ABSTRACT

KHETAWAT, HARSH. Managing Extreme Heterogeneity in Next Generation HPC Systems. (Under the direction of Rainer Mueller).

As traditional high performance computing architectures are unable to meet the energy and performance requirements of increasingly intensive applications, HPC centers are moving towards incorporating heterogeneous node architectures in next-generation HPC systems. While GPUs have become quite popular over the last few years as accelerators, other novel acceleration devices such as FPGAs and neural network processors are also gaining attention. Furthermore, heterogeneity is being incorporated in not just compute capabilities but also in the memory hierarchy with technologies such as HBM, NVRAM and PCM (e.g., Intel Optane); in the storage stack with the introduction of burst buffers, both node-local and distributed; and in the network interconnect with technologies such as GPUDirect and NVLink. This creates the need for a careful study of the compute, storage and network stack of HPC systems to extract the most performance from these increasingly heterogeneous node architectures.

HPC applications are often composed as computational kernels, where each kernel has different computational characteristics, memory access patterns, communication patterns and accesses to storage devices for I/O. Suitability of different architectural features are therefore highly dependent on the application mix at an HPC center. Furthermore, application performance often depends on the application developer's ability to utilize the resources available to them.

To tackle this extreme heterogeneity that is emerging in HPC systems, we first create a simulation framework that allows HPC centers and application developers to study the optimal placement of storage resources in the context of an HPC interconnect topology. We study the storage performance of different applications with node-local and multiple distributed burst buffer placements in popular HPC network topologies. Next, we develop a simulation framework to study the networking performance of applications in systems that employ modern communication technologies like NVLink and GPUDirect. Our framework is intended to be used by application developers and HPC centers to determine the performance gains that can be achieved by leveraging these novel technologies. Finally, we address the heterogeneity in compute resources. We develop a framework for sharing the work of a single kernel amongst multiple accelerators as well as co-scheduling multiple applications on the same HPC node. We use four applications to study work sharing on a node with a CPU, a GPU and an FPGA. We also create workloads from these applications to assess the co-scheduling performance under four scheduling algorithms.

This work shows that a holistic approach is required for the heterogeneity that is emerging in storage, interconnect and compute stacks in modern HPC systems across all components of the system in order to optimally use the variety of resources available in next-generation HPC.

Managing Extreme Heterogeneity in Next Generation HPC Systems

by
Harsh Khetawat

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina
2022

APPROVED BY:

Xipeng Shen	Guoliang Jin
Michela Becchi	Rainer Mueller Chair of Advisory Committee

DEDICATION

To my family for believing in me.

BIOGRAPHY

The author was born in a Kolkata, West Bengal in India where he also went to school at Don Bosco Park Circus. He then moved to Manipal, Karnataka for his undergraduate education where he completed his Bachelor of Engineering degree in Computer Science. After completing his undergraduate studies he worked at Microsoft India Development Center in Hyderabad for 2 years. In order to pursue graduate education he then moved to Raleigh, North Carolina to do his Masters in Science degree at North Carolina State University. Immediately after completing his Masters degree he enrolled in a doctoral degree at the same university under the guidance for Dr. Frank Mueller.

ACKNOWLEDGEMENTS

I would like to thank everyone who has helped me over these trying last few years. My family, especially my parents, have been a constant source of love and affection.

I would like to thank my advisor Dr. Frank Mueller for giving me the opportunity to work under his guidance. His deep insights, encouragement and advice have been invaluable towards the completion of this work. I would also like to thank Dr. Guoliang Jin, Dr. Xipeng Shen and Dr. Michela Becchi for taking interest in my work and being part of my advisory committee. I would also like to thank the people in my research group with whom I've had truly wonderful discussions.

I would like to thank my collaborators at ORNL and LLNL, Chris and Abhinav. This work would not have been possible without their guidance.

I would also like to thank my friends both in the USA and in India, they make the most trying times seem easy.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER

1 INTRODUCTION

Computing has become a cornerstone of scientific discovery ever since the invention of electronic computing. From the discovery of medicines to the processing and visualization of astrophysical data and the simulation of atomic phenomena, computing resources have become intertwined with the ability of researchers to conduct scientific discovery. This has led scientific institutions to create ever more powerful computers with the sole purpose of conducting research giving rise to what we today call high performance computing (HPC). HPC systems traditionally consist of individual nodes that provide the computing resources connected by a high-bandwidth, low latency network and a centralized and high capacity storage system. While improvements in processor technology have served HPC systems well for a long time, traditional HPC has been hitting a bottleneck in terms of performance and energy consumption. Performance of HPC systems has traditionally been measured in floating point operations per second (FLOPS). While Moore's law has enabled HPC centers to achieve increasingly higher FLOPS from one generation to the next, researchers have realized that we are now close to the physical limits of processor technology. Further gains in performance require the integration of various computing, networking and I/O technologies that are best suited for the applications that are expected to run on the system. This has led to fundamental changes in the architecture of HPC systems.

One fundamental advance was the introduction of Graphics Processing Units (GPUs) as co-processors along with the CPU. While GPUs had been used extensively for graphics tasks, its introduction to HPC systems was due its massive parallelism and the ability to process massive amounts of floating point data, which is a great fit for HPC applications. Since then GPUs have become increasingly popular with recent systems such as Sierra and Summit as well as the upcoming Frontier system relying on GPUs for the majority of their computation capabilities. Furthermore, traditional memory subsystems have consisted of primarily Dynamic Random Memory Access (DRAM) along with multiple levels of caching. This was found to be lacking in speed and capacity leading to the incorporation of technologies such as High Bandwidth Memory (HBM) and Non-Volatile RAM (NVRAM) into the memory hierarchy. Similarly the storage system has been augmented with burst buffers to absorb bursty I/O traffic and allowing applications to continue execution rather than waiting on the I/O to be served by the centralized storage system. Furthermore, the network interconnect has also seen enhancements to keep up with the evolution in compute and memory technologies. Paradigms such as GPUDirect and interconnects such as NVLink allow GPUs to utilize the network resources without the active involvement of the host DRAM for communication. This trend has led to what is called "extreme heterogeneity" to describe modern and next generation HPC systems. This trend is likely to continue with the evolution of HPC workloads to include applications for genetics, machine learning and artificial intelligence as well as current applications requiring higher fidelity results. Compute resources, such as FPGAs, DSPs and neural network processors have found their way into large production data centers and are now entering the realm of HPC systems. Figure 1.1 shows examples of heterogeneity in different HPC subsystems.

