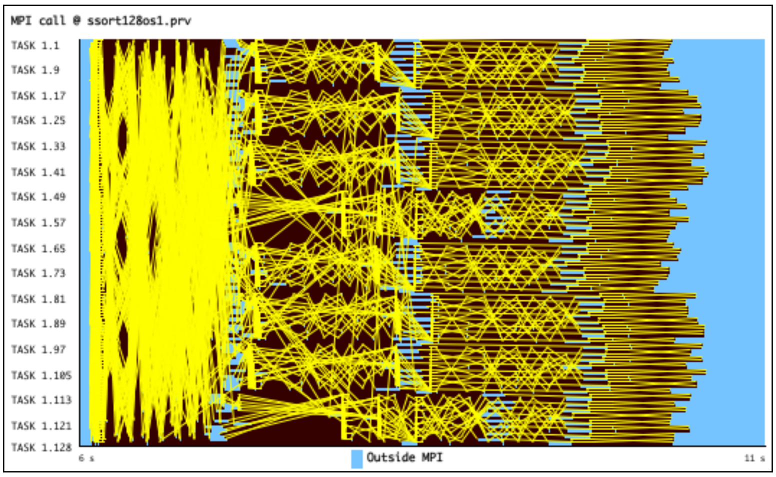


Figure 4.4: **Paraver visualization** | Each horizontal, multi-colored rows represent the execution of different samplesort process, and the yellow lines drawn diagonally across the rows indicate communications among the processes. The blue sections are periods spent outside the MPI library, while the brown sections indicate duration spent inside the MPI library.

visible from the application level. It is impossible to extract any network performance insights from this and other similar visualizations that are generated using PMPI-based instrumentation tools.

Performance Analysis using our `ibprof` Profiler and our Boxfish Module

An execution of samplesort was profiled using `ibprof` and the traffic patterns visualized using the newly developed 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. After the run, the `ibdiagnet` tool was used to collect up-to-date network configuration information. The post-processing step was then performed to combine the application profile with network configuration information to produces visualization information that were feed into Boxfish for analysis. Figure 4.5 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 Figure 4.4. These visualizations have been filtered to show only the network links that were used by the samplesort communication traffic. Network links that were not used during the execution are not shown.

The red links that are visible in area C of Figure 4.5 represent links that were carrying the most traffic during the communication block of the code. Exploring the visualization and applying filters for different traffic intensities allow for the identification 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.

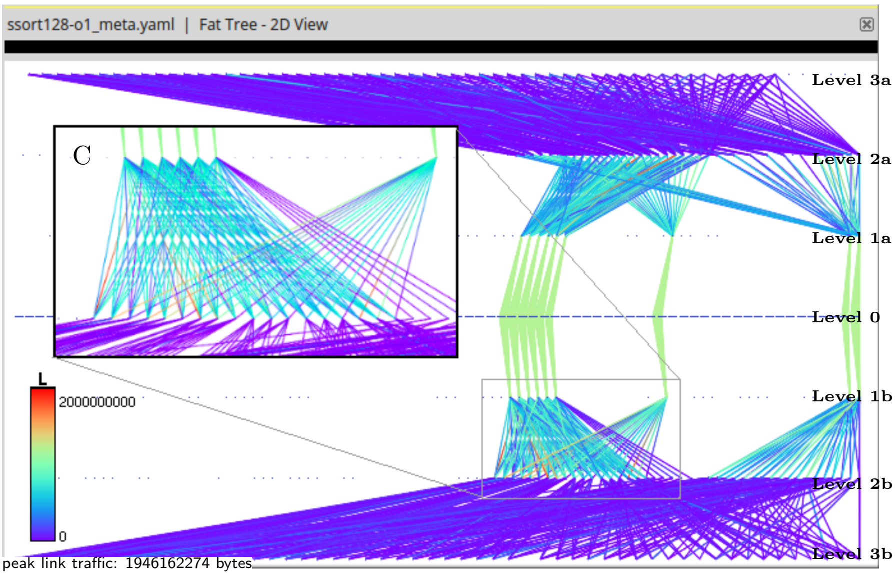

ssort128-o1_meta.yaml | Fat Tree - 2D View
Level 3a
Level 2a
Level 1a
Level 0
Level 1b
Level 2b
Level 3b
C
L
2000000000
0
peak link traffic: 1946162274 bytes

Figure 4.5: **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.

Optimization Efforts

The first attempt at reducing this contention was to identify which processes were sending/receiving the most traffic over the most active links by using data from the profiles. Once identified, these processes were moved to other nodes with the intention of having their traffic sent via 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, such as storage and management nodes, were connected to a single subnet, which results in non-identical port forwarding tables across both subnets. This difference was visually confirm using the Boxfish `Fat Tree` module by noting the difference in traffic patterns across both subnets, i.e., comparing the upper and lower halves of Figure 4.5. The 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.

The second attempt at contention reduction was to use a different set of nodes for the experiment. This is shown in Figure 4.6. The traffic visualization of this new run reveals that the nodes are more tightly clustered. This resulted in the peak application traffic per link being reduced by over 20%. Additionally, the traffic in this setup does not use level 3 switches in either of the subnets, thereby

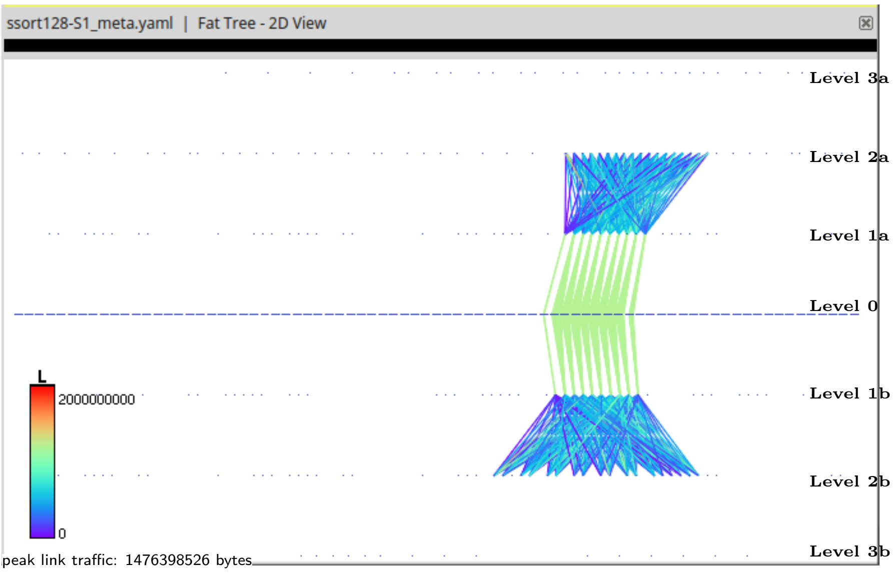

ssort128-S1_meta.yaml | Fat Tree - 2D View
Level 3a
Level 2a
Level 1a
Level 0
Level 1b
Level 2b
Level 3b
L
2000000000
0
peak link traffic: 1476398526 bytes

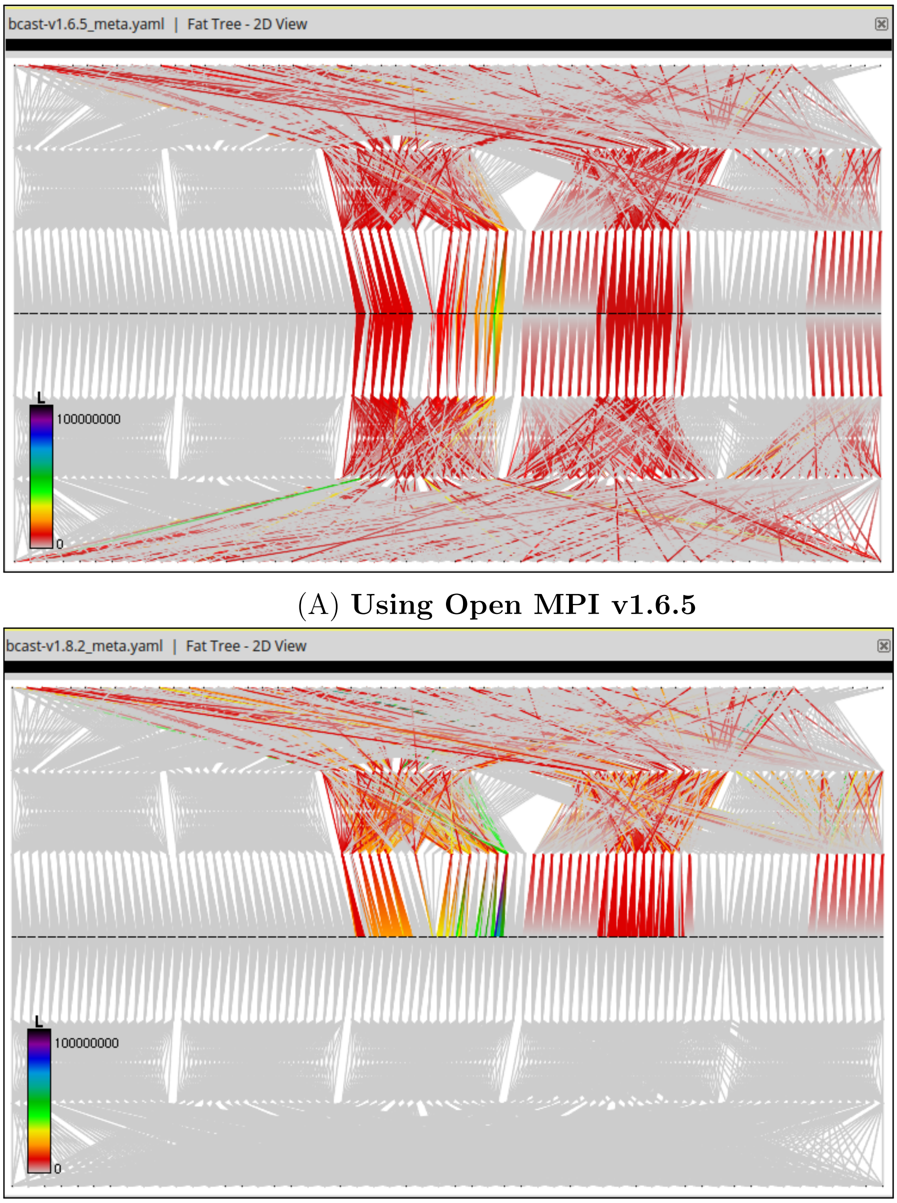
Figure 4.6: **Boxfish visualization of samplesort's communication traffic after using a different node set** | The nodes in the new set are in close physical proximity to each other. Notice that no traffic is going to level 3 switches, and hence those links are not visible.

reducing the communication latency. We compared the runtimes over both node sets and found that the all-to-all performs better on the new allocation 94 times out of 155 trials. The average performance gain was 5.08%.

4.4.2 Visualizing Traffic Patterns Inside the MPI Library

A comparison of benchmark runtimes using different versions of Open MPI on TSUBAME2.5 uncovered the fact a newer 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 [Intel Corportation, a] 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, `ibprof` was used to profile runs of an `MPI_Bcast` microbenchmark with the different versions of Open MPI and the default system parameters. As illustrated in Figure 4.7, 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 introduced a new MCA parameter to control which InfiniBand adapter is used for the interface. The new parameter is called `btl_openib_ignore_locality` whose default value of “0”, causing the library to not use all interfaces. While setting the value to “1” allowed all interfaces to be used, more fine-grained profiling using `ibprof` reported that each operation being being completed by a single interface in v1.8.2. in a round-robin manner. For example, an application with six(6) successive calls to `MPI_Bcast` for large messages would have calls the 1^{st},3^{rd}, and 5^{th} calls being sent to the first inter-


bcast-v1.6.5_meta.yaml | Fat Tree - 2D View
L
100000000
0
bcast-v1.8.2_meta.yaml | Fat Tree - 2D View
L
100000000
0

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

(B) **Using Open MPI 1.8.2**

Figure 4.7: **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 visualizations.

face and 2^{nd}, 4^{th}, and 6^{th} calls sent to the second interface. This is unlike Open MPI v1.6.5, which splits messages across both interfaces within MPI operations instead of across operations. Traffic within each operation would be balanced across both interfaces.

4.4.3 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. This section has demonstrated that by conducting performance analysis at a lower level, the new `ibprof` profiling utility and hardware-centric visualizations can provide the ability to locate potential bottlenecks in a portable and non-intrusive way.

This toolchain can be used 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 this tool-chain extends the analysis capabilities of MPI library developers, network designers, and systems administrators as they are presented with new insights from familiar visual representations of their environment. Research areas that can benefit from this solution include, but are not limited to, the improvement of MPI collectives and the optimization of routing algorithms to increase the efficiency of current and future HPC systems.

4.5 Summary

To support extended analysis of application performance on fat-tree networks, a visualization solution is proposed to efficiently captures the hierarchical structure of the network and show application across the links of the network. Firstly, a network layout tool is presented that can automatically encode the layout network and application performance over the hardware elements in the network use a series of breath-first search operations. Secondly, a new Boxfish `Fat Tree` module is created to produce interactive visualization of the network and application performance. This module can expose the network-level performance of MPI applications running on any 2D network topology such as fat-trees. Furthermore, despite its name, the `Fat Tree` module can natively visualize any 2D network and can be extended to also support 3D networks in future work.

Case studies have proven that this hardware-centric, low-level manner of performance analysis offers insight into the performance of MPI operations in ways that PMPI-based, process-centric tools cannot. This hardware-centric 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.

Chapter 5

Characterizing I/O vs MPI Interference

This chapter presents a quantitative and qualitative characterization of the interference between I/O and MPI traffic on fat-tree networks. The first portion of this chapter deals the case when the system is split evenly between the MPI and I/O jobs. That is, 50% of the compute nodes are used for the I/O job and the remaining nodes are used for the MPI job. This configuration is useful for making a fair comparison between how each class of traffic reacts to interference from the opposing traffic class. However, nodes are not usually partitioned this evenly in practice. The size of a job will depend on the requirements of the user, the application/problem, and the availability of resources: hardware, software licenses, execution budget, etc. Hence, another portion of this chapter covers cases where the jobs vary in sizes in order to understand how a job's scale affects the interference it experiences.

5.1 Overview of the Problem

Unfortunately, most modern supercomputers use the same network infrastructure for both MPI and I/O traffic. The scarcity of studies investigating the interference between these two types of workloads leave several open questions regarding the nature of interference between the both types of traffic. Currently, the impact of message sizes used for the MPI and I/O traffic as well as the impact of frequency of such data movement on the resulting interference is not understood.

Shared Resources

In most fat-tree systems, e.g. Tianhe-2 [Dongarra, 2013], Stampede [Texas Advanced Computing Center, 2013], and TSUBAME3 [Tokyo Institute of Technology, 2017], the network infrastructure is used for both I/O and MPI traffic, i.e. the I/O traffic must traverse the same network links as the MPI traffic when the I/O data is being moved between compute nodes and I/O servers. In most cases, this configuration is more economical than providing each compute node with an additional, independent connection to the storage network.

It is typically anticipated that I/O bottleneck will only exist between the I/O servers and disk because (i) the speed of storage devices is slower than modern network bandwidths and (ii) I/O traffic involves n-I/O clients accessing m-I/O servers, where $n > m$. However, with the advent of novel storage architectures such as burst buffers and in-memory file systems, current expectations may not hold for future systems and more I/O bottlenecks may occur in over the network. It is also unclear if a high bisection bandwidth fat-tree topology would suffer from interference between I/O and MPI traffic. The answers to the issues can guide optimization efforts as well as configuration of I/O subsystems, network infrastructure, and communication libraries.

Isolated Performance Analysis

The performance of MPI application and different MPI implementations have been studied extensively over the past three decades. Similarly, I/O performance features and trends have been studied at great lengths for all available I/O subsystem architectures. Nearly all MPI studies are conducted without consideration of any other forms of network traffic and the same is true for nearly all I/O studies, as discussed in Chapter 2.

When the network is shared between I/O and MPI, the I/O congestion can impact the performance of MPI traffic since congestion reduces the effective network bandwidth. At the same time, congestion caused by MPI traffic can also interfere with I/O traffic. Mubarak et al. [2017a] confirmed and quantified the effect of MPI-I/O interference on MPI performance in the presence of checkpointing I/O traffic on dragonfly networks. However, their study is topology-specific and cannot be used to quantify the degradation experienced on fat-tree networks. At the time of this dissertation, no other study has quantified the effect of MPI traffic on I/O performance on fat-tree networks.

5.2 Requirements of Interference Characterization

In order to carefully understand the aforementioned issues, a characterization of the effects of interference between the MPI traffic and the I/O traffic on each of these traffic types must be considered. The investigation in this dissertation explores several factors that impact the interference, including message sizes, communication intervals, and system allocations. The results aim to characterize the importance of each of these factors and expose performance trends due to variations in their values.

Mubarak et al. [2017a] results showed that packets generated by MPI communication of one job can experience over 4000× increase in maximum latency due to interference from I/O traffic of another job on dragonfly networks. For this dissertation, the maximum latencies are not the focus of the interference characterization since maximum values are more random in occurrence. Instead, the statistical distribution of performance is considered since this enables more consistent interference trends to be identified.

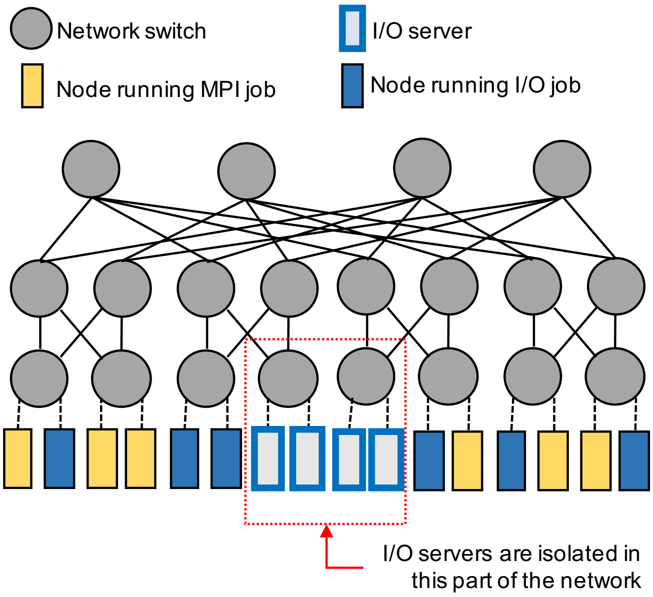


Figure 5.1: Fat-tree network with isolated I/O servers. This illustration does not represent the actual number of nodes or switches.