Along with opportunities, this heterogeneity also creates significant challenges for both HPC centers and application developers. The goal of HPC centers is to ensure maximal usage of computing resources while keeping the capital and operational expenditures in check. The capital expenditure for HPC centers is primarily the cost of system procurement while the operational expenditures includes the energy requirements and cost of maintenance. Both the capital and operational expenditures are functions of the choices made during system procurement. For example, using GPUs and FPGAs as accelerators can provide comparable performance while keeping energy usage in check albeit at the cost of system procurement. Similarly, centers have to choose between procuring fewer fat nodes (performance dense nodes) or a larger number of thinner nodes (fewer computing resources), which impacts the overall cost of the system. Even further, a larger number of thinner nodes

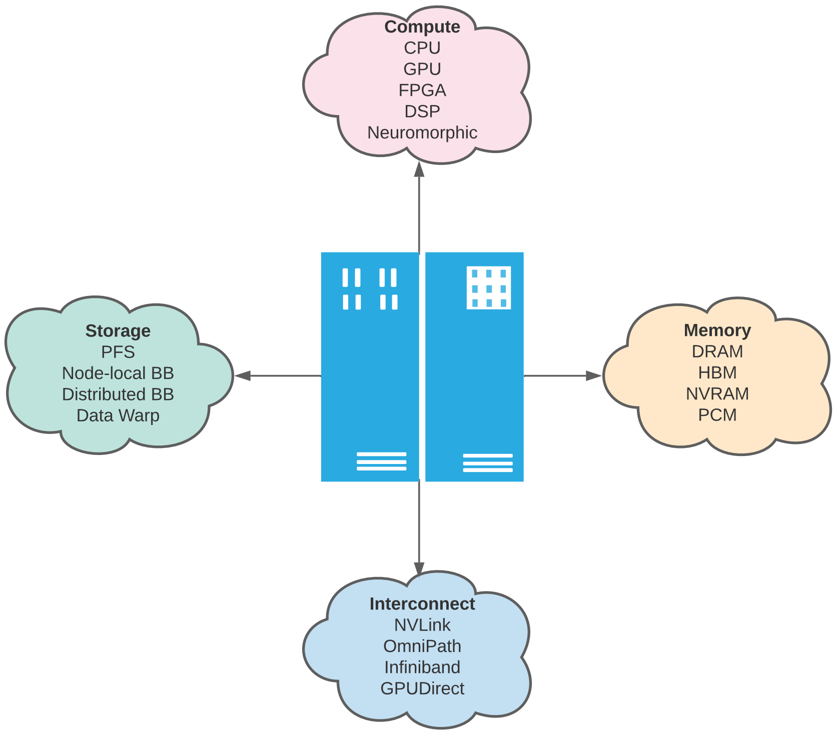


Figure 1.1: Heterogeneity in HPC

ensures better system utilization compared to fewer fat nodes as applications need to be enhanced to use all the compute resources available on a node. The challenges faced by application developers is the effort required to enhance their applications to effectively utilize the resources available on the system. With different application characteristics, not all resources will lead to improved performance. The aim of this work to develop and evaluate tools and frameworks to enable HPC centers and application developers to effectively use the heterogeneous resources available on a modern HPC system.

Hypothesis: In order to effectively utilize heterogeneous resources in HPC, software support is required to identify the best placement of such resources, to guide programmers in application tuning with respect to heterogeneous capabilities, and to support elastic and transparent scheduling of workloads across such resources within and across applications.

We develop tools and frameworks to evaluate our hypothesis. This work is divided into three primary chapters. Chapter 2 evaluates the placement of burst buffers in the context of the inter-node network interconnect in an HPC system. We evaluate the I/O performance of five applications across different placement strategies and network topologies using parallel discrete event simulation. Along with an evaluation of the performance of these placement strategies across network topologies, we also discuss the expenditures involved for the HPC center when making procuring decisions and provide a tool for centers to support these evaluations for their own workloads.

In Chapter 3, we evaluate the messaging performance of applications utilizing heterogeneous compute resources. With the development of technologies such as GPUDirect and NVLink, accelerators such as GPUs can act as first class devices with the ability to directly communicate over the network. Furthermore, HPC centers are making trade-off decisions around accelerator density. We develop and evaluate a parallel discrete event simulator to predict the messaging performance of applications if they were to modify their applications to support these modern communication technologies. We also evaluate the communication performance of these technologies for various node configurations with respect to the number of GPUs per node.

Finally, in Chapter 4, we provide methods to tackle the emerging heterogeneity in the compute resources on HPC systems. We develop a framework to seamlessly share the computation of a single kernel across multiple accelerators. We also develop a scheduler and implement four scheduling algorithms to allow HPC centers to co-schedule multiple applications on the same node with each application getting exclusive access to a subset of accelerators. Our framework allows applications to be migrated between accelerators, to be expanded to more or contracted to fewer devices. We evaluate our framework with 4 applications as well as workloads comprising of these applications to demonstrate that both work sharing and co-scheduling can result in significant performance gains.

CHAPTER

2 EVALUATING BURST BUFFER PLACEMENT IN HPC SYSTEMS

2.1 Introduction

With the increasing scale of computation and data in High Performance Computing (HPC), existing storage systems are becoming a bottleneck for large-scale scientific and data-intensive applications [XCD+12]. The ratio of I/O bandwidth to bytes of memory capacity on Titan (the 27 petaflops (PF) system at Oak Ridge Leadership Computing Facility, OLCF, currently at No. 12 in the Top500 list with a 1 TB/s filesystem) is 0.0016 and the ratio on Summit (the 200 PF system at OLCF, currently No. 1 in the Top500 list with a 2.5 TB/s filesystem) is 0.0001. Data generation rates are increasing faster than traditional parallel file system (PFS) ingestion capabilities. To alleviate this bottleneck, the traditional PFS is being augmented with a tier of intermediate, high-bandwidth flash-based storage devices called burst buffers (BBs). These BBs sit between compute nodes and the parallel file system (PFS), and are designed to absorb the periodic I/O bursts of HPC applications.

BBs allow applications to checkpoint their state more quickly and frequently to persistent storage and data to be staged for input and output, enabling the application to resume computation rather than wait for I/O. The functional advantages and disadvantages of these architectures have been studied in previous work [HOAV16]. In large-scale HPC centers, BBs are a multi-million dollar resource that impact the center's productivity, the I/O performance of the application workload and the scientific progress of the users. The efficacy of BBs and their I/O performance depend on optimal provisioning and architectural details.

Extant BB provisioning solutions are based on either simple rules of thumb or simplistic scenarios (single application I/O behavior in isolation), and are not representative of the complexity involved in the design space. In this thesis, we argue that an optimal provisioning of BBs for a large-scale HPC center should carefully consider and reconcile a variety of factors. Figure 2.1 illustrates the dimensions system designers and practitioners should reconcile during BB provisioning. For example, careful consideration must be given to aspects such as the locality of BBs, the underlying network topology, the application workload's I/O characteristics, and the job scheduling mix of the respective facility. Failure to do so will result in sub-optimal I/O solutions and slowdown of scientific workflows.