5.3 Methodology

5.3.1 Simulation Environment

The simulator is implemented using the CODES simulation toolkit [Mubarak et al., 2017b]. **CO**-**D**esign of **E**xascale **S**torage and data-intensive systems, or CODES, is a framework for studying HPC interconnects, storage systems, and applications using parallel discrete-event simulation. CODES in built on top of the Rensselaer's Optimistic Simulation System (ROSS)[Carothers et al., 2002], a parallel Time Warp system, and CODES provides high-fidelity, packet-level network models for several interconnect topologies including the fat-tree and dragonfly topologies. Several recent studies related to communication on different network topologies have been conducted using CODES and have provided validation results to show that CODES is able to predict similar performance results as the real world systems [Jain et al., 2017; Mubarak and Ross, 2017; Mubarak et al., 2017a].

In order to generate the I/O and MPI traffic, a synthetic workload generator is coupled with the validated fat-tree model provided by CODES. Unlike on real systems, simulations allow for more fine-grained monitoring of the network traffic without inadvertently perturbing the application's behavior or performance. A simulated system is also not limited by the constraints posed by the deployed systems, such as link speeds and the the physical placement of I/O servers.

5.3.2 System Configuration

As stated earlier, CODES-based network simulations are used to conduct experiments by generating synthetic traffic patterns that capture the salient features of typical I/O and MPI workloads in HPC. The simulated system consists of 1,296 nodes that are interconnected by a three-level, InfiniBand-like fat-tree network with 12.5 GB/s links. Each node is connected to one of the 72 36-port level-1 (or leaf) switches. Of the 1,296 nodes, 72 function as I/O servers and 1,224 function as compute nodes. This partitioning is similar to the Cab supercomputer [Laboratory, 2018] that is in production at Lawrence Livermore National Laboratory. These I/O servers are similar to LNET routers described in Chapter 2. Unless otherwise specified, all I/O servers in this system are grouped and connected to one of four leaf switches. These four leaf switches are split between the two halves of the network, similar to the system depicted in Figure 5.1. In the default case, the 1,224 compute nodes are allocated equally between the I/O and MPI jobs, both jobs being assigned to 612 random nodes. In summary, each of 1,296 nodes of the simulated system has one of following three roles in the environment:

- MPI client: a compute node that generates MPI traffic
- I/O client: a compute node that generates I/O traffic
- I/O server: a service node that receives I/O traffic from I/O clients. This can either be an LNET router [lus, 2018] or an actual I/O server with attached storage.

5.3.3 Traffic Workload

The synthetic communication patterns[1] that are used for both I/O and MPI jobs were designed to capture the characteristics of real HPC workloads. Since the aim of this work is to get a broader understanding of the inter-job interference, the results from well chosen synthetic patterns will be more generalizable than a study of patterns from a specific application [Jain et al., 2017; Mubarak and Ross, 2017]. The workload in this study uses samples of small, medium, and large message sizes with moderate and high traffic intensities. These samples expose trends in how a wide cross-section of different traffic sizes and intensities will respond to interference.

MPI Job

Several surveys of representative MPI workloads have indicated that HPC applications generate a wide range of message sizes from small (few KBs) to large (few MBs) at a wide range of communication frequencies [León et al., 2016; Jain et al., 2017; Mondragon et al., 2016]. These characteristics of application communication pattern impact the congestion caused by MPI traffic, and therefore, must be considered when characterizing overall MPI performance.

The MPI job consists of one process per node. MPI processes are paired randomly, and each process sends a fixed amount of data to its partner at a fixed average interval – which is computed as the time between the completion time

[1] For the purpose of this work, "communication pattern" refers to the size and frequency of messages/requests sent by processes in a job. All processes of the same job use the same communication pattern.

of the previous message and the start time of the next message. The message sizes and intervals used in each experiment are chosen to reflect a cross-section of the MPI traffic patterns found in applications running on HPC system [Jain et al., 2017; Kurth et al., 2017]. MPI message sizes range from 4 KB to 4 MB the intervals range from 100 μs to 100 ms in this study.

I/O Job

Using the Lustre I/O-forwarding pattern, each I/O client randomly selects an initial I/O server for its first request. Subsequent requests from that client are sent in a round-robin manner to other I/O servers. Unless otherwise stated, each I/O client writes approximately 4 GB (1000× 4 MB-requests) for experiments reporting I/O performance. Different request intervals are used to represent jobs where I/O data is written in pieces, e.g. periodic visualization data output during a scientific simulation. Previous studies have demonstrated the range of I/O patterns on HPC system [Luu et al., 2015; Kurth et al., 2017; Latham et al., 2012; Oral et al., 2014]. For other cases, there is a negligible interval between I/O requests to represent single checkpoint-style I/O dump where each client writes all of its data in an unbroken stream of requests. Checkpointing, in a more comprehensive sense, can be done by writing all data at once or with data being written at periodic intervals by using multi-level checkpointing libraries [Moody et al., 2010; Bautista-Gomez et al., 2011] and uncoordinated checkpointing [Ferreira et al., 2014]. However, unless otherwise specified, all reference to *checkpoint-style* I/O traffic indicate the case of single checkpoint dump that involves all data being written at once without any significant interval between successive I/O requests.

5.3.4 Execution and Measurement

Interference between MPI and I/O traffic is quantified by comparing the message latency of a job when it uses the system exclusively to the message latency of the same job when it shares the system with another job. The exclusive-system case is referred to as the **baseline run** and the shared-system case as the **interference run**.

The time for every MPI message and I/O request is recorded during each simulation. For consistency, the duration of each run is long enough to collect at approximately 1000 data points per node for the chosen job, i.e. I/O or MPI. This means that two interference runs were conducted for each configuration: one interference run was done to measure the change in I/O time due to MPI interference, and the second interference run was done to measure the change in MPI time due to I/O interference. Initially, 50 warm-up messages/requests are sent from each client before the communication times are recorded. The message/request intervals are varied randomly between a ±5% bound to account for system noise and variations in communication time across the clients. Hence all intervals reported in this work represent the average delay between consecutive messages/requests, and the delay varies uniformly about the arithmetic mean with a maximum variation of ±5%.

SIZES	50 ms	10 ms	5 ms	1 ms	500 μs	100 μs	50 μs	10 μs	5 μs	1 μs
4 KB					✓	✓	✓	✓	✓	✓
512 KB			✓	✓	✓	✓	✓	✓		
4 MB	✓	✓	✓	✓	✓	✓				

Table 5.1: Intervals between consecutive MPI messages or I/O request in the main survey.

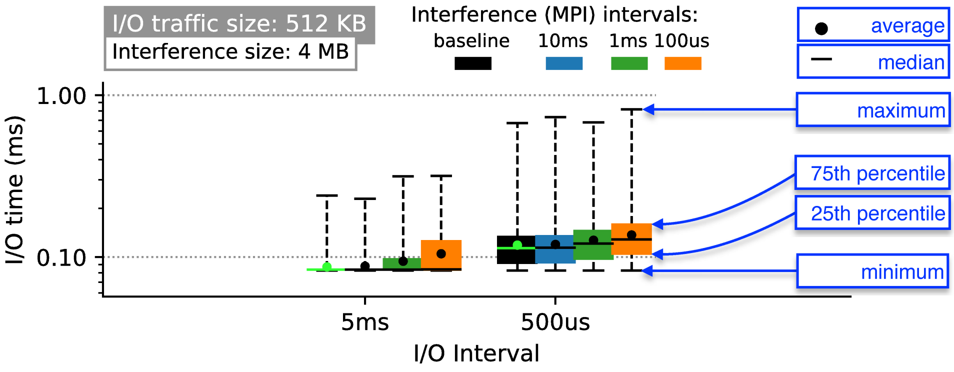


Figure 5.2: **Guide chart** – An example chart showing the format and describing the main elements of charts used in this paper.

In order to characterize the interference, different MPI and I/O traffic patterns are studied and the interference between each pair of patterns is evaluated. The impact of job size on the resulting interference is also investigated by altering the number of MPI and I/O clients on the system.

5.4 Characterization

In order to characterize the interference between MPI and I/O traffic, one must not only capture the extent of the interference, but should also understand how the distinctive features of the traffic patterns result in different interference patterns. This section shows the resulting trends when different MPI traffic patterns and different I/O traffic patterns interfere with each other.

The traffic patterns that are studied in this main survey, depicted in Table 5.1, are defined by the message/request size and its interval. Preliminary experiments informed that this combination of sizes and intervals would effectively demonstrate the progression between low ↔ moderate and between moderate ↔ high interference. The performance trends of I/O and MPI jobs are discussed individually in different subsections since their respective features are unique to their respective communication patterns. The final two subsections evaluate the impact of job sizes on inter-job interference and discuss the reasons for the difference in the performance trends.

Figure 5.2 illustrates an example of how the results will be presented in this section, showing 512 KB I/O traffic performance while 4 MB MPI messages are being sent in the background. MPI's performance is not shown in this figure. Two cluster of bars are presented, reporting performance for 512 KB I/O requests arriving at 5 ms and 100 μs intervals, respectively. The baseline bars show the I/O request time when there is no MPI interference. The color of each bar indicate the interval of the interfering (MPI) traffic. Black bars show the time for the baseline run when there is no interfering traffic in the background, while each colored bar represents a different interval as specified in the legend. Within each cluster of bars, a difference in the positions of colored bars relative to the position of the cluster's baseline bar in a indicator of performance variation due to interference. Therefore, variation in I/O request time due to MPI interference is depicted by a change in the following attributes of a colored bar relative to its corresponding baseline bar:

- the height of the average mark, the circular dot
- the height of the median mark, the horizontal line
- the height and the length of the bar

MPI performance will be reported in separate charts, which will use the aforementioned layout.

In addition to the broad survey across all combination of message sizes, more detailed inspections of the performance trends for 4 MB MPI messages and 4 MB I/O requests are conducted. These inspections include additional message and request intervals that are not shown in Table 5.1.

5.4.1 Performance of the I/O Job

Figure 5.3 shows the I/O performance when different I/O request sizes interact with different MPI message sizes. Maximum and minimum values are omitted in this figure to improve clarity. In the figure, I/O request size increases down each column of charts, while the interfering MPI message size increases across each row of charts from left to right. For example, left column charts A, D, and G show I/O performance for 4 KB, 512 KB, and 4 MB I/O requests, respectively, while the MPI size is fixed at 4 KB for these three chart.

The Least and Most Significant Slowdowns

Chart A shows no significant deviation in the positions of dots nor bars within any cluster, indicating that the performance of 4 KB I/O requests is not affected by 4 KB MPI background interference. However, when the size of the interfering messages increases to 4 MB, as shown in chart C, the time of 4 KB requests increase due to MPI interference. The heights of the bars increase within each cluster of chart C as the MPI interval is reduced, indicating that the request times increases due to increasing the MPI interference. The 4 KB I/O request experiences the greatest average slowdown of all I/O request sizes, highlighted by

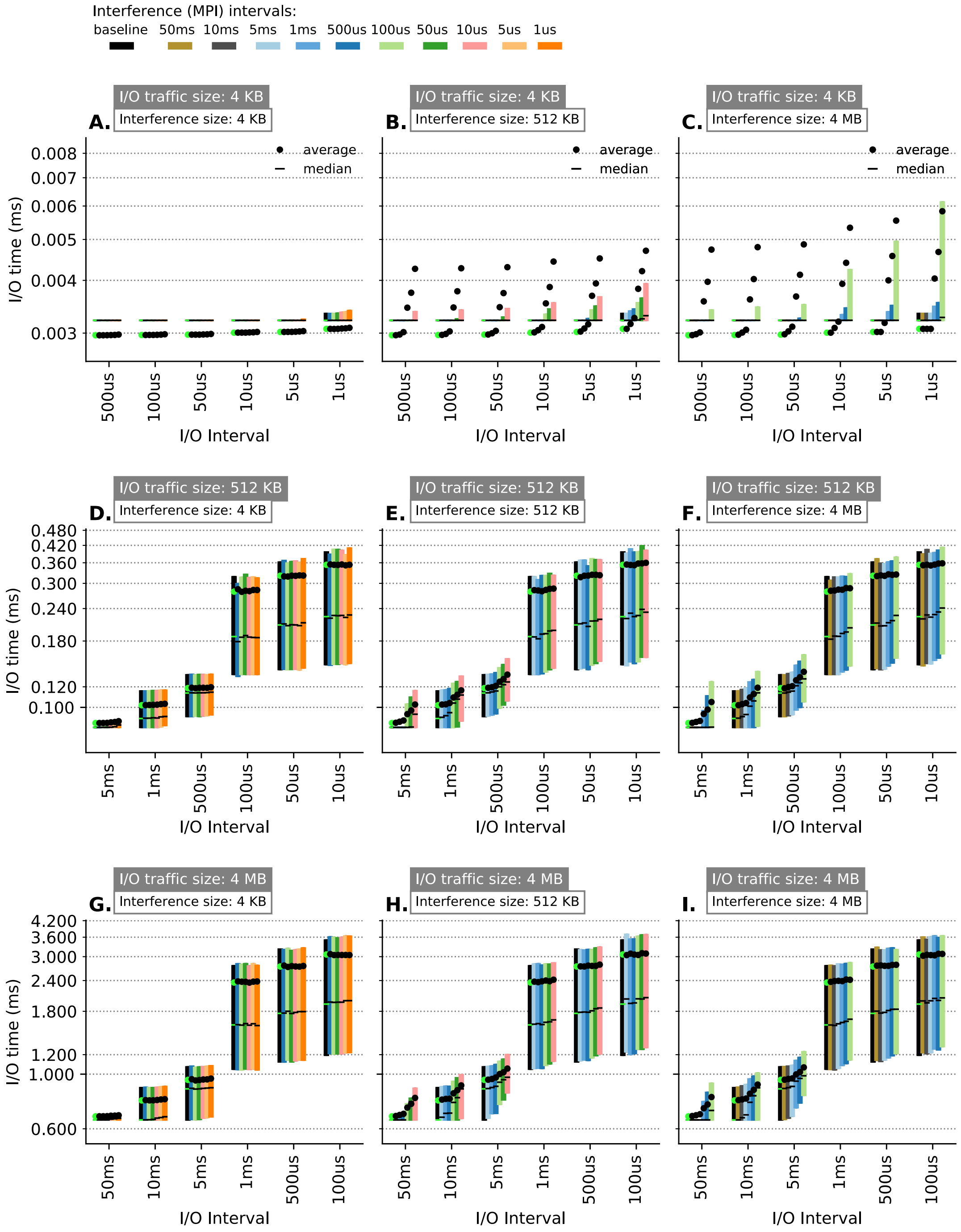


Figure 5.3: Performance of three I/O request sizes being interfered by three MPI message sizes, one traffic-interference pair per chart. A difference in the height of the features of the colored bars (interference runs) relative to the features of the black bar (baseline run) in the same group indicates a performance variation due to interference. For example, Chart C shows significant variations within each cluster, reporting that 4 KB I/O traffic is significantly affected by 4 MB MPI messages.

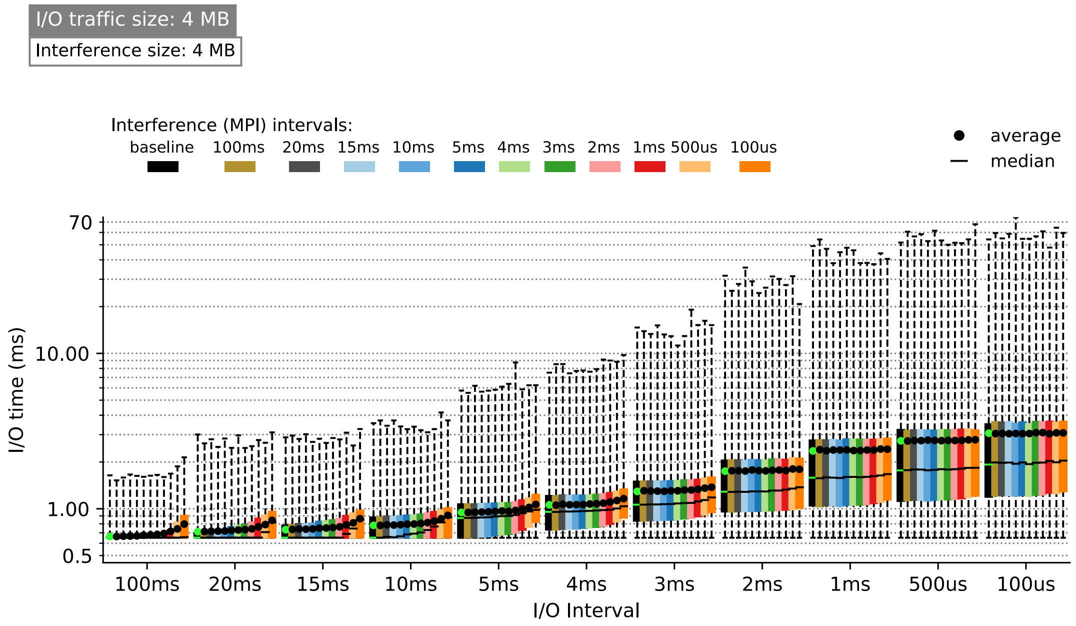


Figure 5.4: A detailed look at the performance of 4 MB I/O requests with interfering (MPI) message size of 4 MB.

the rightmost bar of chart C, with the most significant slowdown occurring when 4 MB MPI messages are sent at 100 μs intervals. For this slowdown, the average 4 KB I/O request time increases from 3.07 μs to 5.9 μs, a 1.9× slowdown.

I/O Congestion and MPI Interference

For charts D–I, there is a significant increase in the I/O times for the last three clusters. As this increase also occurs for the black baseline bars, which show the request times when there is no interference, the increase cannot be attributed to MPI traffic. Rather, the increase happens when the I/O interval is reduced (going from left to right within a chart) and I/O packets are injected into the network at a higher rate. The resulting I/O-congestion causes the increase in I/O request time. For this work, the I/O interval around which this rapid increase in request time happens is referred to as the *I/O-congestion threshold*. The 4 KB I/O traffic scenarios (charts A–C) do not cross its I/O-congestion threshold for the I/O intervals we tested. The I/O-congestion threshold occurs when the I/O interval is between 500-100 μs for 512 KB I/O requests and between 5-1 ms for 4 MB I/O requests.

When the interfering MPI message size is 512 KB (chart B, E, F) and 4 MB (charts C, F, I), an increase in the MPI traffic intensity causes a corresponding increase in the I/O request times before the I/O-congestion threshold is reached. For 4 MB I/O versus 4 MB MPI (chart I), this trend is depicted within each cluster of bars by a gradual increase in the positions of colored bars for I/O intervals 50 ms, 10 ms, as 5 ms. Once the I/O-congestion threshold is crossed at 1 ms I/O intervals, the interfering MPI traffic does not show a significant impact on I/O performance.

Detailed View of Interference Trend

Figure 5.4 presents a more detailed view of the case in which 4 MB I/O requests interact with 4 MB MPI messages. The inclusion of additional intervals shows how the impact of interference caused by MPI gradually declines as the I/O interval approaches the I/O-congestion threshold. At the 2 ms I/O interval, there is little or no increase in time for the I/O averages nor the 75^{th} percentiles within the cluster. Nevertheless, the baseline (black bar) after the I/O-congestion threshold remains higher than all the interference runs (colored bars) prior to the congestion threshold. If the time per I/O request is more important than the volume of requests for an I/O job, it is therefore better to send less frequent requests despite the higher relative impact of interference from MPI.

A comparison of the charts in the Figure 5.3 suggests that different I/O request sizes exhibit a similar pattern around their I/O-congestion threshold. Hence, the detailed trend seen inFigure 5.4 can be used to represent the behavior of different I/O request sizes.

5.4.2 Performance of the MPI Job

The performance results for MPI traffic in the presence of I/O interference are shown in Figure 5.5. These results can be interpreted using Figure 5.2 as a guide. The horizontal arrangement of dots within each cluster and the absence of visible bars in chart A of Figure 5.5 indicate that there is relatively no performance variation due to I/O interference or otherwise for 4 KB MPI versus 4 KB I/O. However, all other charts show some degree of elevation in the interference times relative to the baseline times. MPI messages experienced a peak average slowdown of 7.6× due to I/O interference, as seen in chart C.

MPI Congestion and I/O Interference

The effect of the self-congestion due to MPI messages is visible in the changing heights of the baseline bars across clusters in charts D–I, i.e., for MPI message sizes 512 MB and 4 MB. However, unlike the I/O case, MPI slowdown due to self-congestion caused does not prevent inter-job interference from inducing notable MPI slowdown. This suggests that regardless of how intensely MPI data is injected into the network, it is still vulnerable to further considerable slowdown due to background I/O traffic that use the same network links. The 75^{th} percentile of message times in the 4 MB baseline run increases from 351 μs to 630 μs when the MPI interval is reduced from 5 ms to 1 ms in chart I. Nevertheless, interference continues to affect MPI traffic when the message interval is below 1 ms.

MPI Traffic vs. I/O Congestion

Chart C shows that the overall height of each cluster decreases as the MPI interval is reduced, going from left to right in that chart. This indicates that reducing the interval between MPI messages reduces the average slowdown due to interference. This effect is most conspicuous in the rightmost bar of each cluster of this chart, where the I/O packet injection rate is highest in this chart – due

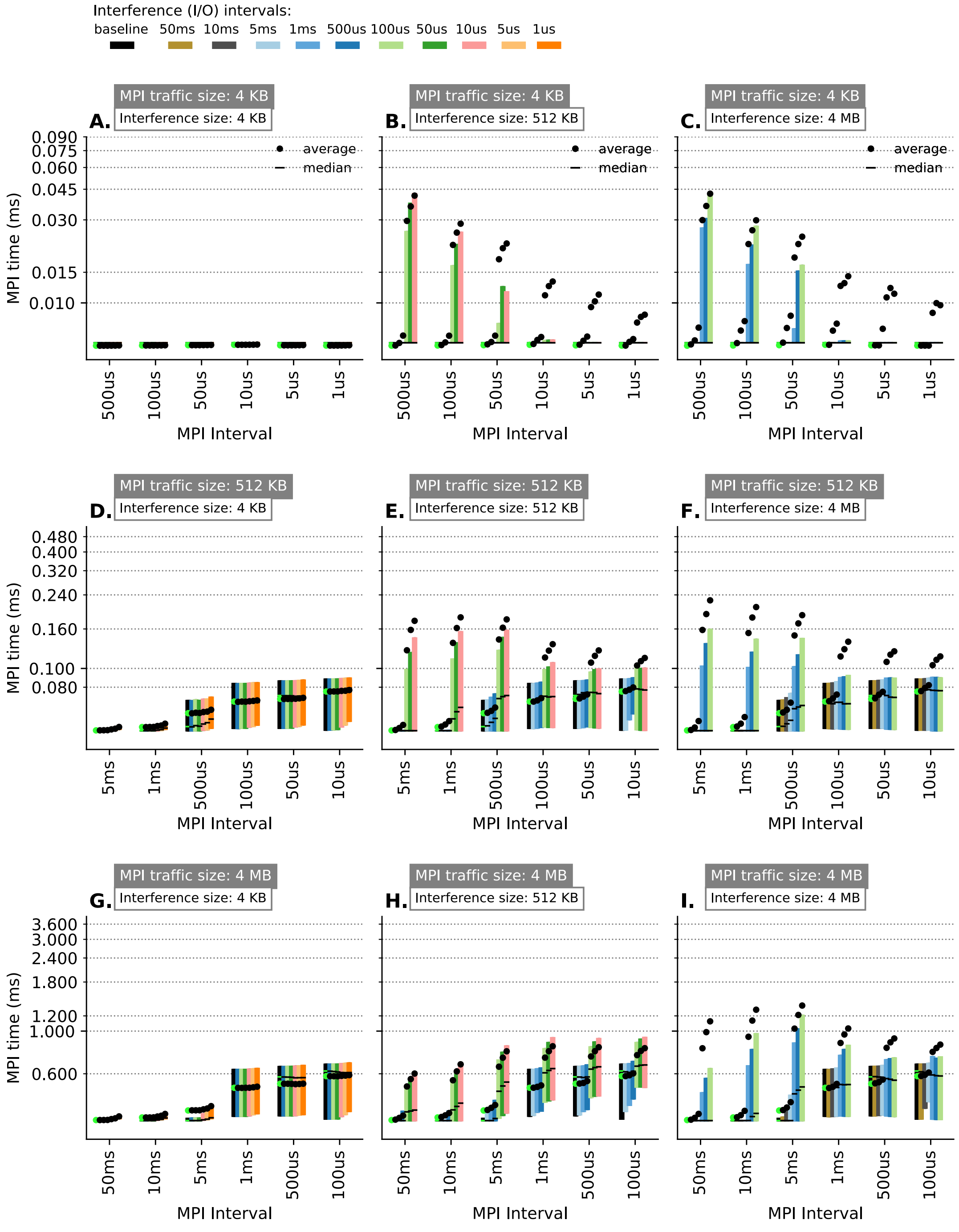


Figure 5.5: Performance of three MPI message sizes being interfered by three I/O request sizes, one traffic-interference pair per chart. A difference in the height of the features of the colored bars (interference runs) relative to the features of the black bar (baseline run) in the same group indicates a performance variation due to interference.

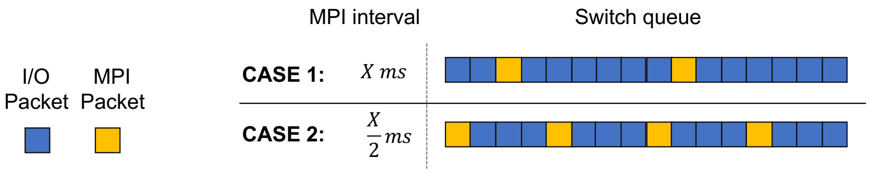


Figure 5.6: Expected state of switch queues when MPI interval is reduced by half.

to the low 10 μs I/O interval. This phenomenon is visible in all charts where the I/O interference size is 512 KB (charts B, E, F) or 4 MB (charts C, F, I). Higher I/O injection rates and larger I/O request sizes cause more network congestion, as shown in previous subsection, resulting in more I/O packets waiting in the queues of network switches. By reducing the MPI interval, MPI packets are injected into the network more frequently and are en-queued more quickly, preempting a portion of the I/O packets. With less I/O packets ahead of MPI packets, MPI packets spend less time queued on the network and are transferred faster, results in overall better MPI performance in-spite of the high interference. Figure 5.6 illustrates the expected change in state of a the switch queue when the MPI interval is reduced while the links are congested with I/O traffic.

Detailed View of Interference Trend

To get a more detailed view of MPI performance characteristics, the case where 4 MB MPI messages are communicated alongside 4 MB interfering I/O requests is presented Figure 5.7. The results in this figure report that when MPI messages are sent infrequently, the impact of I/O request is highest: all MPI intervals larger than 4 ms experience a maximum slowdown above 300% of the baseline times when the I/O interval is 100 μs. The other significant slowdowns are caused by I/O request arriving at intervals of 2 ms and smaller. This corresponds to the I/O-congestion threshold observed in Figure 5.4.

Overall, the results from subsections 5.4.1 and 5.4.2 can be summarized as follows:

- For small interfering messages (4 KB of MPI or I/O), the impact on performance is minimal for both cases.
- For I/O jobs, the impact of interference is moderate if the I/O interval is larger than the I/O-congestion threshold. Below the I/O-congestion threshold, interference has negligible effect for all I/O request sizes.
- For MPI jobs, interference always cause performance variations when there is a moderate amount of interfering I/O traffic on the network. The most significant interference is caused when the I/O traffic has passed its I/O-congestion threshold.
- Overall, the impact of interference is higher for MPI jobs than for the I/O jobs.

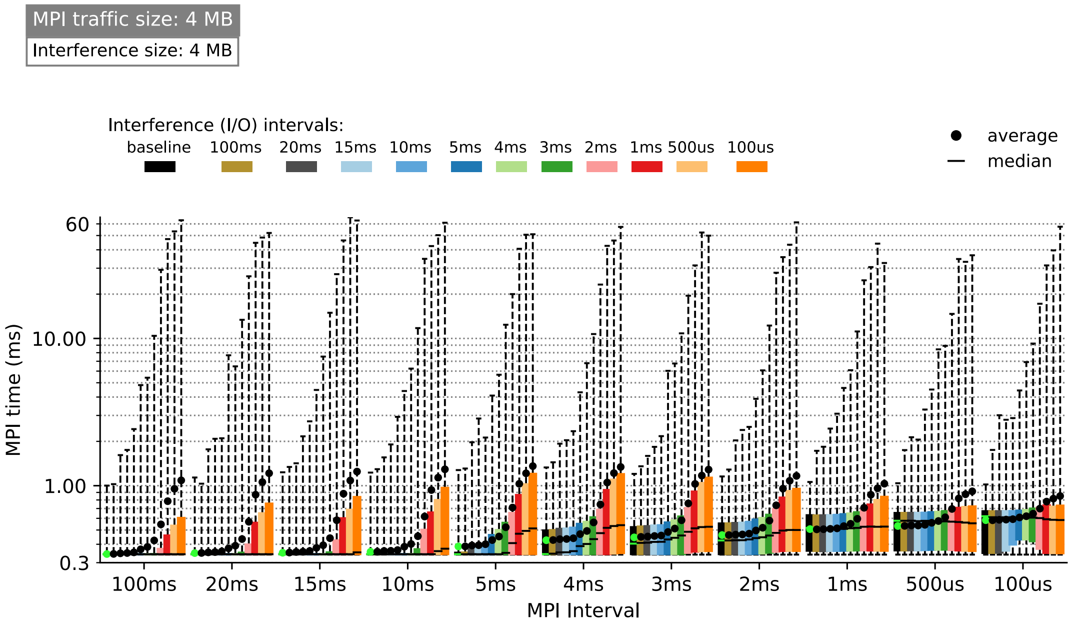
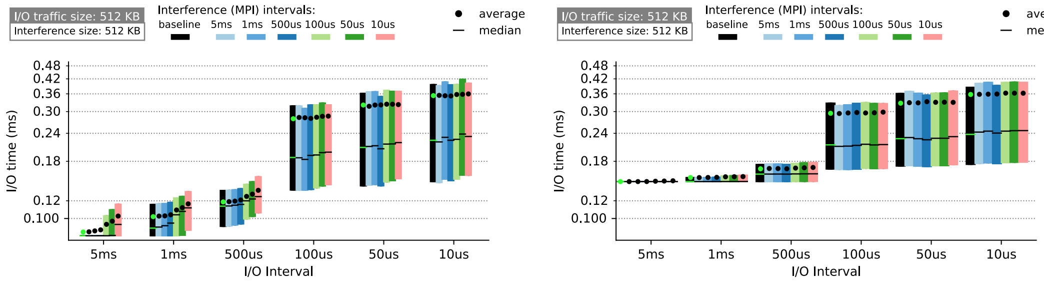


Figure 5.7: A detailed look at the performance of 4 MB MPI messages.



(A) Static Fat-tree routing

(B) Adaptive routing

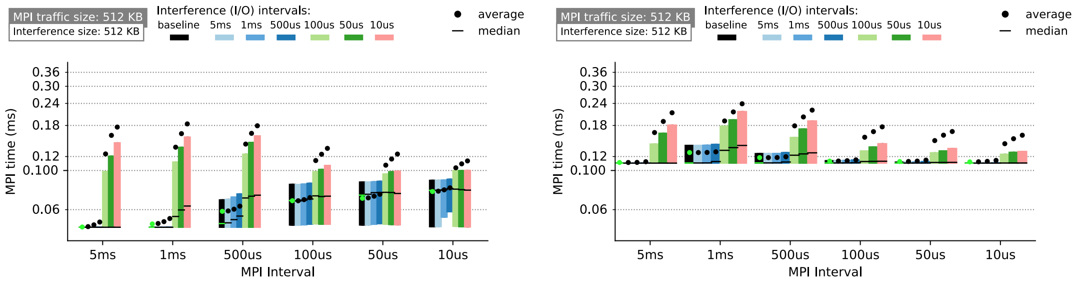
Figure 5.8: I/O performance with different routing algorithms

5.4.3 Static versus Adaptive Routing

The routing policy of the network affects the distribution of application traffic and how it will interact with other traffic that share the network. All previous results have been generated from a network that uses the static fat-tree routing algorithm. This subsection, however, shows the interference when the fat-tree network uses adaptive routing.

Adaptive routing involves each packet being routed to avoid congestion in the network. As described in Section 4.2 of Chapter 4, typical fat-tree networks have multiple redundant paths between pairs of nodes that are not connected to the same leaf switch. Essentially, adaptive routing algorithms attempt to balance the traffic load across the network by using alternative paths when the primary paths are congested.

The 512 KB I/O traffic versus 512 KB MPI traffic experiment from Section 5.4 is repeated with adaptive routing instead of static routing. A comparison of the I/O results are shown in Figure 5.8 and MPI results are shown in Figure 5.9. The



(A) Static Fat-tree routing

(B) Adaptive routing

Figure 5.9: MPI performance with different routing algorithm

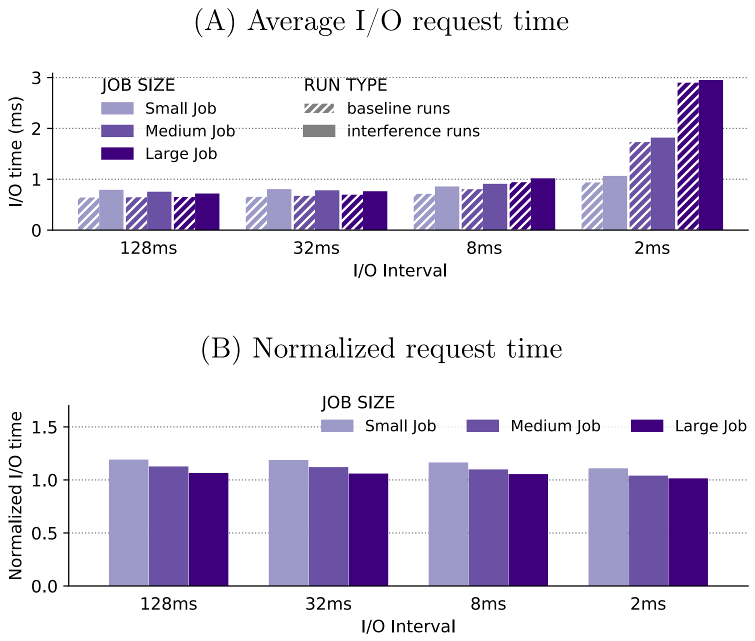
first point of note is that adaptive routing causes higher baseline latencies than static routing. The exact cause for the higher baseline latencies with adaptive routing is unknown at the point of writing this document, however, Requena et al. [2007] demonstrated that some communication patterns perform better with static routing than with dynamic routing. This issue will be the focus of future studies.

The I/O results in Figure 5.8 indicate that adaptive routing effectively eliminates the slowdown due to interference. The rise in colored bars for the first three cluster of bars in Sub-figure 5.8A (5 ms, 1 ms, and 500 μs I/O intervals) does not exist in Sub-figure 5.8B. However, the I/O-congestion threshold remains when network's routing is changed to adaptive routing. Furthermore, Figure 5.9 report that MPI messages experience no slowdown due to interference when the I/O traffic interval is not below the I/O-congestion threshold, i.e. when the I/O interval is not 100 μs or lower.

In summary, adaptive routing in this configuration increases the overall message/request latencies while eliminating the slowdown due to interference prior to the I/O-congestion threshold. Below the I/O-congestion threshold, I/O traffic and MPI traffic display similar performance characteristics for both static and adaptive routing.

5.4.4 Job Scaling

The results presented in the previous subsections are for the system configuration with the compute nodes being evenly split between I/O and MPI clients. This configuration is useful for making a fair comparison between how each traffic class reacts to interference from the opposing traffic class. However, nodes are not usually partitioned this evenly in practice. Next, results are presented for a set of experiments in which the job sizes are alter in order to understand how a job's scale affects the interference it experiences. Three node allocation sizes are tested: *small*, *medium*, and *large* using 25%, 50%, and 75% of the compute nodes for the analyzed job, respectively. All compute nodes in the system are occupied by either the I/O job or the MPI job, and a reduction in the size of one job means an increase in the size of the other job. The medium jobs discussed in this subsection use the same configuration as the jobs in the previous subsections


(A) Average I/O request time

(B) Normalized request time

Figure 5.10: Impact of job sizes on I/O performance. Remaining nodes are running a MPI job.