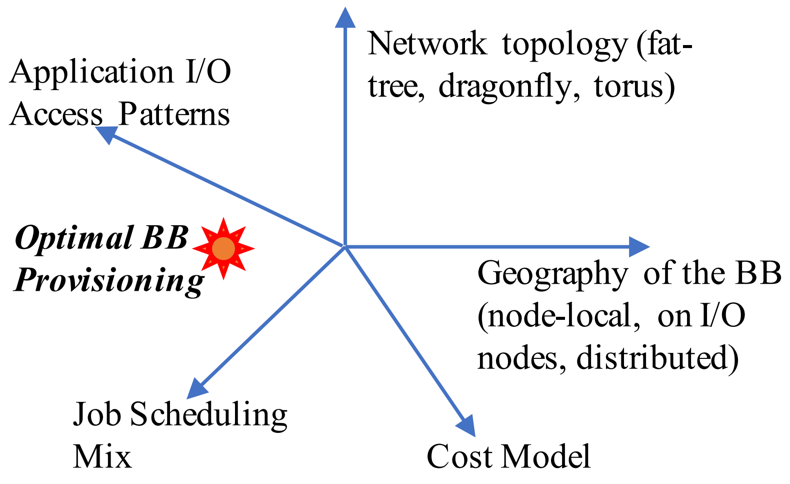

Network topology (fat-tree, dragonfly, torus)
Application I/O Access Patterns
Optimal BB Provisioning
Geography of the BB (node-local, on I/O nodes, distributed)
Job Scheduling Mix
Cost Model

Figure 2.1: BB Provisioning Dimensions

To understand the design space further, let us consider the geographic locality of BBs within the HPC system. Various hierarchies exist for the placement of BBs in HPC systems: (a) BBs co-located with compute nodes, e.g., used in Summit, the next-generation OLCF system [Lab21c]; (b) BB nodes located alongside I/O nodes, e.g., used in Cori, an HPC system at NERSC [Cen21]; (c) global BBs. In terms of network topology, modern HPC systems use a range of network topologies, e.g., fat-tree [Lei85], dragonfly [KDSA08], torus, etc. These network topologies have different blocking characteristics, diameters, bisection bandwidths, and costs. The performance of BBs is most closely affected by the network topology, and then the application workload's I/O access patterns (N-N, N-1, etc.), the scheduling mix of the jobs and the interplay of their traffic. All of the above factors play a vital role in the eventual BB experience that user applications perceive.

In this work, we present a more rigorous approach to BB provisioning that carefully reconciles the aforementioned vectors. We have created a provisioning framework using Parallel Discrete Event Simulation (PDES), which provides a tool to HPC system designers allowing them to rapidly model their workloads against different BB architectures, placement strategies, and network configurations. System designers can then use the framework to make provisioning decisions. Using our simulations and models, we can evaluate various architectures to determine the ideal choices based on the application workload, performance requirements, and cost constraints.

The primary contributions of this work are as follows:

• the development of a complete framework that system designers and practitioners can use to input BB locality, network topology, I/O workload patterns, job scheduling mix and cost to study "what-if" scenarios for BB provisioning.

• the development of a variety of network and BB models (e.g., node-local) for a large-scale HPC system, and then simulating them via the CODES [MCRC17][CLL+11] suite (and extending it) to assess their performance for varying workloads;

• the ability to replay workloads composed of *multiple* applications with customizable node allocation policies to accurately model an HPC center;

• the development of a novel capability to *strongly and weakly scale traces* of the Darshan I/O logs with the simulator to project future workloads for larger I/O sizes than any I/O traces collected on today's platform;

• the development of novel features within Darshan to replay I/O traces with semantics for *barrier-based blocking I/O collectives* to improve replay accuracy.

Besides assessing system configurations for procurement (as in this thesis), the framework can further facilitate the development of parallel file systems, data staging schemes, traffic shaping algorithms and resiliency techniques by performing sensitivity studies to parametric variations.

2.2 Background

In this section we will discuss tiering in modern HPC storage systems and HPC network topologies. We will also discuss the functional advantages and disadvantages that different burst buffer placement techniques have and how they impact the network interconnect.

HPC Storage Tiers

Bursty I/O traffic from applications has been reported to create a bottleneck at the level of shared disk-based parallel file systems [XCD+12]. In order to absorb these spikes in I/O traffic, the addition of a fast tier of SSD-based storage, called BBs, has been both proposed and implemented [LCC+12][BBR+16]. BBs are closer to the compute nodes than the PFS and offer the applications significantly higher bandwidth, albeit at a much lower capacity. Applications that read/write a large amount of data can now keep the compute nodes busy by utilizing the high-bandwidth SSD-based storage for checkpointing, reading input data, writing preliminary results, and their final output to persistent storage. BBs also allow applications to perform significantly faster in-situ analysis and jobs in a workflow to have a high-bandwidth, low-latency scratchpad.

Several leadership-scale computing systems have deployed BBs. Theta at the Argonne National Laboratory, Summit at the Oak Ridge National Laboratory, and Sierra at the Lawrence Livermore National Laboratory have node local BBs. On the other hand, Trinity at the Los Alamos National Laboratory, Conrad at the Zuse Institute Berlin, and Cori at the Lawrence Berkeley National Laboratory have distributed BBs.

Network Topologies

The performance of HPC applications is often limited by communication rather than computation making the network interconnect a key determinant of system performance. A host of network topologies exist, including tori, trees and meshes plus recent ones like dragonfly, slimfly [BH14], and HyperX [ABD+09]. Each one of these topologies offer different characteristics such as diameter, bisection bandwidth, direct/indirect network, and cost (see Table 2.1).

The choice of network topology and allocation of jobs to nodes has a significant impact on system performance. Furthermore, placement of BB nodes can impact network contention in these interconnect topologies. Therefore, it becomes important to study the impact of application communication, BB I/O traffic and shared parallel file system traffic

Table 2.1: Network Topologies

Network	Bisection	Diameter	Network Type	Cost
3-level Fat-Tree	Full	4	Indirect	High
3D Torus	Low	High	Direct	Low
Dragonfly	High	3	Direct	Medium
Slimfly	High	2	Direct	Medium

on them.

Application Workloads

Leadership computing facilities run a variety of application workloads, from large jobs that occupy a significant fraction of the system to several small to medium sized jobs each of which occupy a smaller fraction of the system. Workloads can also vary by their duration of execution from a few hours to a few days. These factors have an impact on the frequency and size of I/O traffic that these applications have.

Applications also have different checkpointing requirements, input/output characteristics, and I/O stages that affect the overall performance of the application workload. The use of traces from leadership class machines allows us to replicate application behavior at a high resolution.

2.3 Related Work

Prior work has focused on the exploration and implementation of BB architectures in HPC systems. Kimpe et al. [KMM+12] describe the design of a container abstraction to manage in-system storage devices and to transfer data in the storage hierarchy. Herbein et al. [HAL+16] use I/O aware batch scheduling to reduce contention with novel scheduling techniques. This reduces contention on the parallel file system, resulting in reduced job variability. BurstMem [WOW+14] provides storage and communication strategies for BB systems. It shows that if handled efficiently BBs can significantly speed up application I/O performance. TRIO [WOP+15] is another framework that coordinates flushing from BBs to the parallel file system in order to maximize storage bandwidth by reducing the contention between storage servers. In contrast, we focus on the impact of BB placement on application performance.