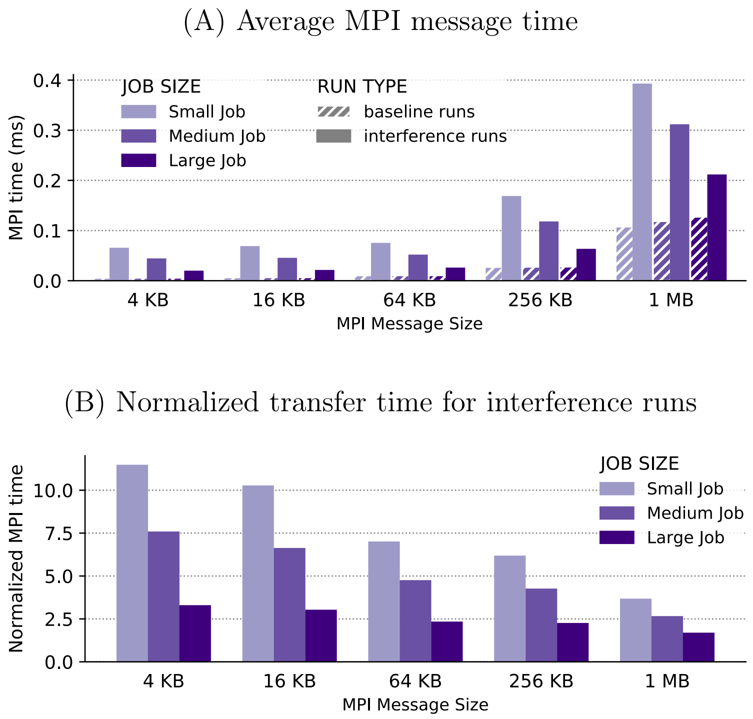
and report the same performance.

Based on the I/O performance trends presented in Section 5.4.1, checkpoint-style I/O traffic, which have request sizes in megabytes and intervals close to zero, are not affected by MPI interference because of having much lower intervals than its I/O-congestion threshold. Therefore, I/O performance experiments in this subsection use I/O intervals that show distinct runtime perturbations in the presence of MPI interference. A request size of 4 MB and intervals between 2 ms and 128 ms are used, which are more susceptible to interference than checkpoint-style traffic. Similar patterns can be found in various applications, such as the FLASH simulation [Flash Center for Computational Science, 2017], that supports controlling the frequency at which output files are written to the file system.

The MPI performance evaluation is done for MPI messages between 4 KB and 1 MB with a 500 μs interval in the presence of worst-case I/O traffic, i.e. checkpoint-style traffic. These MPI traffic patterns represent message characteristics of diverse communication-bound MPI applications [León et al., 2016; Jain et al., 2017].

I/O Jobs

The I/O request times are presented in Figure 5.10A and the resulting interference trends are shown in Figure 5.10B. The baseline times reported in Figure 5.10A for 2 ms-interval requests show that the I/O requests in medium and large jobs are approximately 1.8× and 3× higher than the time for the small I/O job, respectively. Increasing the job's size results in more I/O clients requests being sent to the 72 I/O servers, thereby increasing the self-congestion and mean time



(A) Average MPI message time

(B) Normalized transfer time for interference runs

Figure 5.11: Impact of job sizes on MPI performance. Remaining nodes are running a I/O job.

per request.

An increase in self-congestion means that the I/O-congestion threshold will be reached at larger intervals when the job size is increased for a given I/O traffic pattern. The baseline request time for the 2 ms-interval traffic in the this large job is actually equivalent to the baseline time of 100 μs-interval requests in the medium job (see Figure 5.4). The large job is least affected by MPI interference because the performance degradation due to self-congestion supersedes any effect of interference from MPI traffic. Smaller jobs with more infrequent requests are more sensitive to interference. The most significant relative increase in I/O request time due to MPI traffic is 18% and occurs when the the small job sends requests at 128 ms intervals.

MPI Jobs

The performance trends in Figure 5.11 indicate that MPI messages are substantially affected by I/O traffic for all message sizes and job scales. The slowdown due to interference reported in Figure 5.11B shows that, for all message sizes, relatively lower interference is seen for the small MPI jobs than with the medium and large jobs. The fewer I/O clients that compete with the MPI job large means that there are less I/O packets being injected into the network and that the path of I/O packets are less likely to overlap with the path of MPI traffic. Nevertheless, even with large MPI job, the slowdown due to I/O interference ranges between 1.6-3.2× of the baseline performance. Figure 5.11B also shows that MPI jobs with small message sizes are affected the most by I/O interference.

Figure 5.11A indicates that the baseline times for a given message size across all jobs scales do not significantly deviate from each other. Therefore, the difference in interference times for that message sizes is due almost entirely to the I/O interference. The slight increase in baseline times of the 1 MB messages for larger scales reveals an increase in congestion due to MPI. It is a more intense degree of this congestion that produced the slowdown in I/O performance that was reported in the I/O jobs in this subsection.

5.5 Discussion

Difference between I/O and MPI Traffic Patterns

In analyzing the differences between the I/O and MPI interference trends, it must be re-stated that there is a key difference between the traffic patterns of the two jobs: each MPI client has a single, unique destination while all I/O clients write data to the same set of few I/O servers in a round-robin manner. In our environment, there is a 1:1 mapping from MPI client to its pair and a 612:72 mapping from I/O clients to I/O servers. Thus, the MPI traffic is dispersed more evenly across the network while I/O traffic is focused at switches servicing I/O servers, causing congestion where the paths from the different I/O clients converge. This disparity is the reason for the following phenomena:

- the average baseline time of a given I/O request size is higher than the average baseline for an MPI message of the same size and moderate interval
- the MPI congestion is not as extreme as the I/O congestion, which is primarily responsible for the severe I/O performance degradation beyond the I/O-congestion threshold

Interference Sensitivity and Interfering Potential

The interfering potential and sensitivity of each traffic class can also be attributed to the difference in their traffic patterns. I/O congestion not only slows I/O traffic, but also delays MPI packets that traverse the paths where I/O packets are concentrated. Since I/O packets spend longer time over the network, there is a higher probability that they will block MPI packets. The MPI congestion is not as extreme, but it is shown that it has the potential to slowdown I/O requests when the MPI traffic is relatively intensive. This was reported in chart C of Figure 5.3 with small I/O request size versus large MPI message size and in Figure 5.10 with small I/O job size versus the large MPI job size; the latter being a common situation on HPC systems.

I/O-congestion Threshold and Job Sizes

Figure 5.3 in Section 5.4.1 illustrated that different I/O request sizes have different I/O-congestion thresholds. Considering that 4 MB I/O request traffic reaches its I/O-congestion threshold at 2 ms intervals for the medium job, the results in

Figure 5.10 indicate that 4 MB I/O traffic pattern has passed its I/O congestion threshold at 2 ms for the large job while the small job has not reach this threshold for the same interval. In other words, larger jobs will reach their congestion thresholds at higher intervals than smaller jobs for a given message size.

Performance Distribution due to Interference

Both I/O and MPI results exhibited average interference run values that have large ranges with mean values significantly higher than their respective medians. Figure 5.3C and Figure 5.5C illustrates examples of these runs. This feature a indicates heavy tail distribution of times and is an issue where quality-of-service (QoS) is concerned.

5.5.1 Implications for Communication Optimization

The results of this interference characterization has several implications for managing and optimization communication performance in a mixed workload environment. The major findings of this study are restated here along with their implications and suggested guidelines for attaining good performance.

FINDINGS

- I/O-congestion threshold varies with request size and job size.
  - Congested I/O traffic is virtually unaffected by interference.
- MPI self-congestion doesn't prevent slowdown due to interference.
- Small I/O jobs can experience up to 18% slowdown due to interference.
- When competing with high-intensity I/O traffic, increasing the MPI traffic intensity can improve.
- The general interference trend are shown in Figure 5.12.

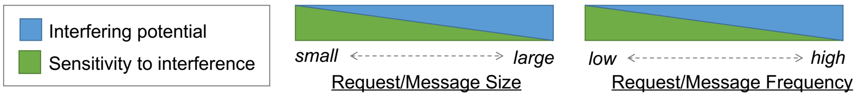


Figure 5.12: An illustration of the interfering potential and interference sensitivity of communication sizes and frequencies. *Note: This figure illustrates general trends and not precise performance results. The changes in potential and sensitivity are not expected to be linearly proportional.*

IMPLICATIONS AND GUIDELINES

- Application developers should consider the opposing workload when implementing application communication patterns.

- System administrators should consider the difference in interfering potential of each job when configuring job schedulers.

- To improve the performance of I/O request that suffer from I/O congestion without changing the traffic pattern, changes should be made to the traffic distribution in order to avoid congested links. This could involve adjusting the node placement or network routing.

- Since the large portion of HPC workloads is usually MPI jobs, usual I/O traffic will potentially be impacted by MPI interference.

- Small reduction in I/O interference can lead to significant reduction in MPI slowdown.

Chapter 6 investigates these implications in more detail.

5.6 Summary

Interference due to resource contention over the network can be a major issue for jobs whose performance depend on efficient inter-process communication and/or I/O operations. To understand the interaction of these two types of communication sources, the performance characteristics of MPI and I/O traffic on a fully provisioned fat-tree network were investigated. The investigations were done using CODES-based simulations with synthetics traffic patterns that are representative of real HPC workload traffic patterns. The differences in the performance trends exhibited by the MPI and I/O traffic with and without interference were studies to reveal their sensitivity to interference.

The characterization of the I/O-MPI interference on fat-tree network proves that I/O traffic is less sensitive to interference than MPI traffic. This is because the network congestion caused by high-intensity I/O workloads negatively affects I/O performance and renders the interference from MPI traffic inconsequential. Nevertheless, the results show that intensive MPI jobs can slow the performance of the lowest-intensity I/O traffic that was studied by up to 1.9×. For a more typical mixed workload on modern HPC system, the I/O job experiences 18% slowdown. MPI traffic does not suffer as badly from self-congestion and is more sensitive to the effect of I/O interference. The largest slowdown for MPI traffic was 7.6×.

For I/O traffic, we identified the presence of I/O-congestion threshold in various scenarios, i.e., the point at which the frequency of I/O request is so high that I/O packets congest the network and degrades its own performance more severely than interference from MPI. This threshold varies based on the size of the request and the scale of the I/O job, and it was discovered to be an important cause of both I/O performance slowdown and MPI performance slowdown.

Chapter 6

I/O–MPI Interference Mitigation

This chapter evaluates the efficacy of three communication optimization strategies that are targeted at mitigating the performance impact of the I/O-MPI interference on fat-tree networks. Knowledge of the architecture of fat-tree networks and the mixed-workload interference patterns, from Chapter 5, are used to guide the optimizations of the I/O and MPI workloads. In particular, the following three interference mitigation strategies are discussed:

Relocation the I/O servers: placing the I/O servers on different switches.

Placement of jobs to nodes: varying the allocation of nodes to jobs in order to have jobs placed at different locations on the network.

Throttling I/O traffic: controlling the flow of I/O requests issued from each I/O client in order to reduce the congestion caused by I/O traffic.

6.1 Overview of the Problem

A characterization of the impact of I/O-MPI interference on the performance of MPI and I/O jobs running on fat-tree networks has been developed and presented in Chapter 5. MPI jobs have exhibited performance increases in average message times of up to 7.6× and I/O jobs have shown increases in average request times of up to 1.9× due to I/O-MPI interference. The degradation in performance reduces user productivity as well as overall system throughput, and the nature of the interference suggests limitations to which optimizations will be effective (see Section 5.5).

I/O and MPI studies have detailed numerous strategies for improving I/O performance or MPI performance, as discussed in Section 2.3. However, with the exception of Mubarak et al. [2017a] work for dragon fly networks, other works limit their study to only interference from jobs in the same class (I/O or MPI) and do not examine the effects of these strategies in reducing the I/O-MPI interference. Most HPC centers run mixed workloads on their supercomputers, so failure to consider the influence of communication optimizations on I/O-MPI interference signify that the optimizations may have unknown effects in a production environment. Currently, no studies have been found to guide communication optimizations for mitigating the I/O-MPI interference on fat-tree networks.

6.2 Mitigation Considerations

Designing appropriate optimization strategies requires careful understanding of the cause of the interference. The I/O-MPI interference occurs when the congestion of one job affects the traffic the another job. From the topology aspect, a full bisection bandwidth fat-tree can provide congestion-free routing. However, if the points at which traffic is being injected/consumed are not appropriately aligned on the network, paths will overlap and cause congestion. Hence the I/O-MPI interference can be address by aligning jobs to ensure inter-job congestion-free routing for each other.

For cases when it is not possible to change the placement of jobs, consideration has to be given to altering the traffic patterns. The I/O-congestion threshold indicates the point where I/O traffic becomes most insensitive to interference while simultaneously having its highest interfering potential. The performance of I/O requests is not easily improved nor is it easily degraded when changing the I/O traffic pattern about this point. However, MPI traffic is highly sensitive to interference from this type of I/O traffic; the MPI latency experiences substantial variations due to slight changes in the pattern of congested I/O traffic. Therefore, a compromised may be reached to significantly reduce I/O interference for the cost of a tolerable reduction in performance by throttling the I/O traffic.

6.3 Review of Interference Mitigation Strategies

6.3.1 Job Placement

Job placement refers to the deliberate allocation of compute nodes to an application in order to have the application's processes arranged in a pre-specified manner on the network. Some placement strategies attempt to put processes on nodes that are physically close to each other in order to reduce the number of hops between nodes for lower latency communication, while some placements try to spread nodes across the network in order to access more communication pathways.

Past works hae shown that the placement of jobs impacts self-congestion and interference observed on a system and thus can affect job performance [Mubarak et al., 2017a; Yang et al., 2016; Zahavi, 2011]. This is because placement determines the nodes at which data is injected/consumed in the network, which, along with the routing scheme, impacts how the job's traffic is spread across the network links.

6.3.2 I/O Server Placement

I/O server placement has the same importance as job placement. Traffic is consumed by I/O servers when I/O clients write to the parallel file system, and traffic is injected into the network when clients read from the parallel file system. I/O servers are often placed in a systematic manner that ensure manageability along with ensuring good performance for a range of I/O workloads [Oral et al., 2014]. Some placements include, grouping all servers in one part of the network and distributing server uniformly among some element in the network: switches, pods, cabinets, clusters, or scalable units.

The I/O servers of large scale parallel file systems like Luster [lus, 2018], GPFS [B. Schmuck and Haskin, 2002], and PVFS [Ross and Latham, 2006] must be configured at installation time and cannot be relocated after installations. The optimal placement must be determined before the system is started by the administrators. On-demand file systems such as CRUISE [Rajachandrasekar et al., 2013], BurstMem [Wang et al., 2014], and HuronFS [Xu et al., 2018] allow users determine which nodes in the system will function as I/O servers at runtime. The user can choose the optimal server placement based on their individual application requirements.

6.3.3 I/O Throttling

Throttling I/O traffic refers to the process of regulating the rate at which I/O requests are sent or processed in order to manage congestion on the clients, links, or servers. I/O throttling has been presented as a solution to reducing I/O congestion and preventing low-priority I/O workloads from monopolizing the system's I/O bandwidth [Qian et al., 2017; Liu et al., 2013]. There are many approaches to throttling, but all approaches result in varying the rate of I/O requests. The characterization of I/O performance in Section 5.4.1 showed how I/O congestion can be more detrimental to I/O performance than interference caused by MPI traffic. Furthermore, the interference caused by I/O congestion can lead to over 7× slowdown in MPI message performance. Reduction in I/O congestion would potentially lessen this detrimental slowdown in MPI performance.

6.4 Methodology

Evaluations are conducted on the 1,296 node simulated fat-tree network described in Chapter 5. The following configurations are used with jobs to report the results obtained in this chapter:

> **I/O performance:** 4 MB I/O requests sent at 128 ms intervals with background interference from 4 MB MPI messages sent at the interval of 500 μs.
>
> **MPI performance:** 4 KB MPI messages are sent at 500 μs intervals with background interference from checkpoint-style I/O, i.e. 4 MB request with a negligible interval between requests.

The jobs are executed in a similar manner as described in subsection 5.3.4 with configuration modifications based on the mitigation strategy being evaluated. That is, approximately 1000 data points are recorded for each client of the job being studied with 50 warm up messages/requests being sent before recording begins. The actual interval between each message/request is varied randomly with a maximum bound of ±5% of reported interval. For each case, a baseline

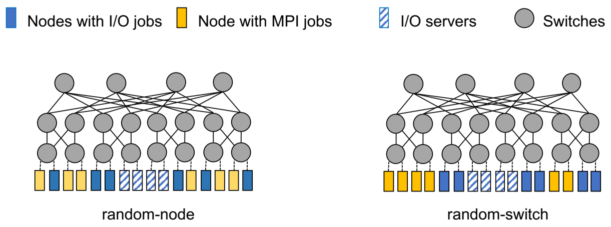


Figure 6.1: Illustrations of different job placement configuration.

run is done of the job being studied without any other traffic running in the background. The interference runs are done for the same job with interfering traffic running in parallel in order to measure how the interfering traffic affects the performance of the job being studied.

6.5 Evaluation

6.5.1 Job Placement

All results presented in Chapter 5 used a topology-oblivious job placement in which application tasks are randomly mapped to compute nodes across the network, as is done on some production systems. This allocation will be referred to as the *Random-node* placement. The performance of jobs under the Random-node placement is compared against the performance of jobs under another placement: the *Random-switch* placement. The placement are described as follow:

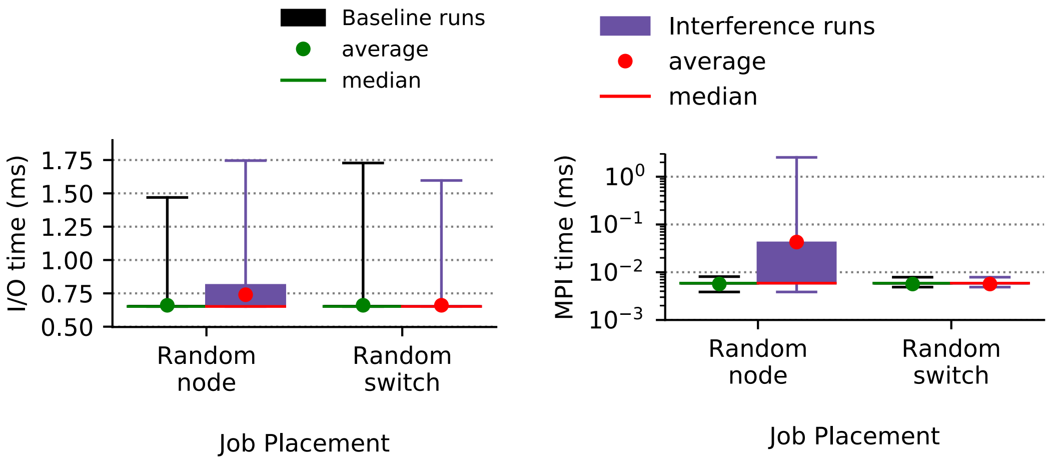
- **Random-node**: nodes are randomly allocated to one of the two jobs regardless of the location of the nodes on the network.

- **Random-switch**: nodes connected to the same switch are grouped together, and each group is randomly assigned to one of the two jobs. The tasks within a job are mapped sequentially to the nodes on the assigned switches.

Results

Figure 6.2A compares the I/O performance for executions with and without I/O-MPI interference for the Random-node and Random-switch job placement schemes. The results show that the average I/O request times for the baseline and interference runs are similar for the Random-switch placement, indicating that there is minimal slowdown due to MPI interference with this type of allocation. In contrast, the Random-node placement exhibits an 11.8% increase in the average I/O request time and the standard deviation increases from 8% to 19% when MPI-I/O interference is present.

The maximum values are not scrutinized or discussed in much details since they represent the performance of a single request/message and are largely probabilistic.



(A) Performance of I/O jobs

(B) Performance of MPI jobs

Figure 6.2: Impact of job placement placement on performance

The impact of these job placement schemes on the performance of the MPI job is shown in Figure 6.2B. It can be seen that with Random-switch placement, the impact of I/O traffic on MPI traffic is negligible. This is in sharp contrast to the default Random-node case in which an average slowdown of 800% is observed for MPI messages.

Analysis

A visual comparison of the network traffic for the placement schemes indicate that these improvements are obtained for the Random-switch scheme because all links connected to switches with I/O servers and clients carry only I/O traffic; the same assertion holds for switches with MPI clients as well. Figure 6.3 shows the traffic for the I/O job with random switch placement. For clarity, this figure shows a 512-node network instead of the 1,296-node network used to generate the aforementioned results. The visualization of the I/O traffic for the random-node placement on 1296-node network is shown in Appendix D.

The I/O-MPI interference on links between level-1 and level-2 switches is avoided since I/O traffic is isolated to the switches assigned to I/O jobs, and therefore, the links connecting those switches will carry only I/O traffic. Furthermore, for the commonly used fat-tree routing algorithm, which is used in our network, dedicated ports of level-3 switches are responsible for forwarding traffic for a given level-1 switch. Thus, the links connecting to the level-3 switches that are used to relay I/O traffic do not carry MPI traffic since MPI clients do not share leaf-level switches with I/O servers in this placement scheme. The Random-switch placement provides complete isolation of I/O traffic from MPI traffic, resulting in no I/O-MPI interference over the network.

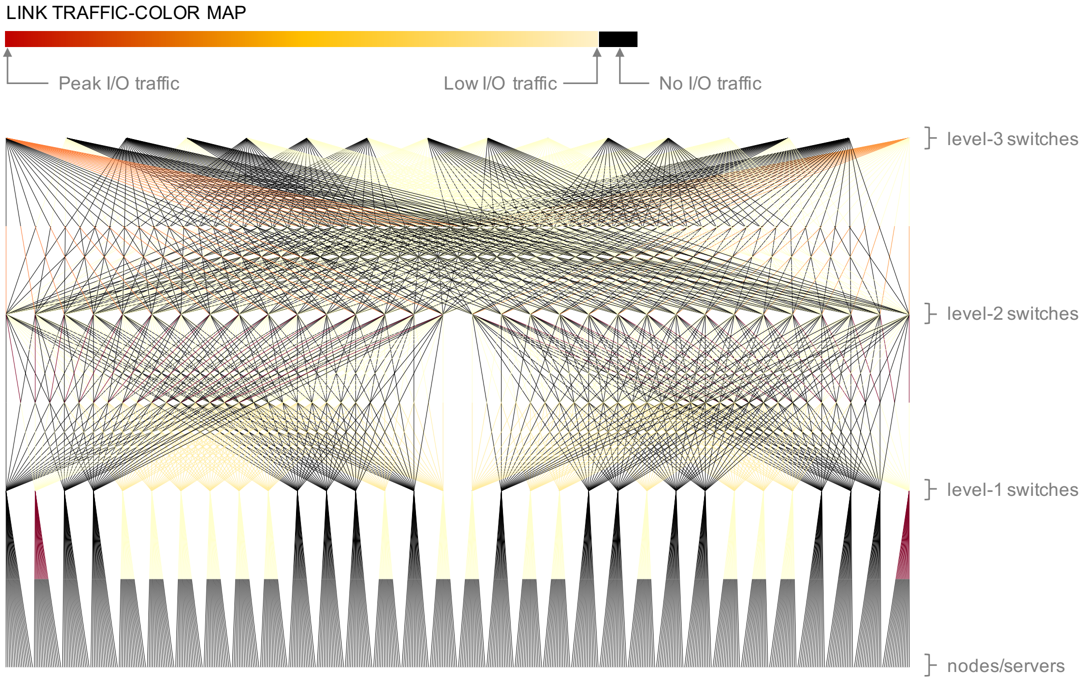


Figure 6.3: The distribution of I/O traffic under the Random-switch job placement on a 512-node network.

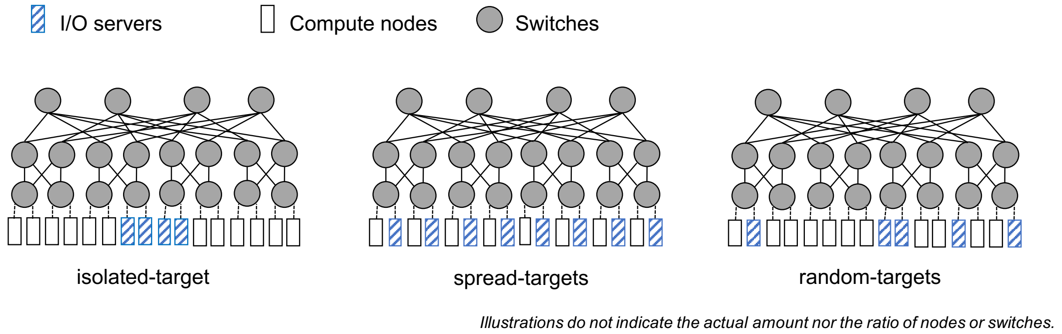
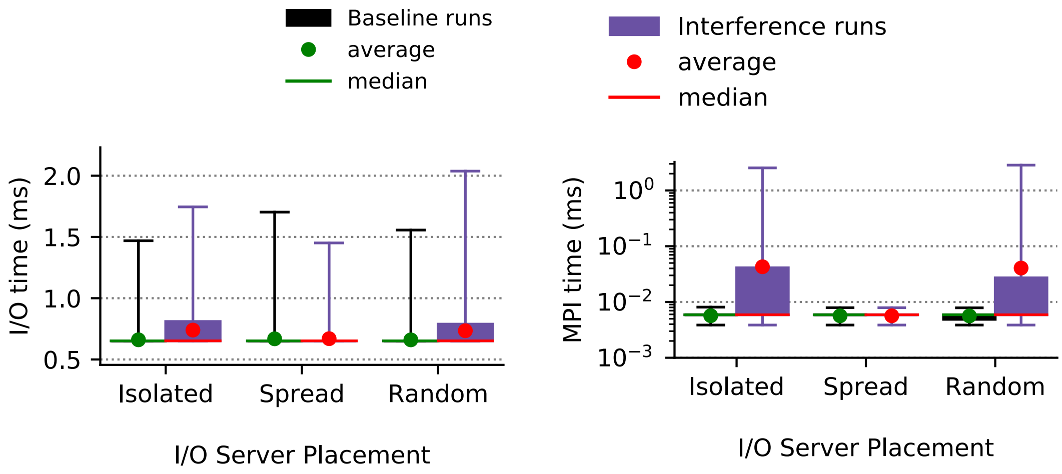

I/O servers
Compute nodes
Switches
isolated-target
spread-targets
random-targets
Illustrations do not indicate the actual amount nor the ratio of nodes or switches.

Figure 6.4: Illustrations of different I/O server placement configuration.

6.5.2 I/O Server Placement

During the analysis of I/O performance and I/O-congestion threshold in Chapter 5, the grouping of I/O servers on the same switches resulted in heavy usage of a few down links connected to different level-2 switches in the network. This created congestion throughout the system and impacted performance significantly. The second mitigation strategy explores relocating the I/O servers using the following placement schemes:

- **isolated-target**: In this scheme, dedicated switches are used to host I/O servers. This I/O server placement scheme has been used for all other experiments in this work. For the 1296-node simulated system, four leaf switches – two from each half of the network – are dedicated to I/O servers.
- **spread-target**: In this scheme, I/O servers are assigned in a round-robin



(A) I/O performance

(B) MPI Performance

Figure 6.5: Impact of job placement on traffic performance.

fashion to the switches. For our system, one I/O server is assigned to each of the 72 leaf switches. Each I/O server is connected to port 18 of a switch.

- **random-target**: I/O servers are randomly distributed across the system with no restrictions.

The schemes are illustrated in Figure 6.4.

Regardless of the I/O server placement scheme, I/O clients always use the same I/O server selection policy. That is, I/O clients always uses a round-robin policy to select the I/O target that will receive its next request. This policy is commonly used in production systems and the default for Lustre clients. The I/O server selection policy is not varied in order to ensure that our results are consistent and allow for a fair comparison. The Random-node placement was used to assign jobs to clients in these experiments.

Results

Figure 6.5 shows the impact of MPI-I/O interference on performance when the I/O server placement is varied. The spread-target placement of I/O nodes reports the lowest average I/O request times and MPI message times; it performs the best at mitigating interference for both types of jobs. The isolated-target and random-target server placements result in similar slowdown for in the MPI case and the I/O case.

Analysis

I/O traffic over the network for the isolated-target scheme causes heavily used links throughout the system. The use of random-target scheme does not improve the situation since the I/O traffic still heavily uses few down links connected to different level-2 switches. However, when the I/O servers are connected to the 18^{th} port of every level-1 switch in the spread-target scheme, it is observed that

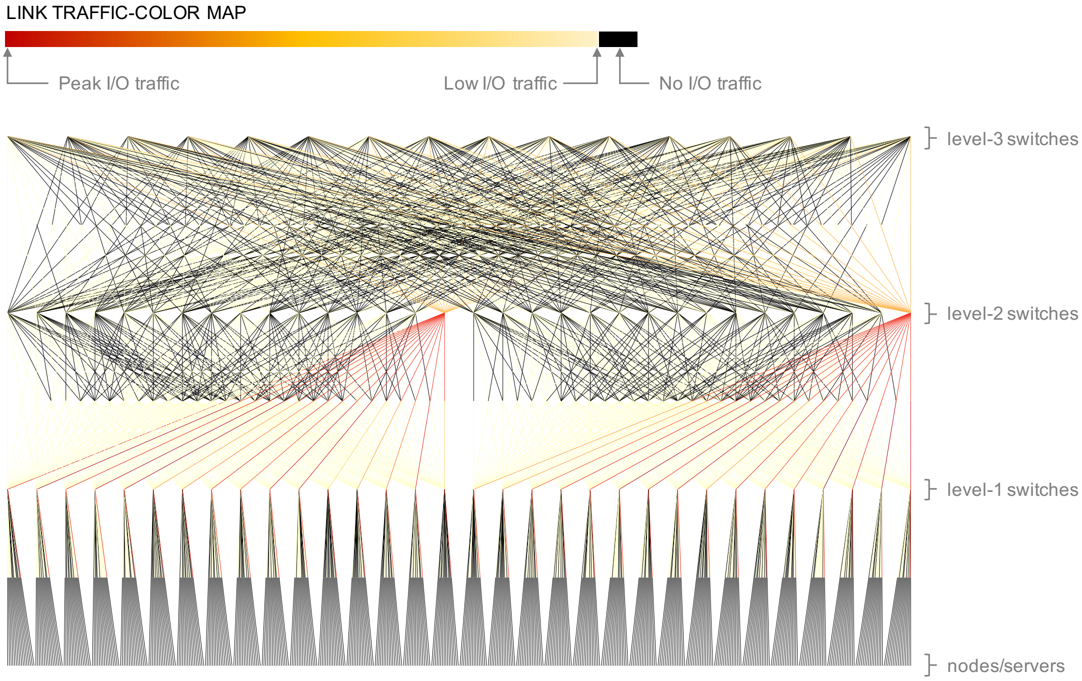


Figure 6.6: The distribution of I/O traffic under the Random-switch job placement on a 512-node network.

only four of the 72 level-2 switches (one in each pod) are used for sending down I/O traffic to I/O servers. By parsing the routing tables of all switches, it was confirmed that that the traffic destined to the servers connected to the 18th port of every leaf switch is routed via the 18th level-2 switch in the pod of the destination server. As a result, the I/O traffic is confined to a subset of the network and thus does not interfere with the MPI traffic.

Figure 6.6 shows the traffic for the I/O job with spread-target placement. For clarity, this figure shows a smaller network than the 1,296-node network used in our experiment, while the visualizations of the 1296-node network is provided in Appendix D. The red links in this visualization show that paths used by I/O traffic between level-2 and level-3 switches.

6.5.3 Throttling I/O Traffic

The random-node job placement and isolated-target I/O server placement are used to study the effects of I/O throttling on the performance of both I/O and MPI jobs. Each I/O client writes 4.3 GB of total data, which is broken up into 4 MB requests and sent to I/O servers in a round-robin manner. This operation is analogous to writing checkpoint data to disk, whereby the total checkpointing time is more important than the time to send individual 4 MB requests. In addition to measuring the individual time of each I/O request, the time taken for each I/O client to complete writing its entire payload is recorded. I/O throttling is achieved by varying the time between initiating consecutive I/O requests. This gap is referred to as the *throttle window*. The *throttle window* represents the duration between the start times of two consecutive requests. Note that this is

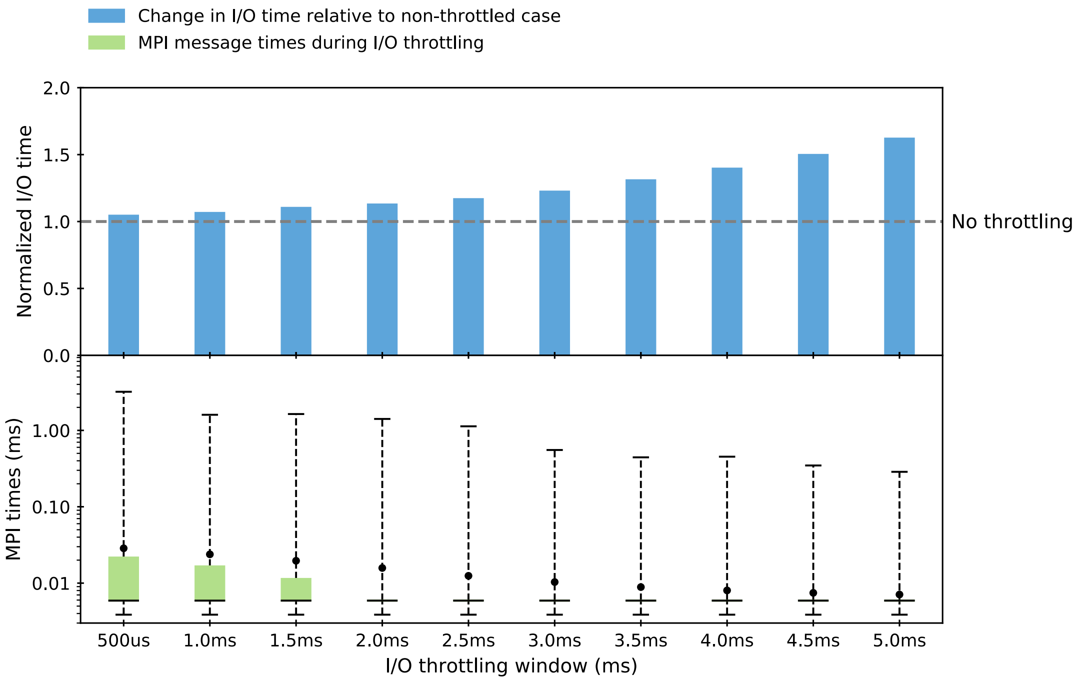


Figure 6.7: Impact of I/O request throttling on I/O and MPI performance.

different from request *interval*, which is used to represent the duration between the completion time of a request and the start time of the next request. The MPI traffic consist of 4 KB messages being sent at 500 μs intervals.

Results

In Figure 6.7, the baseline I/O performance is time taken for all I/O clients to finish their data writes without any I/O throttling and with the MPI job running in the background. The I/O completion time increases gradually as the throttling window is increased. The average MPI performance also improves gradually with large throttling windows, with a visible reduction in the distribution of times occurring at 2 ms. With 2 ms throttling, the I/O time increases by only 11.2% while the average MPI's latency improves by over 200%. Table **??** shows average I/O request times for the different throttling windows.

Analysis

The average I/O time becomes less than the throttling window for window duration of 2 ms and above, and the average time between successive I/O requests becomes 1 ms for a throttling window of 2.5 ms. For 4 MB I/O requests, the 1 ms interval is around the point of the I/O-congestion threshold where I/O traffic starts to saturate the network and, hence, the improvement in MPI performance and reduction in I/O request time as the interval starts to leave the most congested range. Once the throttling window results in the mean interval between request rising above the I/O-congestion threshold, the degradation of MPI performance begins to lessen. This trend was confirmed in Chapter 5

6.6 Discussion

Node allocation on shared systems does not always ensure users can achieved their desired job placement. Multiple users with varying request often cause partitions that don't align with switch boundaries. However, the system administration may enforce scheduling policies that try to allocate contiguous nodes and place them in fragments that observe the switch boundaries. In the cases where the placement of nodes and servers can be controlled, the results of this interference study can be applied in several ways. Administrators can use the lessons from these simulations and similar experiments to determine the optimal placement of I/O servers before deploying new systems as well as when re-configuring current systems; developers of on-demand file systems such as BurstMem [Wang et al., 2014] can also use these findings as a guide to managing the I/O-MPI interference; and users of on-demand file systems can request nodes and that are located in positions that reduce interference. These optimizations may be tailored for improving the performance of certain application or improving overall system throughput.

Throttling delays the sending I/O requests and does incur an increase in the job's runtime, so it must be applied carefully to ensure that a meaningful trade-off between reducing the interference and increasing the I/O job time. These results indicate that I/O throttling can potentially be used to reduce MPI-I/O interference if a relatively small loss in I/O performance is acceptable. By using the I/O-congestion threshold for the I/O traffic pattern, the throttling window may be chosen that can yield worthwhile improvements in MPI performance while incurring tolerable slowdown in I/O traffic.