Bhimji et al. [BBR+16] explore the use of the Cori BB system at the National Energy

Research Scientific Computing Center (NERSC) of the Lawrence Berkeley National Laboratory. They discuss the performance gains achieved using BBs compared to only a shared Lustre parallel file system. Liu et al. [LCC$^+$12] have used parallel discrete event simulation to evaluate BBs in leadership-class storage systems. They analyze common burst patterns in applications and use simulation to analyze the I/O performance of applications. While Liu et al. delve into application performance in the presence of a BB system, our work focuses on creating a reproducible framework for analyzing application and BB performance for different BB placement strategies and network configurations.

More recently, Mubarak et al. [MCJ$^+$17] have used simulations to show the effects of interference of network and I/O traffic in dragonfly network topologies equipped with BBs. They use different routing strategies with realistic workload sizes to demonstrate that balancing I/O and network traffic requires a careful selection of routing policies, and job and data placement. Harms et al. [HOAV16] describe the use cases of BBs and how different BB architectures are more suited for certain functionality than others. Cao et al. [CSB17] compare the performance of local and shared BB systems. Their results show that shared BB organizations can result in higher I/O throughput than local BBs. Instead of static checkpoint sizes, we use application traces collected from real executions to more accurately simulate application I/O behavior.

2.4 Overview

As part of this work, we create a framework using CODES to rigorously examine placement of storage resources in modern HPC systems. The network models allow HPC centers to simulate current and proposed HPC topologies. The burst buffer models in our framework also facilitate a what-if comparison of various burst buffer architectures against which HPC centers can simulate their workloads. HPC centers can use Darshan traces collected from previous application executions to project application I/O behavior with significant resolution to future systems. Our framework can also be used to augment the Darshan traces with MPI synchronization primitives, to increase the number of I/O phases, and to scale the traces via extrapolation, both strongly and weakly. These capabilities allow for projections to future workload sizes while preserving application behavior. Our framework can aid burst buffer provisioning by allowing HPC centers to study various network and storage architectures with multi-job workloads and node allocation policies specific to the center.

2.5 CODES Simulation Suite

Parallel Discrete Event Simulation

In Discrete Event Simulation (DES) [HOP+86], the system is modeled as a sequence of discrete events. Each event changes the state of the system. Since these events occur at particular instances in time and trigger state changes, the system state is assumed to be constant between state changes. Parallel Discrete Event Simulation (PDES) [Fuj90] exploits the parallelism to significantly speed-up simulation performance, while also allowing us to scale the simulation to larger sizes. On the other hand, PDES is hard to implement because some events might affect others, and therefore require sequencing constraints. Without sequencing constraints, causality errors can occur.

Two mechanisms manage sequencing constraints — conservative, and optimistic. For conservative PDES, causality errors are prevented from occurring by issuing an event only when it is safe. Events are not processed until all events that might affect it have completed. Alternatively, optimistic PDES allows causality errors to occur but has mechanisms to detect such errors and roll back to a correct state.

Rensselaer's Optimistic Simulation System (ROSS)

ROSS [CBP02] is a DES that uses Time Warp [Jef85] for synchronization. Each component in the system is modeled as a logical process (LP), which communicates by exchanging timestamped event messages. It has support for both sequential and parallel (conservative and optimistic) simulations.

The Time Warp mechanism synchronizes computation by detecting events that occurred out of timestamp order, rolling-back these events, and finally re-executing them. In order to improve performance, ROSS uses a technique called Reverse Computation[CPF99][PC03]. In Reverse Computation, instead of saving and recovering state in case of causality errors, roll-back is done by reverse executing code. This allows us to scale the simulation to highly parallel machines and saves memory as states between events need not be preserved.

CODES

CODES builds on ROSS in order to enable highly parallel simulations of exascale network and storage architectures in HPC environments. CODES abstracts the network models as components to create packet-level simulations of the most popular HPC network topologies. It supports dragonfly, slimfly, torus and fat-tree topologies. It includes support for packetization of messages and provides an API to simulate both MPI and RPC style communication. It also has support for storage models including a local storage and a CODES store model, which can be used to simulate BBs.

Once the HPC system has been modeled using the storage and network components, CODES can simulate a range of I/O and network workloads. CODES has support for synthetic workloads, checkpoint workloads as well as replaying network traces such as SST DUMPI and I/O traces from Darshan. For our experiments, we use Darshan traces from real- world HPC applications to evaluate the performance of BBs. Finally CODES allows for the collection of several metrics related to the simulation, which we use for our evaluation.