6.7 Summary

In order to mitigate the impact of MPI-I/O interference, an evaluation of three different optimization strategies was conducted. These strategies were (i) varying the job placement on the system, (ii) varying the I/O server placement, and (iii) throttling the I/O traffic. The two placement strategies were guided by knowledge of the fat-tree network's ability to provide congestion-free routing, and the I/O throttling strategy was guided by the interfering characteristics of congested I/O traffic.

It was confirmed that each of these strategies can reduce interference and improve performance, especially for MPI jobs, which is more sensitive to the impact of I/O-MPI interference. By assigning all nodes of a given leaf switch to a single job, the MPI and I/O traffic were isolated from each other, thereby preventing interference. The placement of an I/O server on the last port of each leaf switch was equally successful in preventing interference between MPI and I/O traffic. Each of these two solutions was made possible by leveraging the network topology and routing scheme of the fat-tree network and the deterministic paths of I/O traffic from clients to I/O servers. The the throttling strategy, on the other hand, was able to reduce the I/O congestion by slowing the rate at which clients generate I/O request. I/O throttling resulted in an 18% increase in the

I/O job time while the average MPI message latency improved by over 200%. The choice of throttling window was be guided by the I/O-congestion threshold of I/O traffic being throttled.

Chapter 7

Summary, Conclusions, and Limitations

7.1 Summary

Ensuring good communication performance on large-scale supercomputers is challenging because of the complexity of these systems and the difficulties in identifying the nature of communication bottlenecks. Commonly used analysis methods are compartmentalized and fail to expose application traffic performance over the physical links of the network, the main location of communication bottleneck. Furthermore, there exists little support for this type of network-level communication performance analysis on fat-tree topologies.

7.1.1 Performance Measurement and Visualization

This work presents a method of measuring MPI traffic over network links on production systems in order to support communication analysis that incorporates both application and hardware information. The PERUSE utility of Open MPI is extended to support reporting communication events over the each network adapter used by the application. PERUSE provides an interface for tracking activities within the MPI library, so the extension made in this work was to track events when data is sent or received by the network adapter. A new profiler was created, called `ibprof`, to record data using the new PERUSE event and report the point-to-point traffic information within MPI operations. `ibprof` can be pre-loaded to an application and incurs less than 1% overhead with application benchmarks.

To analyze the performance data recorded by `ibprof`, a flexible fat-tree visualization process was developed. This process involves first automatically detecting the hierarchical structure of the network and encoding performance data onto the nodes and links of the network. The second step in this process is the visualization of the network and performance data using a newly created, interactive visualization module in the Boxfish tool. This module supports encoding performance data in the color of the nodes and network links, enabling the visual analysis of application traffic over the network topology. Case studies demonstrate that this method of visualizing `ibprof` profiles can identify network hot-spots and communication anomalies in MPI operations on production systems by overlapping profiling with data transmission.

7.1.2 Interference between MPI and I/O Traffic

The nature of the interference between I/O and MPI traffic was studied along with its impact on both I/O and MPI jobs. Simulation-based experiments were conducted to quantify and qualify the interference between these two workloads. Various traffic patterns with different messages sizes and communication frequencies exposed the different interference patterns that can occur when MPI and I/O jobs share the network.

The results of this interference study indicates that MPI traffic is more sensitive than I/O traffic. The average MPI message latency increased by up to 7.6× due to I/O interference in one study, while the maximum slowdown experienced by I/O traffic was 1.9× for the same study. It was discovered that I/O traffic has an I/O-congestion threshold, which limits the I/O performance as well as reduces the impact of MPI interference on I/O performance. This threshold is the interval between I/O requests, which affects how quickly the network becomes congested by I/O traffic. The I/O-congestion threshold is different for each request size and was shown to also vary based on the job size, where larger jobs have a higher congestion threshold. Due to this congestion threshold, I/O performance consistently degrades when the frequency of I/O requests increases. I/O performance degradation due to I/O congestion occurs regardless of the presence of MPI interference. On the other hand, MPI performance actually improves if messages are sent more frequently when the network is congested with I/O requests. This occurs because sending MPI messages more frequently causes MPI packets to enter switch queues before some I/O packets and, therefore, spend less time in the queues.

The impact of the I/O-MPI interference on application inspired the design of optimization strategies to reduce this interference on fat-tree networks. Two placement strategies, namely job placements and I/O server placement, were tailored to the fat-tree network's ability to provide congestion-tree routing. The job placement strategy was proven to be most effective when all nodes connected to a leaf switch belong to the same job, and the I/O server placement strategy was proven to be most effective when the I/O servers are placed on the last node of each leaf switch. Finally, the insights gained from the interference characterization was used to design an I/O throttling strategy that could improve MPI performance by over 200% while costing only 18% degradation of I/O performance.

7.2 Conclusions

The Need for Low-level, Low-overhead Communication Analysis

The layers of abstractions that hide the complexities and improve the programability of large HPC systems also limits the ability to easily analyze communication performance. Conducting profiling with `ibprof` and the Peruse extension showed the potential of low-level profiling and analysis. By overlapping profiling with communication, it is possible to collect detailed communication information with negligible impact on communication performance. Such analysis can easily expose communication anomalies on real systems without significant performance overhead and without exclusive access to the network. This can enable further studies into the development of communication libraries, routing algorithms, and network design.

The use of `ibprof` is currently limited to profiling due to the prohibitive overhead of tracing. The lack of infrastructure to support low-level performance analysis prevents certain studies from being conducted on real systems. The study presented in Chapter 5 is one such study that requires tracing and could not have been performed on a real system because the runtime perturbations caused by fine-grained measurements would affect the final results.

Application-specific Traffic Visualization on Fat-tree networks

Visual representation of performance remains useful when the data is presented in a form that the analyst can easily consume. By encoding network-level, application specific metrics on the hierarchical topology of the network, analysts can gain insights from detailed, low-level performance data because the data is presented in a form that matches the user's mental model of the network. Additionally, showing the metrics from a single application also helps to focus the analysis by removing extraneous and distracting data.

Job's Interfering Potential and Sensitivity to I/O-MPI Interference

The impact of the I/O-MPI interference on performance is dependent on the message/request size, job size, and frequency at which messages/requests are sent for both the I/O and MPI jobs. There is a trade-off between how sensitive a job is to interference and how much interference that job can cause to other jobs. Jobs that send high intensity traffic have low interference sensitivity and high interfering potential. However, the sensitivity and potential are not linearly proportional to each other because each job has a different communication pattern and traffic distribution pattern. Sending traffic from n-I/O clients to m-I/O servers, where $n > m$, creates a different traffic pattern than the 1:1 MPI communicating pairs.

I/O-Congestion Threshold

The identification of the I/O-congestion threshold for a particular traffic pattern can expose important performance features for an application that uses the pattern. If an application's request interval is above the I/O-congestion threshold, it indicates that traffic can be sent more frequently to gain substantial improvement in the application's I/O performance. However, if the requests is already being sent at a intervals below the congestion threshold, increasing the request frequency may not yield meaningful improvements in the job's performance but will severely increase the latency of the individual I/O requests.

Optimizations for Cross-workload Interference on Fat-tree Networks

The characterization of the I/O-MPI interference suggests that users, developers, and administrators should consider the degree of interference caused by their jobs as well as how sensitive their jobs are to interference. This knowledge is helpful in guiding optimization efforts to manage the I/O-MPI interference since different degrees of interference can benefit from different mitigation strategies.

Scheduling and node allocation policies, job queue partitioning strategies, and I/O server placement can be tailored to keep I/O and MPI traffic on separated network paths. On-demand file systems can also be used to isolate an application's I/O traffic from other traffic on the system.

7.3 Limitations and Future Works

While this dissertation provides insight into performance measurement and performance analysis, there are limitations to the scope and applicability of these results. Most importantly, this work supports only fat-tree networks that use InfiniBand-like architectures. These findings cannot be applied "as-is" to other topologies like 3D-torus nor dragonfly topologies.

This work demonstrates profiling on real systems instead of tracing, which is typically more intrusive. Further studies could be done to determine trade-off between the impact of tracing and insights provided by application traces for this level of analysis. The MPI profiling method is demonstrated using Open MPI and, while the methodology can be applied to other MPI libraries, the overhead with other MPI libraries will vary based on how network-level communication is implemented in those libraries. Future work can extend to other network topologies and MPI implementations by using the methodologies presented here.

Where I/O traffic is concerned, only write operations are considered in this work. While tasks such as checkpointing and visualization outputs are predominantly write-intensive, other tasks are more dependent on the performance of other operations. The interference trends for read and metadata operations could be evaluated in future work. The I/O server selection policy can also be reviewed more closely and potentially optimized to minimize cross-workload interference.

The study of the I/O-MPI interference uses synthetic traffic patterns instead of traffic patterns from real application in order to highlight interference trends that are directly related to three variables: message/request size, frequency, and job size. A real application usually has an intricate mix of patterns, which does not easily expose the impact of each variable on the interference caused and experienced by the applications. The interference results in this dissertation are a useful starting point towards investigating more complex traffic patterns in real workloads and improving the prediction accuracy of existing performance models.

Finally, further studies can be conducted on other interference mitigation strategies such as different types of adaptive routing algorithms and 2-phase I/O using MPI-IO (i.e., performing parallel I/O using the MPI library). These have been proposed as ways of improving communication performance but have not be evaluated in the context of cross-workload interference.

Appendix A

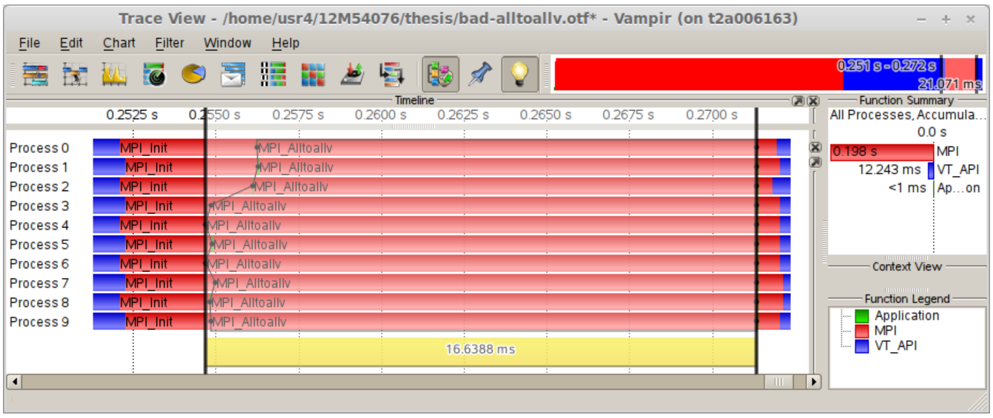
Analyzing MPI Collective Operations using Vampir

Figure A.1 shows the visualization of an `MPI_Alltoallv` operation using the Vampir performance analysis tool, and Figure A.1A shows the code that is used to generate the exchange pattern in this collective operations. The Vampir visualization of this `MPI_Alltoallv` over 10 processes is provided in A.1B. The timeline clearly states the overall duration of the collective but provides no information about the internals of the operation. It gives no insight into whether or not the operation can be optimized and, if so, how this can be achieved. Furthermore, it does not indicate whether the node mapping, the routing/switching protocol, or possible load imbalance, etc., has impacted the performance. This limitation is a result of analysis being done above the MPI layer, not within.

To perform a comparative analysis of the collective's runtime performance, the `MPI_Alltoallv` call is replaced with the sequence of point-to-point calls shown in A.1C. The visualization in A.1D shows the same application using the point-to-point routines instead of the collective. In contrast to this version of the application, the `MPI_Alltoallv` version takes almost twice as long to distribute the data. Therefore, the exchanging of this data using the `MPI_Alltoallv` can be optimized. This loss of performance was not, in any way, identifiable when using the Vampir visualization in A.1B.

```
for (i=0; i<size; i++) {
  sendcounts[i] = i;
  recvcounts[i] = rank;
  rdispls[i] = i * rank;
  sdispls[i] = (i * (i+1))/2;
}
MPI_Alltoallv( sbuf, sendcounts, sdispls, MPI_INT,
  rbuf, recvcounts, rdispls, MPI_INT, MPI_COMM_WORLD );
```

(A) The code that determines the data exchange pattern used by `MPI_Alltoallv`

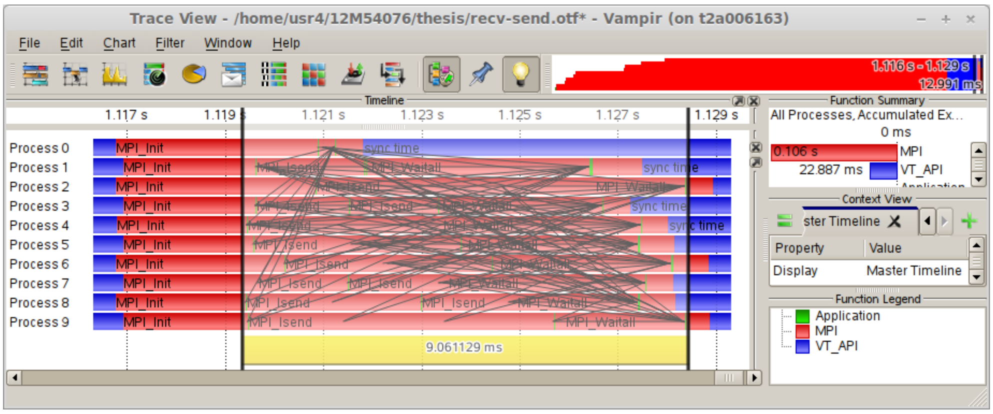


(B) The visualization of data exchange being done by `MPI_Alltoallv`

```
for (i=0; i<size; i++) {
        MPI_Isend(&sbuf[sdispls[i]], sendcounts[i], MPI_INT, i,
                rank*size+i, MPI_COMM_WORLD, &request[i]);
}
for (i=0; i<size; i++) {
        MPI_Irecv(&rbuf[rdispls[i]], recvcounts[i], MPI_INT, i,
                i*size+rank, MPI_COMM_WORLD, &request[i+size]);
}

MPI_Waitall(20, request, status);
```

(C) The replacement code using point-to-point communication, i.e., `MPI_Isend` and `MPI_Irecv` calls



Property Value
Display Master Timeline

(D) The visualization of data exchange being done by `MPI_Isend` and `MPI_Irecv`

Figure A.1: **Comparative Vampir visualizations** | A graphical comparison of the performance of a collective vs. point-to-point operations to perform the same data exchange pattern.

Appendix B

ibprof: The Technical Details

The `ibprof` profiler is a low-overhead InfiniBand traffic profiler for MPI applications. Chapter 3 discusses the design and implementation choices of the `ibprof`, while this appendix provides supplementary technical details.

B.1 Usage Examples

Preloading the library at runtime

```
// preloading at runtime
#> MPI_PARAM= ‘‘-x LD_PRELOAD=$HOME/libibprof.so.0.2’’
#> mpirun -n <num> $MPI_PARAM ./user_program
```

Linking the library to the application

```
// linking at compile link time
#> mpicc user_program.c -libprof
#> mpirun -n <num> ./user_program
```

Profiling all communications within an application

```
MPIRUN_ARGS=‘‘-hostfile hostlist.txt’’    // hostlist.txt stores the list of hosts to be used by MPI
MPIRUN_ARGS=‘‘${MPIRUN_ARGS} -x LD_LIBRARY_PATH -x PATH’’
MPIRUN_ARGS=‘‘${MPIRUN_ARGS} -x LD_PRELOAD=./libibprof.so.0.2’’ // library is preloaded

mpirun -n <num> ${MPIRUN_ARGS} ./user_app
```

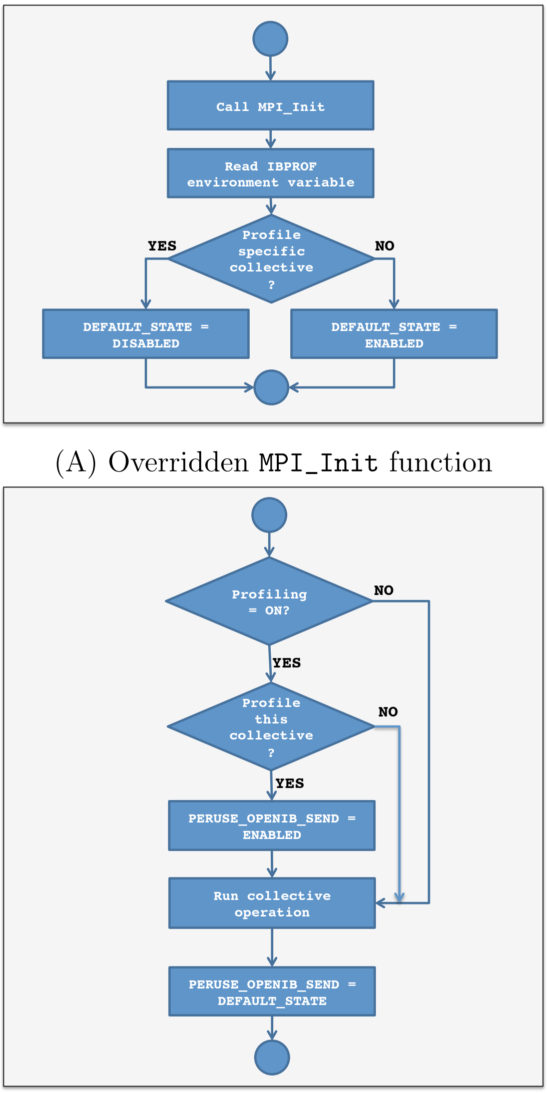
Profiling specific collectives within an application

```
MPIRUN_ARGS=‘‘-hostfile hostlist.txt’’    // hostlist.txt stores the list of hosts to be used by MPI
MPIRUN_ARGS=‘‘${MPIRUN_ARGS} -x LD_LIBRARY_PATH -x PATH’’
MPIRUN_ARGS=‘‘${MPIRUN_ARGS} -x LD_PRELOAD=./libibprof.so.0.2’’ // library is preloaded
MPIRUN_ARGS=‘‘${MPIRUN_ARGS} -x IBPROF=ALLTOALL,BCAST’’  // Only MPI_Alltoall() and MPI_Bcast will be
     profiled

mpirun -n <num> ${MPIRUN_ARGS} ./user_app
```

B.2 Performing Selective Collective Profiling

To selectively profile collectives using environment variables, we use the PMPI interface to override calls to `MPI_Init` and all collective functions. Figure B.1 shows the procedure that is used.


Call MPI_Init
Read IBPROF environment variable
Profile specific collective ?
YES
NO
DEFAULT_STATE = DISABLED
DEFAULT_STATE = ENABLED
Profiling = ON?
NO
YES
Profile this collective ?
NO
YES
PERUSE_OPENIB_SEND = ENABLED
Run collective operation
PERUSE_OPENIB_SEND = DEFAULT_STATE

(A) Overridden `MPI_Init` function

(B) Overridden collective function

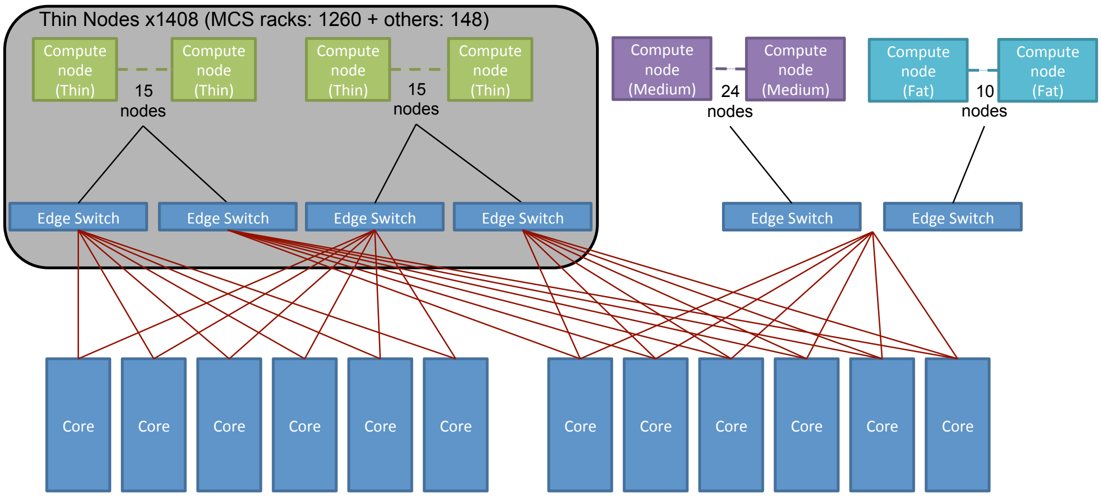
Figure B.1: **Selective profiling of collectives** | The flowcharts show the procedure used to selectively profile specific collective functions. We show only the elements that are relevant to the topic being discussed.

Appendix C

Hardware and Software Specifications of Operating Environments

Software	Version
OS	SUSE Linux Enterprise Server 11 SP1
Compiler	gcc (SUSE Linux) 4.3.4
MPI	Open MPI 1.6.5
OTF	1.12.4 "salmon"

Table C.1: **TSUBAME2.5 Software Specification**



Inter-Node Connection Network

Edge Switches: Voltaire GridDIrector 4036 x179, 4036e x6

Core Switches: Voltaire GridDIrector 4700 x12

Figure C.1: Partial Network Diagram of TSUBAME2.5
Partial Network Diagram of TSUBAME2.5 | information for this diagram was extracted from the Hardware and Software Specifications manual for TSUBAME2.5 GSIC, Tokyo Institute of Technology [2013] published by the Global Scientific Information and Computing Center, Tokyo Institute of Technology

Appendix D

Visualization of I/O Traffic on 1296-node Network

This appendix shows the I/O traffic distribution across the links of the 1296-node fat-tree network used in Chapters 5 and 6. Each end of a link is colored independently based on the amount of traffic sent by the node/switch connected to the respective link-end. Traffic from nodes/servers are not shown in these illustrations.

I/O traffic for **random-node** job placement configuration is used along with two different I/O server placements: the **isolated-target** I/O server placement (see Figure D.1) and the **spread-target** I/O server placement (see Figure D.2). The spread-target placement is used in Section 6.5.2, while the random-node and isolate-target configurations are the defaults most other experiments.

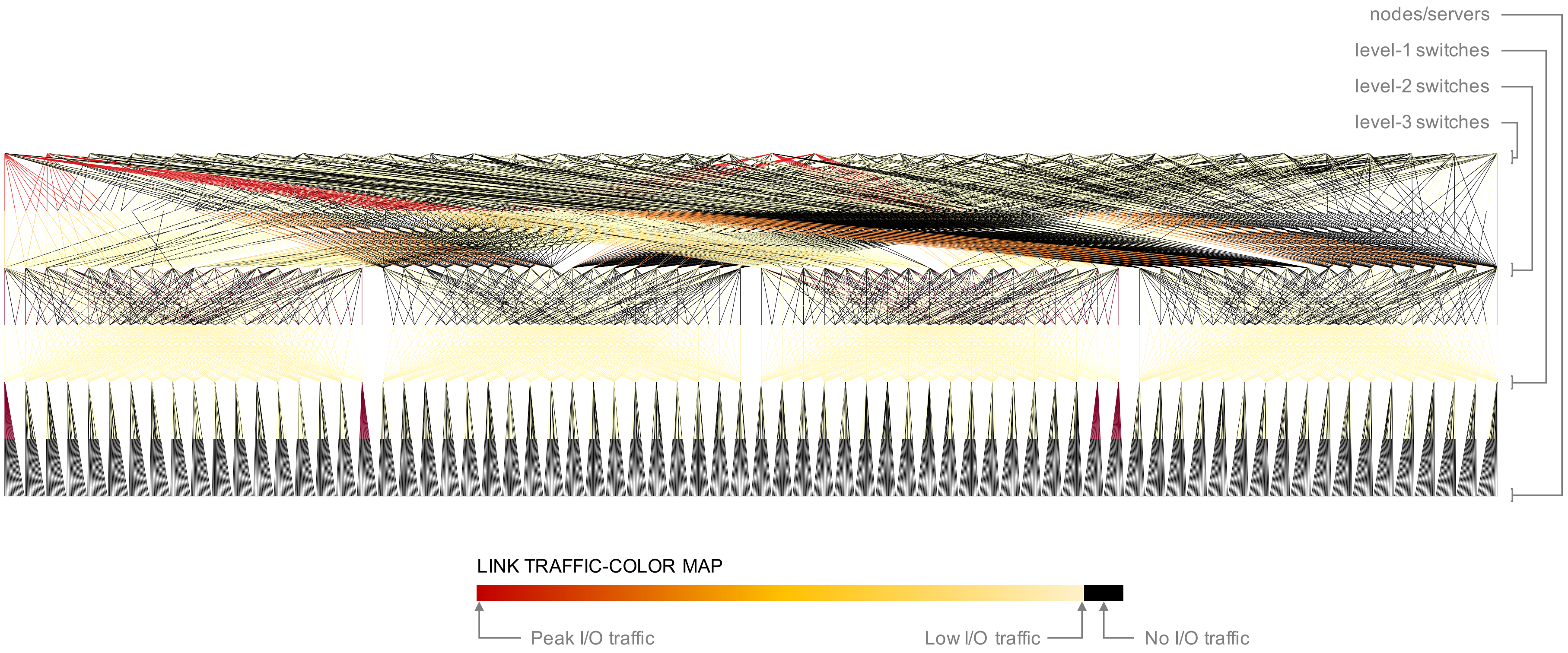


Figure D.1: Visualization of I/O traffic using random-node job placement with isolated-target I/O servers placement on 1296-node network.

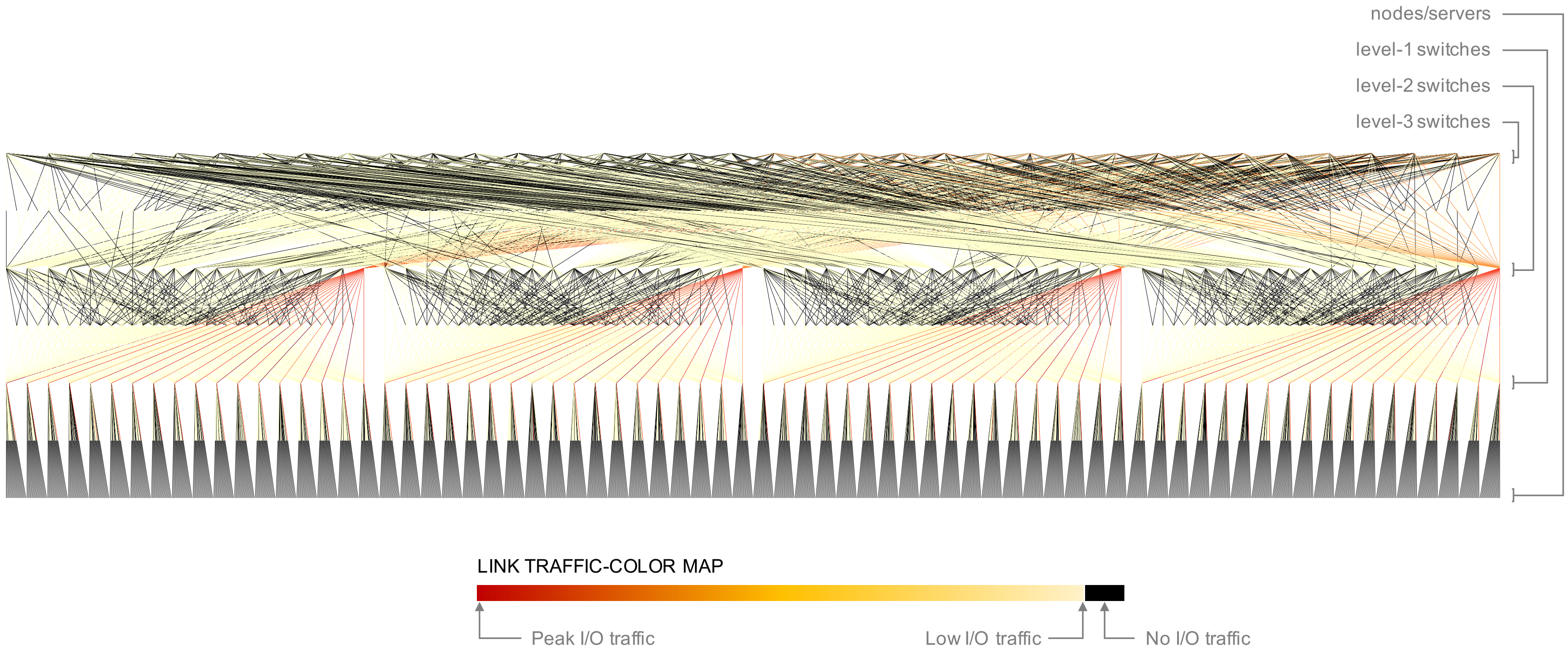


Figure D.2: Visualization of I/O traffic using random-node job placement with spread-target I/O servers on 1296-node network.

Bibliography

(2006). 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. Accessed: 2014-06-29. [Cited on pages 13 and 21.]

(2018). High-performance storage list. Virtual Institute for I/O. [Cited on page 3.]

(2018). The lustre filesystem. http://lustre.org/. Accessed: April 01, 2018. [Cited on pages 3, 48, and 67.]

Adiga, N. R., Blumrich, M. A., Chen, D., Coteus, P., Gara, A., Giampapa, M. E., Heidelberger, P., Singh, S., Steinmacher-Burow, B. D., Takken, T., Tsao, M., and Vranas, P. (2005). Blue gene/l torus interconnection network. *IBM Journal of Research and Development*, 49(2.3):265–276. [Cited on page 1.]

Agelastos, A., Allan, B., Brandt, J., Cassella, P., Enos, J., Fullop, J., Gentile, A., Monk, S., Naksinehaboon, N., Ogden, J., Rajan, M., Showerman, M., Stevenson, J., Taerat, N., and Tucker, T. (2014). The lightweight distributed metric service: A scalable infrastructure for continuous monitoring of large scale computing systems and applications. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 154–165. [Cited on page 5.]

Ajima, Y., Shimizu, T., and Sumimoto, S. (2009). Tofu: A 6d mesh/torus interconnect for exascale computers. *Computer*, 42:36–40. [Cited on page 1.]

Al-Fares, M., Loukissas, A., and Vahdat, A. (2008). A scalable, commodity data center network architecture. *SIGCOMM Comput. Commun. Rev.*, 38(4):63–74. [Cited on page 32.]

Argonne National Laboratory (2). MPICH: High-Performance Portable MPI. http://www.mpich.org/. Accessed: 2014-06-29. [Cited on page 3.]

B. Schmuck, F. and Haskin, R. (2002). GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST '02)*. [Cited on page 67.]

Barcelona Supercomputing Center (2014). Paraver: a flexible performance analysis tool. http://www.bsc.es/computer-sciences/performance-tools/paraver/general-overview. Accessed: 2014-06-29. [Cited on page 38.]

Bautista-Gomez, L., Tsuboi, S., Komatitsch, D., Cappello, F., Maruyama, N., and Matsuoka, S. (2011). Fti: High performance fault tolerance interface for hybrid systems. In *2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–12. [Cited on page 49.]

Bell, C., Bonachea, D., Nishtala, R., and Yelick, K. (2006). Optimizing Bandwidth Limited Problems Using One-Sided Communication and Overlap. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*. [Cited on page 3.]

Bhatele, A., Gamblin, T., Isaacs, K., Gunney, B., Schulz, M., Bremer, P., and Hamann, B. (2012a). 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*, pages 1–11. [Cited on pages 12, 14, and 19.]

Bhatele, A., Gamblin, T., Langer, S. H., Bremer, P.-T., Draeger, E. W., Hamann, B., Isaacs, K. E., Landge, A. G., Levine, J. A., Pascucci, V., Schulz, M., and Still, C. H. (2012b). Mapping Applications with Collectives over Sub-communicators on Torus Networks. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 97:1–97:11. [Cited on page 17.]

Bhatele, A., Jain, N., Isaacs, K. E., Buch, R., Gamblin, T., Langer, S. H., and Kale, L. V. (2014). Improving application performance via task mapping on IBM Blue Gene/Q. In *Proceedings of IEEE International Conference on High Performance Computing*, HiPC '14. IEEE Computer Society. LLNL-CONF-655465. [Cited on pages 6 and 7.]

Bhatele, A., Jain, N., Livnat, Y., Pascucci, V., and Bremer, P. T. (2016). Analyzing network health and congestion in dragonfly-based supercomputers. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 93–102. [Cited on page 14.]

Bhatele, A., Mohror, K., Langer, S. H., and Isaacs, K. E. (2013). There goes the neighborhood: Performance degradation due to nearby jobs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '13*. ACM Press. [Cited on pages 7 and 16.]

Bhatia, H., Jain, N., Bhatele, A., Livnat, Y., Jens Domke, V. P., and Bremer, P.-T. (2018). Interactive investigation of traffic congestion on fat-tree networks using treescope. *20th EG/VGTC Conference on Visualization (EuroVis '18)*, 37(3). [Cited on pages 12 and 14.]

Bhatia, N., Song, F., Wolf, F., Dongarra, J., Mohr, B., and Moore, S. (2005). Automatic Experimental Analysis of Communication Patterns in Virtual Topologies. In *Proceedings of International Conference on Parallel Processing, 2005. ICPP 2005*, pages 465–472. [Cited on page 13.]

Boito, F. Z., Inacio, E. C., Bez, J. L., Navaux, P. O. A., Dantas, M. A. R., and Denneulin, Y. (2018). A checkpoint of research on parallel i/o for high-performance computing. *ACM Comput. Surv.*, 51(2):23:1–23:35. [Cited on page 6.]

Brehmer, M. and Munzner, T. (2013). A multi-level typology of abstract visualization tasks. *IEEE Transactions on Visualization and Computer Graphics*, 19(12):2376–2385. [Cited on page 33.]

Brown, K., Xu, T., Iwabuchi, K., Sato, K., Moody, A., Mohror, K., Jain, N., Bhatele, A., Schulz, M., Pearce, R., Gokhale, M., and Matsuoka, S. (2017). Accelerating big data infrastructure and applications (ongoing collaboration). In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 343–347. [Cited on page 2.]

Brown, K. A., Domke, J., and Matsuoka, S. (2015). Hardware-centric analysis of network performance for mpi applications. In *2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS)*, pages 692–699. [Cited on page iii.]

Brown, K. A., Jain, N., Matsuoka, S., Schulz, M., and Bhatele, A. (2018). Interference between i/o and mpi traffic on fat-tree networks. In *Proceedings of the 47th International Conference on Parallel Processing*, ICPP 2018, pages 7:1–7:10, New York, NY, USA. ACM. [Cited on page iii.]

Buntinas, D. and Gropp, W. (2005a). Designing a common communication subsystem. In Di Martino, B., Kranzlmüller, D., and Dongarra, J., editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 156–166, Berlin, Heidelberg. Springer Berlin Heidelberg. [Cited on pages 19 and 20.]

Buntinas, D. and Gropp, W. (2005b). Understanding the requirements imposed by programming-model middleware on a common communication subsystem. techreport 284, ARGONNE NATIONAL LABORATORY, 9700 South Cass Avenue, Argonne, IL 60439. [Cited on page 19.]

Carns, P., Harms, K., Allcock, W., Bacon, C., Lang, S., Latham, R., and Ross, R. (2011). Understanding and improving computational science storage access through continuous characterization. *Trans. Storage*, 7(3):8:1–8:26. [Cited on page 15.]

Carns, P., Latham, R., Ross, R., Iskra, K., Lang, S., and Riley, K. (2009). 24/7 characterization of petascale i/o workloads. pages 1–10. [Cited on pages 6 and 15.]

Carothers, C. D., Bauer, D., and Pearce, S. (2002). Ross: A high-performance, low-memory, modular time warp system. *Journal of Parallel and Distributed Computing*, 62(11):1648 – 1669. [Cited on page 47.]

Chen, D., Eisley, N. A., Heidelberger, P., Senger, R. M., Sugawara, Y., Kumar, S., Salapura, V., Satterfield, D. L., Steinmacher-Burow, B., and Parker, J. J. (2011). The ibm blue gene/q interconnection network and message unit. In *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–10. [Cited on page 1.]

Cray Research, Inc. (1985). The Cray-2 Computer System. http://s3data.computerhistory.org/brochures/cray.cray2.1985.102646185.pdf. Accessed on: 23 August 2018. [Cited on page 2.]

Dongarra, J. (2013). Visit to the National University for Defense Technology Changsha, China. http://www.netlib.org/utk/people/JackDongarra/PAPERS/tianhe-2-dongarra-report.pdf. Accessed: 2014-07-05. [Cited on page 45.]