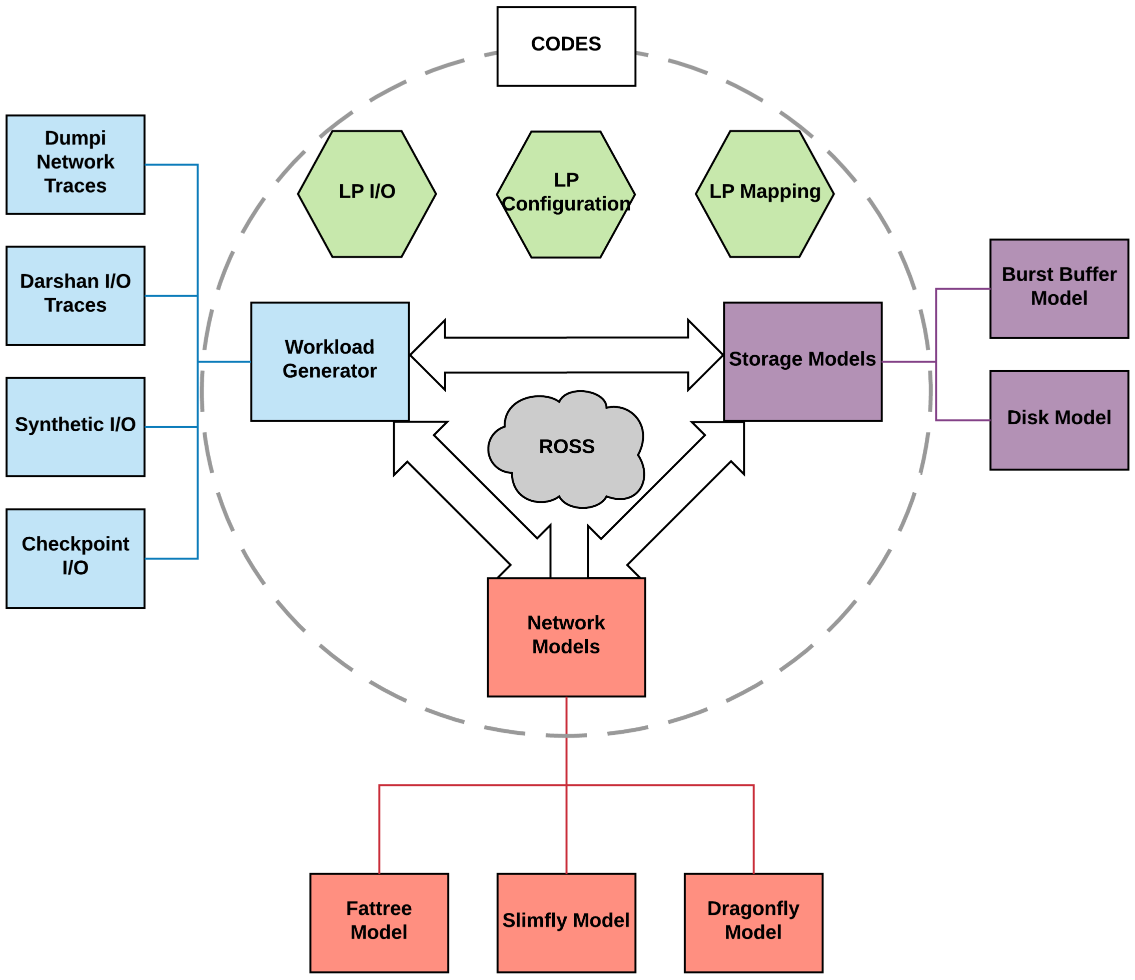


Figure 2.2: The CODES simulation suite

Darshan

Darshan [CLR+09] is an I/O characterization tool developed by researchers at Argonne National Laboratory that allows application developers and HPC system administrators to capture an accurate picture of the I/O behavior of an application. It consists of two parts, a runtime, which is a lightweight library used to instrument the application at execution time, and the Darshan utility, which is a collection of tools used to analyze Darshan traces.

The Darshan runtime is lightweight enough to be used on several current generation HPC systems in order to instrument I/O behavior and has been deployed at the Argonne Leadership Computing Facility (ALCF), the National Energy Research Scientific Computing Center (NERSC), and the Oak Ridge Leadership Computing Facility (OLCF), among others. We use the traces collected by these systems to resemble the I/O behavior of scientific applications in our simulation. We also use the Darshan utility to scale the Darshan traces collected by these HPC centers for future, larger scale HPC systems.

2.6 Workload and Simulation Design

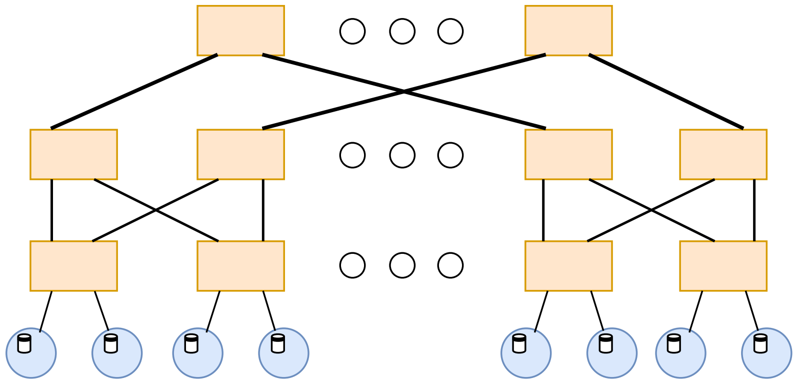
2.6.1 BB Placement

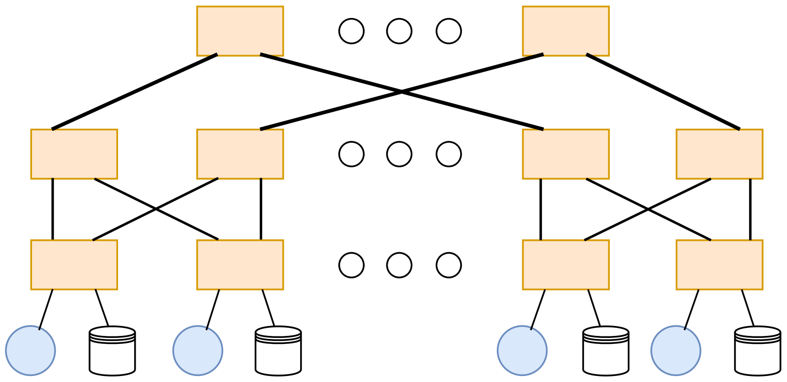
The placement of BBs in an HPC system has a significant impact not only on the I/O performance of the application but also on the performance of the interconnect. In order to evaluate the performance of the placement models, we simulate each model under varied configurations. Their performance is then evaluated with different workloads and network topologies (described later). Each model has its own functional advantages and disadvantages, which we also discuss. Figure 2.3 shows the different BB architectures we are evaluating in a fat-tree network configuration.

Node-Local BBs

A storage device is placed within the compute node allowing applications to have exclusive access to the storage resource while enabling I/O performance to scale linearly with the number of nodes in the system. Further, I/O operations do not result in network traffic as it goes through the local bus. One drawback of using this model is that the BBs are tightly coupled with the compute nodes, which creates a single failure domain. It also makes it difficult to support applications that utilize shared files.

To simulate the node-local BB model, we integrate the CODES storage server into the


(a) Node-Local (BB in leaf nodes)


(b) Grouped (BB in adjacent nodes)

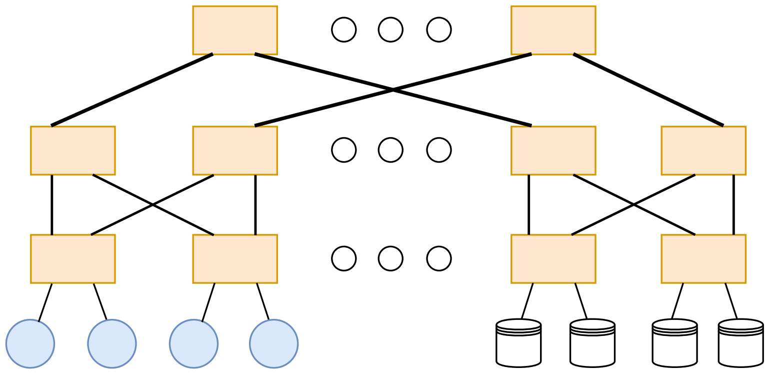

(c) Global (BB is dense node subset)

Figure 2.3: BB architectures on a fat-tree network.

client. Since reading/writing from/to the local buffer does not result in network traffic, one can model the BB in this manner. The I/O is modeled to be synchronous, and application execution resumes after I/O completes. Bandwidth, latency and seek times of storage devices are configurable via the CODES configuration.

Grouped BBs

BB servers are placed in the group along with the compute nodes in order to exploit local network links, which are typically non-blocking. This model allows applications to write to shared files and supports easy stage-in and stage-out. It is also much more straightforward to share data between nodes. Here, the bursty I/O traffic could negatively impact the PFS as buffers share network connections. This configuration incurs an additional server, networking, and cabling costs. We simulate this model by placing the CODES storage server model in each group along with the client LPs (logical processes). The CODES storage server is built using the local storage model that simulates a disk along with networking and has support for threading, which simulates multiplexed transfers. Memory size, storage size, threads, read/write bandwidth, latency, and seek times are configurable in the CODES configuration file. Read/writes from compute nodes result in network traffic to I/O nodes.

Global BBs

BB servers are placed on nodes separate from both the compute nodes and the I/O nodes. This allows the compute nodes to use shared files, and the BBs might still be usable even if the parallel file system goes down. The bursty I/O here results in additional traffic on the global network, which could cause degraded performance. It also increases the cost of the system due to additional servers, network interface cards (NICs), and network switches and cables. The simulation of global BBs is done using the same CODES storage server placement as in the last model, except that the BB servers are placed in a separate group with a configurable number of burst buffer nodes. The I/O in this model causes network traffic to be generated from the compute nodes to the BBs. A significant difference from the previous model is that flushing data from the BBs to the parallel file system would result in network traffic from the BB nodes to the I/O nodes. Parameters such as memory, storage, bandwidth, etc. are configurable in the CODES configuration file.

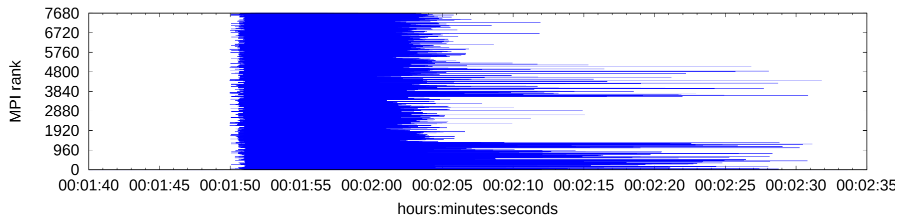
Locality and Striping

For the node-local model, the MPI ranks on a node conduct I/O with their local BBs without accessing the network. In the grouped model, ranks of the application perform I/O with the BB device located in the same group as well as with the BB devices in the adjacent groups based on the striping configuration. Similarly, for the global BB model, the application performs I/O with a respective BB node from the set of global BBs as well as a certain number of adjacent nodes as specified in the striping configuration. In our experiments, the I/O in the node-local model is not striped across multiple BB devices, i.e., the only cause of contention is the SSD device on the BB node. For grouped and global BB models, we stripe the I/O across 4 BB nodes with a stripe width of 128KB. This configuration closely mirrors that of common HPC storage hierarchies including Data Direct Network's (DDN) Infinite Memory Engine (IME).

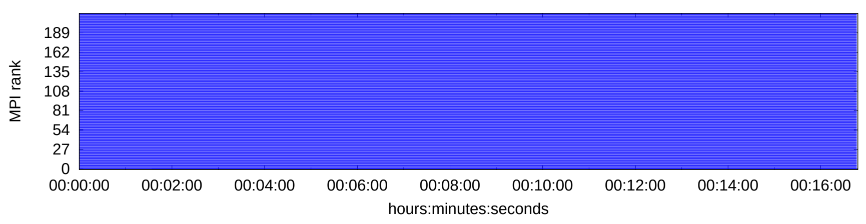
2.6.2 Workloads

Gyrokinetic Toroidal Code, GTC [LHL+98], is a highly scalable scientific application that simulates billions of plasma particles inside a reactor. S3D [LYC+18] is a direct numerical simulation (DNS) code used in computational fluid dynamics, which solves the Navier-Stokes equations. LAMMPS [PCT07] stands for Large-scale Atomic/Molecular Massively Parallel Simulator and is used to model particles at the mesoscale or continuum levels. HACC or Hardware/Hybrid Accelerated Cosmology Code [HPF+16] is used to conduct high resolution simulations of cosmological structure for modern-day galactic surveys. IOR is a benchmarking application developed by LLNL to test the performance of parallel file systems. We use it to simulate adversarial traffic in the system allowing us to study the impact of artificially high I/O and network traffic on applications. In order to evaluate the effects of different workloads on the I/O performance of the system, we use HPC applications widely used in leadership computing facilities. We use Darshan traces collected from runs of the application to replay the expected I/O behavior on our simulated systems. These representative applications perform blocking I/O. We also created tools to weakly scale the Darshan logs to occupy a specific fraction of the system. We discuss the scaling approach in a later section.

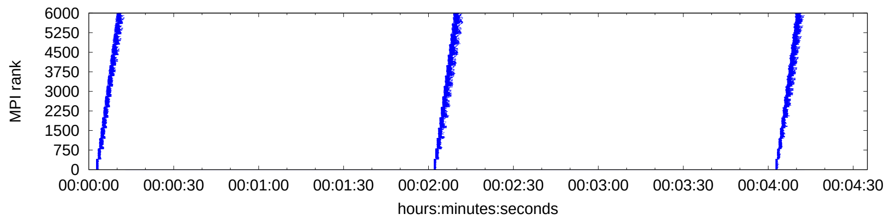
Table 2.2 lists the applications used in our workloads and their I/O properties, and Table 2.3 lists the combinations of these applications we use as representative workloads for our simulation experiments. Figure 2.4 shows the I/O patterns of our representative



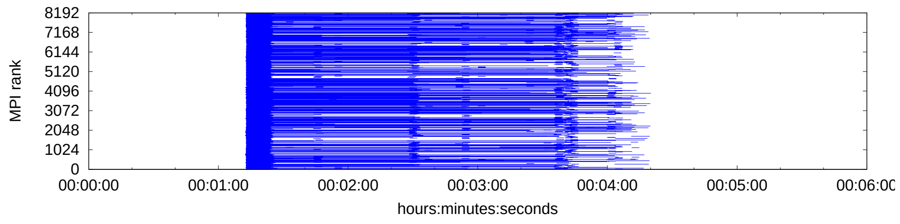
(a) GTC



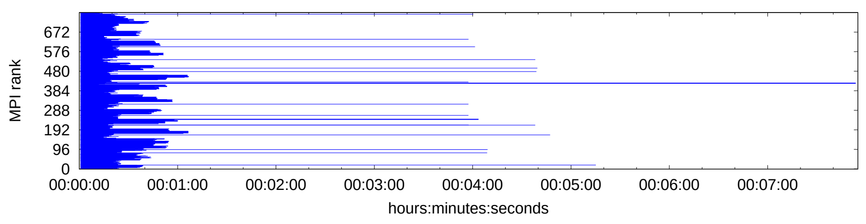
(b) LAMMPS



(c) S3D



(d) HACC



(e) IOR

Figure 2.4: File write patterns for different applications from Darshan traces.