Faraj, A., Kumar, S., Smith, B., Mamidala, A., Gunnels, J., and Heidelberger, P. (2009). MPI Collective Communications on the Blue Gene/P Supercomputer: Algorithms and Optimizations. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pages 489–490. [Cited on pages 3 and 17.]

Ferreira, K. B., Widener, P., Levy, S., Arnold, D., and Hoefler, T. (2014). Understanding the effects of communication and coordination on checkpointing at scale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 883–894, Piscataway, NJ, USA. IEEE Press. [Cited on page 49.]

Flash Center for Computational Science, U. o. C. (2017). Flash user's guide. http://flash.uchicago.edu/site/flashcode/user_support/flash4_ug_4p5/. Accessed: June 27, 2018. [Cited on page 59.]

Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R. H., Daniel, D. J., Graham, R. L., and Woodall, T. S. (2004). Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary. [Cited on page 20.]

Gao, Q., Zhang, X., Rau, P.-L. P., Maciejewski, A. A., and Siegel, H. J. (2011). Performance visualization for large-scale computing systems: A literature review. In Jacko, J. A., editor, *Human-Computer Interaction. Design and Development Approaches*, pages 450–460, Berlin, Heidelberg. Springer Berlin Heidelberg. [Cited on page 31.]

Geimer, M., Wolf, F., Wylie, B. J. N., Abraham, E., Becker, D., and Mohr, B. (2010). The Scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience*, 22:702–719. [Cited on pages 4 and 13.]

Ghoniem, M., Fekete, J. D., and Castagliola, P. (2004). A comparison of the readability of graphs using node-link and matrix-based representations. In *IEEE Symposium on Information Visualization*, pages 17–24. [Cited on page 34.]

Ghoniem, M., Fekete, J.-D., and Castagliola, P. (2005). On the readability of graphs using node-link and matrix-based representations: A controlled experiment and statistical analysis. *Information Visualization*, 4(2):114–135. [Cited on page 34.]

Gropp, W. and Lusk, E. L. (1999). Reproducible Measurements of MPI Performance Characteristics. In *Proceeding of Euro PVM/MPI*. [Cited on page 26.]

GSIC, Tokyo Institute of Technology (2013). TSUBAME2.5 Hardware and Software Specifications. http://www.gsic.titech.ac.jp/en/node/420. Accessed: 2014-06-30. [Cited on page 86.]

Huang, W., Santhanaraman, G., Jin, H., Gao, Q., and Panda, D. (2006). Design of High Performance MVAPICH2: MPI2 over InfiniBand. In *Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on*, volume 1, pages 43–48. [Cited on page 3.]

Huck, K. A. and Malony, A. D. (2005). Perfexplorer: A performance data mining framework for large-scale parallel computing. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, SC '05. [Cited on page 13.]

IBM Blue Gene Team (2008). Overview of the IBM Blue Gene/P project. *IBM Journal of Research and Development*, 52(1.2):199–220. [Cited on page 14.]

InfiniBand Trade Association (2014). http://www.infinibandta.org/. [Cited on page 20.]

Intel Corportation. Intel MPI Benchmarks 4.0 Update 2. https://software.intel.com/en-us/articles/intel-mpi-benchmarks. Accessed: 2015-01-26. [Cited on pages 26 and 41.]

Intel Corportation. Intel Trace Analyzer and Collector 8.1. https://software.intel.com/en-us/intel-trace-analyzer. Accessed: 2014-06-29. [Cited on page 13.]

Isaacs, K. E., Giménez, A., Jusufi, I., Gamblin, T., Bhatele, A., Schulz, M., Hamann, B., and Bremer, P.-T. (2014). State of the Art of Performance Visualization. In Borgo, R., Maciejewski, R., and Viola, I., editors, *EuroVis - STARs*. The Eurographics Association. [Cited on page 31.]

Isaacs, K. E., Landge, A. G., Gamblin, T., Bremer, P.-T., Pascucci, V., and Hamann, B. (2012). Exploring Performance Data with Boxfish. In *Proceedings of SC Companion: High Performance Computing, Networking, Storage and Analysis (SCC), 2012*. [Cited on pages 4, 14, 19, 32, and 35.]

Jain, N., Bhatele, A., Howell, L. H., Böhme, D., Karlin, I., León, E. A., Mubarak, M., Wolfe, N., Gamblin, T., and Leininger, M. L. (2017). Predicting the performance impact of different fat-tree configurations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, pages 50:1–50:13, New York, NY, USA. ACM. [Cited on pages 14, 20, 47, 48, 49, and 59.]

Jain, N., Bhatele, A., White, S., Gamblin, T., and Kale, L. V. (2016). Evaluating hpc networks via simulation of parallel workloads. In *SC16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, pages 14:1–14:12, Piscataway, NJ, USA. IEEE Press. [Cited on page 4.]

Keller, R., Bosilca, G., Fagg, G., Resch, M., and Dongarra, J. J. (2006). Implementation and Usage of the PERUSE-Interface in Open MPI. In *Proceedings of Euro PVM/MPI*. [Cited on page 22.]

Kim, J., Dally, W., Scott, S., and Abts, D. (2009). Cost-efficient dragonfly topology for large-scale systems. *IEEE Micro*, 29(1):33–40. [Cited on page 7.]

Knüpfer, A., Brendel, R., Brunst, H., Mix, H., and Nagel, W. E. (2006). Introducing the Open Trace Format (OTF). In *Proceedings of the 6th International Conference on Computational Science - Volume Part II*, ICCS'06, pages 526–533. [Cited on page 24.]

Knüpfer, A., Brunst, H., Doleschal, J., Jurenz, M., Lieber, M., Mickler, H., Müller, M., and Nagel, W. (2008). The Vampir Performance Analysis Tool-Set. In Resch, M., Keller, R., Himmler, V., Krammer, B., and Schulz, A., editors, *Tools for High Performance Computing*, pages 139–155. Springer Berlin Heidelberg. [Cited on pages 4 and 13.]

Kunkel, J., Tsujita, Y., Mordvinova, O., and Ludwig, T. (2009). Tracing Internal Communication in MPI and MPI-I/O. In *Proceedings of International Conference on Parallel and Distributed Computing, Applications and Technologies, 2009.* [Cited on page 13.]

Kunkel, J. M., Betke, E., Bryson, M., Carns, P., Francis, R., Frings, W., Laifer, R., and Mendez, S. (2018). Tools for analyzing parallel i/o. *ArXiv e-prints.* [Cited on page 5.]

Kurth, T., Zhang, J., Satish, N., Racah, E., Mitliagkas, I., Patwary, M. M. A., Malas, T., Sundaram, N., Bhimji, W., Smorkalov, M., Deslippe, J., Shiryaev, M., Sridharan, S., Prabhat, and Dubey, P. (2017). Deep learning at 15pf: Supervised and semi-supervised classification for scientific data. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, pages 7:1–7:11, New York, NY, USA. ACM. [Cited on pages 3 and 49.]

Laboratory, L. L. N. (2018). Cab: Intel Xeon system in Livermore Computing. `http://computation.llnl.gov/computers/cab`. Accessed: 2018-07-03. [Cited on page 48.]

Lammel, S., Zahn, F., and Fröning, H. (2016). Sonar: Automated communication characterization for hpc applications. In Taufer, M., Mohr, B., and Kunkel, J. M., editors, *High Performance Computing*, pages 98–114, Cham. Springer International Publishing. [Cited on page 5.]

Landge, A., Levine, J., Bhatele, A., Isaacs, K., Gamblin, T., Schulz, M., Langer, S., Bremer, P.-T., and Pascucci, V. (2012). Visualizing Network Traffic to Understand the Performance of Massively Parallel Simulations. *IEEE Transaction on Visualization and Computer Graphics*, pages 2467–2476. [Cited on pages 12 and 19.]

Lang, S., Carns, P., Latham, R., Ross, R., Harms, K., and Allcock, W. (2009). I/o performance challenges at leadership scale. In *SC09: Proceedings of the International Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 40:1–40:12, New York, NY, USA. ACM. [Cited on page 6.]

Latham, R., Bautista-Gomez, L., and Balaji, P. (2017). Portable topology-aware mpi-i/o. In *ICPADS IEEE International Conference on Parallel and Distributed Systems.* [Cited on page 17.]

Latham, R., Daley, C., keng Liao, W., Gao, K., Ross, R., Dubey, A., and Choudhary, A. (2012). A case study for scientific i/o: improving the flash astrophysics code. *Computational Science & Discovery*, 5(1):015001. [Cited on pages 3 and 49.]

Leiserson, C. (1985). Fat-trees: Universal Networks for Hardware-efficient Supercomputing. *IEEE Transactions on Computers*, C-34:892–901. [Cited on pages 1, 20, and 32.]

León, E. A., Karlin, I., Bhatele, A., Langer, S. H., Chambreau, C., Howell, L. H., D'Hooge, T., and Leininger, M. L. (2016). Characterizing parallel scientific applications on commodity clusters: An empirical study of a tapered fat-tree. In *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 909–920. [Cited on pages 14, 48, and 59.]

Liu, Q., Podhorszki, N., Logan, J., and Klasky, S. (2013). Runtime i/o re-routing + throttling on HPC storage. In *Presented as part of the 5th USENIX Workshop on Hot Topics in Storage and File Systems*, San Jose, CA. USENIX. [Cited on pages 3, 16, 17, and 67.]

Lofstead, J., Zheng, F., Liu, Q., Klasky, S., Oldfield, R., Kordenbrock, T., Schwan, K., and Wolf, M. (2010). Managing variability in the io performance of petascale storage systems. In *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. [Cited on page 17.]

Luu, H., Winslett, M., Gropp, W., Ross, R., Carns, P., Harms, K., Prabhat, M., Byna, S., and Yao, Y. (2015). A multiplatform study of i/o behavior on petascale supercomputer. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, pages 33–44, New York, NY, USA. ACM. [Cited on pages 3, 6, 15, and 49.]

Mellanox Technologies, Inc. (2018). *ibutils2 - InfiniBand Diagnostic Utilities README*, rev 2.1.1-0.42 edition. Document Number: MLNX-15-4024. [Cited on page 12.]

Mellanox Technologies Ltd. (2004). Mellanox Delivers 300Server Chipset. http://www.mellanox.com/pdf/press_releases/pr_080204.pdf. Accessed: 2018-08-20. [Cited on page 1.]

Mellanox Technologies Ltd. (2017). Mellanox InfiniBand and Ethernet Solutions Accelerate New Intel® Xeon® Scalable Processor-Based Platforms for High Return on Investment. http://www.mellanox.com/page/press_release_item?id=1938. Accessed: 2018-08-20. [Cited on page 1.]

Miguel-Alonso, J., Navaridas, J., and Ridruejo, F. (2009). Interconnection Network Simulation Using Traces of MPI Applications. *International Journal of Parallel Programming*, 37(2):153–174. [Cited on page 13.]

Mondragon, O. H., Bridges, P. G., Levy, S., Ferreira, K. B., and Widener, P. (2016). Understanding performance interference in next-generation hpc systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, pages 33:1–33:12, Piscataway, NJ, USA. IEEE Press. [Cited on page 48.]

Moody, A., Bronevetsky, G., Mohror, K., and d. Supinski, B. R. (2010). Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. [Cited on page 49.]

MPI Forum (2018). MPI: A Message-Passing Interface Standard. http://www.mpi-forum.org/. [Cited on pages 3, 6, 11, and 20.]

Mubarak, M., Carns, P., Jenkins, J., Li, J. K., Jain, N., Snyder, S., Ross, R., Carothers, C. D., Bhatele, A., and Ma, K.-L. (2017a). Quantifying i/o and communication traffic interference on dragonfly networks equipped with burst buffers. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE. [Cited on pages 7, 16, 46, 47, 65, and 66.]

Mubarak, M., Carothers, C. D., Ross, R. B., and Carns, P. (2017b). Enabling parallel simulation of large-scale hpc network systems. *IEEE Transactions on Parallel and Distributed Systems*, 28(1):87–100. [Cited on page 47.]

Mubarak, M. and Ross, R. B. (2017). Validation study of codes dragonfly network model with theta cray xc system. Technical Report ANL-MCS-TM-369, Argonne National Laboratory. [Cited on pages 47 and 48.]

Muelder, C., Gygi, F., and Ma, K.-L. (2009). Visual Analysis of Inter-Process Communication for Large-Scale Parallel Computing. *Visualization and Computer Graphics, IEEE Transactions on*, 15(6):1129–1136. [Cited on page 13.]

Munzner, T. (2009). A nested model for visualization design and validation. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):921–928. [Cited on page 32.]

NASA Ames Research Center (2012). Nas parallel benchmarks. https://www.nas.nasa.gov/publications/npb.html. Accessed: 2014-04-16. [Cited on page 26.]

Öhring, S., Ibel, M., Das, S., and Kumar, M. (1995). On Generalized Fat Trees. In *Proceedings of the 9th International Parallel Processing Symposium, 1995*, pages 37–44. [Cited on pages 20 and 32.]

Oral, S., Simmons, J., Hill, J., Leverman, D., Wang, F., Ezell, M., Miller, R., Fuller, D., Gunasekaran, R., Kim, Y., Gupta, S., Vazhkudai, D. T. S. S., Rogers, J. H., Dillow, D., Shipman, G. M., and Bland, A. S. (2014). Best practices and lessons learned from deploying and operating large-scale data-centric parallel file systems. In *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. [Cited on pages 3, 49, and 66.]

Peña, A. J., Carvalho, R. G. C., Dinan, J., Balaji, P., Thakur, R., and Gropp, W. (2013). Analysis of Topology-dependent MPI Performance on Gemini Networks. In *Proceedings of the 20th European MPI Users' Group Meeting*, EuroMPI '13, pages 61–66. [Cited on pages 3 and 17.]

Petrini, F. and Vanneschi, M. (1997). k-ary n-trees: High Performance Networks for Passively Parallel Architectures. In *Parallel Processing Symposium, 1997. Proceedings., 11th International*, pages 87–93. [Cited on pages 1 and 20.]

Qian, Y., Li, X., Ihara, S., Zeng, L., Kaiser, J., Süß, T., and Brinkmann, A. (2017). A configurable rule based classful token bucket filter network request scheduler for the lustre file system. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, pages 6:1–6:12, New York, NY, USA. ACM. [Cited on pages 17 and 67.]

Rajachandrasekar, R., Moody, A., Mohror, K., and Panda, D. K. D. (2013). A 1 pb/s file system to checkpoint three million mpi tasks. In *Proceedings of the 22Nd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '13, pages 143–154, New York, NY, USA. ACM. [Cited on page 67.]

Requena, C. G., Villamón, F. G., Gómez, M. E., López, P., and Duato, J. (2007). Deterministic versus adaptive routing in fat-trees. *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–8. [Cited on page 58.]

Ross, R. and Latham, R. (2006). Pvfs: A parallel file system. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA. ACM. [Cited on page 67.]

Solomonik, E., Bhatele, A., and Demmel, J. (2011). Improving communication performance in dense linear algebra via topology aware collectives. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, New York, NY, USA. ACM. [Cited on page 7.]

Subramoni, H., Chakraborty, S., and Panda, D. K. (2017). Designing dynamic and adaptive mpi point-to-point communication protocols for efficient overlap of computation and communication. In Kunkel, J. M., Yokota, R., Balaji, P., and Keyes, D., editors, *High Performance Computing*, pages 334–354, Cham. Springer International Publishing. [Cited on page 3.]

Subramoni, H., Vienne, J., and Panda, D. (2013). A scalable infiniband network topology-aware performance analysis tool for mpi. In *Euro-Par 2012: Parallel Processing Workshops*, volume 7640 of *Lecture Notes in Computer Science*, pages 439–450. Springer Berlin Heidelberg. [Cited on page 14.]

Sundar, H., Malhotra, D., and Biros, G. (2013). HykSort: A New Variant of Hypercube Quicksort on Distributed Memory Architectures. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13. [Cited on page 38.]

Texas Advanced Computing Center (2013). Stampede. https://www.tacc.utexas.edu/systems/stampede. Accessed: 2018-08-20. [Cited on page 45.]

The Open MPI Project (2014). Open MPI: Open Source High Performance Computing. http://www.open-mpi.org/. Accessed: 2014-06-29. [Cited on pages 3 and 21.]

Tokyo Institute of Technology (2017). Tsubame3. http://www.t3.gsic.titech.ac.jp/en/hardware. Accessed: 2018-08-20. [Cited on page 45.]

TOP500.org (2018). Top500 List. http://www.top500.org/. [Cited on page 1.]

TU Dresden, Center for Information Services and High Performance Computing (ZIH) (2014). Vampir 8.3: A Use Case. https://www.vampir.eu/tutorial/a_use_case. Accessed: 2014-06-29. [Cited on page 4.]

Vishwanath, V., Hereld, M., Morozov, V., and Papka, M. E. (2011). Topology-aware data movement and staging for i/o acceleration on blue gene/p supercomputing systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 19:1–19:11, New York, NY, USA. ACM. [Cited on pages 6 and 7.]

Wang, T., Oral, S., Wang, Y., Settlemyer, B., Atchley, S., and Yu, W. (2014). Burstmem: A high-performance burst buffer system for scientific applications. In *2014 IEEE International Conference on Big Data (Big Data)*, pages 71–79. [Cited on pages 67 and 74.]

Wong, F., Martin, R., Arpaci-Dusseau, R., and Culler, D. (1999). Architectural Requirements and Scalability of the NAS Parallel Benchmarks. In *Proc. of SC*. [Cited on page 4.]

Xie, B., Chase, J., Dillow, D., Drokin, O., Klasky, S., Oral, S., and Podhorszki, N. (2012). Characterizing output bottlenecks in a supercomputer. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–11. [Cited on pages 6 and 16.]

Xu, T., Sato, K., and Matsuoka, S. (2018). Huronfs : Hierarchical, user-level and on-demand burst buffer file system. https://2018.isc-program.com/?page_id=10&id=post109&sess=sess113. Accessed: June 27, 2018. [Cited on page 67.]

Yang, X., Jenkins, J., Mubarak, M., Ross, R. B., and Lan, Z. (2016). Watch out for the bully! job interference study on dragonfly network. In *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 750–760. [Cited on pages 7, 16, and 66.]

Yuan, X., Mahapatra, S., Nienaber, W., Pakin, S., and Lang, M. (2013). A new routing scheme for jellyfish and its performance with hpc workloads. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 36:1–36:11, New York, NY, USA. ACM. [Cited on pages xiii and 15.]

Zahavi, E. (2011). 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*, pages 761–770. [Cited on pages 3, 17, 20, and 66.]