Table 2.2: Application Traces

Application	MPI Ranks	Avg. I/O / Rank (MB)
GTC	7680	669.41
LAMMPS	21600	19.78
S3D	6000	154.49
HACC	8192	386.39
IOR	768	1024

applications as derived from the Darshan traces. GTC has a single I/O phase towards the end of the execution of the application with all ranks (except rank 0) writing very similar amounts of data to persistent storage. Each rank in LAMMPS opens several files for writing small amounts of data into each file. S3D has 3 I/O phases for each rank. Except rank 0, each of the ranks writes to an independent file in each of the 3 I/O phases. The write operations are also staggered with increasing rank numbers. For HACC, all the ranks have 2 write I/O phases to the same file. While each of the ranks open their respective files at the same time, the operations exhibit a scattered pattern with each rank experiencing a delay before write operations commence. This significantly reduces the overlap between writes from different ranks. Finally, IOR as a benchmark has a single I/O phase with each rank writing 1 GB to independent files from the very beginning of application execution. The nodes are modeled to be Summit-style (200 petaflop CPU/GPU system at OLCF) fat nodes with 6 GPUs, i.e, we allocate 6 MPI ranks to each compute node in our simulation. Our choice for application combinations were restricted by the number of nodes in the system we are simulating and memory limitations on our compute nodes.

Table 2.3: HPC workloads

Workload	Total MPI Ranks	Compute Nodes	%age of Total Nodes
GTC	7680	1280	23-27
GTC, IOR	8448	1408	25-30
GTC, S3D	13680	2280	40-47
GTC, S3D, HACC	21872	3646	64-75
HACC	8192	1366	25-29
HACC, IOR	8960	1494	27-32
LAMMPS	21600	3600	63-74
LAMMPS, IOR	22368	3728	65-77
S3D	6000	1000	17-21
S3D, IOR	6768	1128	20-23
S3D, LAMMPS	27600	4600	80-95

Augmenting Darshan Traces

More powerful machines allow application developers to increase the size of their datasets, create higher fidelity simulations, and perform analysis at finer granularity. To more effectively represent application I/O behavior in next-generation machines, we develop tools that allow us to scale Darshan application traces and add synchronization primitives to them. First, our tool allows us to scale the number of ranks in an application trace. Following the weak scaling paradigm, it uses the I/O behavior of the other ranks to scale-up the number of ranks within a job. Second, we can also scale the size of the I/O per rank in the job for strong scaling. Third, we added the ability to repeat the number of iterations that the Darshan trace is replayed for with a certain interval. And finally, we have the ability to collate several consecutive small writes in the trace into bigger ones. We can, therefore, represent future application I/O sizes while being consistent with application I/O behavior. We can also add global and sub-communicator barriers to the traces to more accurately represent I/O behavior of the application.

Job Placement

Applications in leadership-class systems are rarely allocated to a completely contiguous set of nodes. The HPC batch scheduling algorithm often results in fragmentation with several small chunks of contiguous nodes that are available. This exacerbates inter-job and intra-job interference (see Yang et al. [YJM+16]) for MPI communication. This can also have severe effects on I/O performance with interference resulting in large performance variations for non node-local BB configurations and the PFS. We use job allocation logs from the Titan supercomputer at OLCF to guide the allocation of applications to nodes in our workload. This allocation strategy lets us evaluate the impact of application I/O patterns not only on interference in the network but also in terms of interference between SSD devices of BBs.

2.6.3 System Configuration

In this section, we describe the systems we simulate using the CODES simulation suite. We broadly categorize the systems based on their network topology. For each of the network topologies, we create configurations for the different BB architectures subject to evaluation.

Fat-Tree Network

We use the fat-tree model in CODES to configure a fully non-blocking fat-tree network resembling Summit. Our configuration is a 3 level fat-tree with 270 edge switches (radix of 36) and up to 4860 terminal nodes. Like Summit, our configuration has a network bandwidth of 25 GB/s per node. For the node-local BBs, each of the terminals has compute nodes with a local SSD. The grouped configuration has one BB server per edge switch, and the global BB configuration has 15 edge switches with only BB servers for a total of 270 BB servers. For effective comparison we have both the configurations, grouped and global, with the same number of BB nodes. Furthermore, all the 3 configurations have nearly identical aggregate bandwidth (within 1% of each other.)

Dragonfly Network

We use the dragonfly model in CODES to simulate a Cray XC30 system along the lines of the Edison supercomputer at NERSC. It uses an Aries-style interconnect for a total of 5,760 terminal nodes with a bandwidth of 8 GB/s per node. The local and global channels are configured with a bandwidth of 5.25 GB/s and 1.5 GB/s, respectively. We use adaptive routing which has been shown to perform better than minimal routing for adversarial traffic patterns [KDSA08]. Here, the node-local BBs are also situated on each compute node. The grouped configuration has a BB for every 2 blades in the network, where each blade consists of 4 terminal nodes connected by an Aries SOC. The global BB configuration consists of 180 blades with BB servers.

A 2:1 Tapered Fat-Tree Network

We also use the fat-tree model in CODES to configure a 2:1 tapered fat-tree with twice the number of links for terminal nodes as the links going to the upper-level switches. This reduces the cost by reducing the number of switches and cables required for the same number of terminal nodes. Tapering of a fat-tree might negatively impact performance of the network as it is no longer non-blocking. To facilitate a comparison with a full non-blocking fat-tree network, the 2:1 tapered fat-tree retains the other configuration parameters from our fat-tree configuration.

BB Nodes

Node-local BBs are simulated as a single SSD device per compute node with write and read bandwidths of 1,400 MB/s and 2,300 MB/s, respectively, for both dragonfly and fat-tree networks. The grouped and global BB nodes consist of 18 SSDs each for the fat-tree configuration and 6 SSDs each for the dragonfly configuration. The aggregate write and read bandwidth of BB nodes is 25,000 MB/s and 41,000 MB/s in the fat-tree configuration and 8,400 MB/s and 13,700 MB/s in the dragonfly configuration, respectively, for both the grouped and global BB models.

2.7 Evaluation

We evaluate the performance of different workloads subject to a range of factors. We run 100 simulation experiments on an HPC cluster using as input application traces collected from runs on the Titan supercomputer at OLCF [Lab21d].

First, we look at how the performance of single job workloads are impacted by the choice of BB architectures. We compare the architectures across the different network topologies based on total time spent on I/O, network hops and BB performance. We then study the impact of adversarial jobs on our representative workloads. This involves a workload co-scheduled with an IOR job, which periodically writes 1 GB files to BBs, enabling us to gauge performance in a worst-case scenario. Finally, we study the impact of co-scheduling a combination of representative jobs on performance parameters.

2.7.1 Validation of Node-Local BB Model

To validate our node-local BB model, we run the IOR benchmark on Summit-style [Lab21c] nodes in a fat-tree network with node-local BBs. We run the benchmark with 96 ranks over 16 nodes, and compare the results with those obtained from simulation. Each rank writes 1GB of data to independent files on their local BBs. We also set the parameters of our simulated BBs to match the configuration of the actual devices. We set the read/write bandwidth to 3,200 MB/s and 2,100 MB/s respectively with negligible seek overhead.

Table 2.4 shows the comparison between the simulated and actual run of the IOR benchmark on 16 nodes. We can see that the write time and bandwidth obtained from the simulation deviates from the actual results by less than 3%. This validates our model to simulate node-local BBs with considerable accuracy.

Table 2.4: Node-Local BB Validation

Parameter	Actual	Simulated
Size/Rank (GB)	1	1
Aggregate Size (GB)	96	96
Write Time (s)	3.01	2.92
Aggregate Bandwidth (MB/s)	32600.56	33616.48

2.7.2 Validation of Distributed BB Models

We validate our distributed BB models (which combines grouped and global from Fig. 2.3 for now) by running the IOR benchmark on the Konrad [Ber21] TDS (Test Development System) for 6 - 192 ranks over 64 compute nodes. The Konrad TDS has an Aries Dragonfly network topology and distributed BBs. We use their BB configuration within CODES for a write bandwidth of 1,900 MB/s and $15\mu s$ of write overhead, based on a single node with 6 ranks writing to one SSD of the BB, in order to validate against the performance of the actual machine. Figure 2.5 shows the comparison between the actual and simulated runs of the IOR benchmark. The results from the simulation deviate from the actual runs by less than 11%. This validates our CODES model to simulate global and grouped distributed BBs accurately.

Cost Modeling

In a cost-benefit analysis, we consider DOE models of workloads for supercomputing centers. Capability computing centers cater to a workload that requires 20% or more of the resources in a supercomputer to achieve the desired science. These centers tune scheduling to prioritize large jobs to access the machine. These workloads often use file-per-process outputs to mitigate performance degradation due to parallel file system locking at scale. Capacity computing center resources are targeted toward smaller average job sizes dominated by single-shared-file outputs. Smaller jobs are less impacted by file system overheads and single-shared-files do not require post-processing like file-per-process.

We make the following assumptions for cost modeling that are based on Summit [Lab21c]. We assume that distributed BB file-per-process performance and single-shared file performance are equivalent, and there is no locking overhead. We assume that there is no opportunity to perform single-shared file operations on node-local systems. Using Summit's storage performance numbers at 2.5 TB/s for PFS and 9.7 TB/s for the node-local BB, a 35% memory checkpoint takes approximately 5 minutes to the parallel file system and

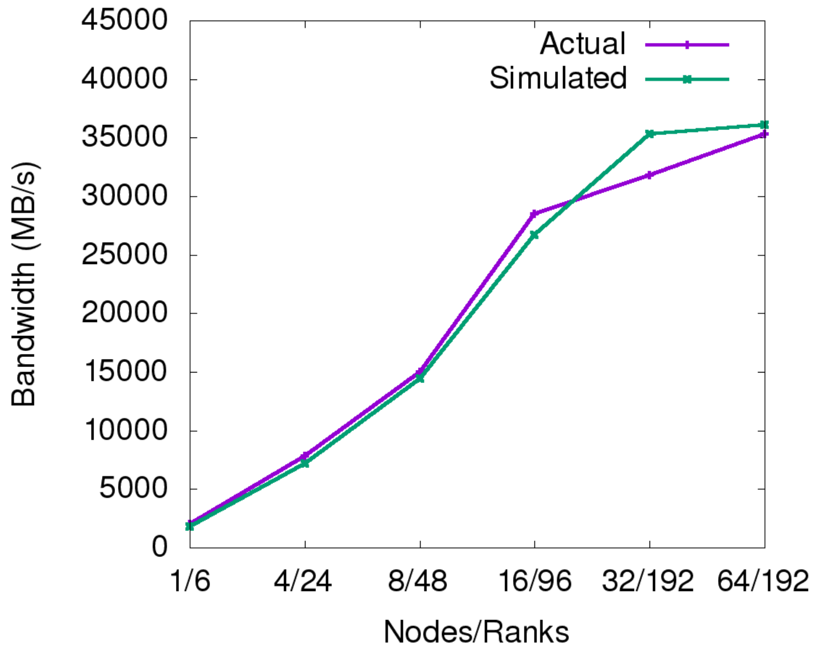


Figure 2.5: Distributed BB Validation

2 minutes to the burst buffer based on related work [VdSB+18]. Over the course of a year, such workloads in a capacity center result in 100% acceleration of the I/O portions of the workload by 2.5X. In contrast, the capability center accelerates the I/O portions by 1.8X as only 70% of the I/O workload can be accelerated (on the BB side).

Table 2.5 compares Capital Expenditure (CAPEX) and the Operational Expenditure (OPEX) of the node local and distributed models using current hardware. The numbers presented are a simplification of actual OPEX values and do not account for voltage scaling during idle phases. These approximations also omit the OPEX maintenance costs that are necessary for running such systems. The numbers demonstrate that the distributed model's CAPEX is 2.4X higher than the node-local model. The energy-based OPEX is also higher, consuming over double the power due to CPU, network, and infrastructure overheads. Compared to node local BBs, a capacity center using a distributed BB would more than double the CAPEX and OPEX. A capability center (same CAPEX+OPEX) would only benefit from 70% of the I/O being accelerated, still making distributed BBs less appealing.

Table 2.5: Distributed vs. Node-Local Capital and Operational Expenditure Comparison

Node Local CAPEX	Total Cost	$4,608,000
4608	x8 NVMe Devices	$1000
Distributed CAPEX	Total Cost	$11,415,600
4832	x4 NVMe Devices	$800
302	AMD EPYC Servers	$25,000
Node Local OPEX	5 Yr Cost	$1,000,000
NVMe	4608	25W
	Total	115KW
Distributed OPEX	5 Yr Cost	$2,600,000
NVMe	4832	25W
Server	604	300W
	Total	302KW

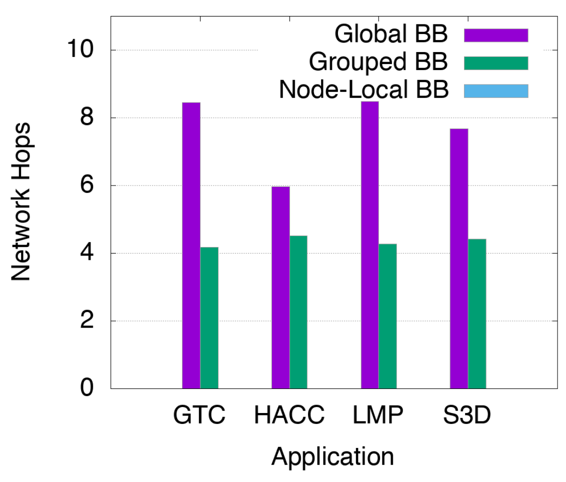
2.7.3 Effect of BB Placement on Performance

We simulated the 4 representative jobs (GTC, HACC, LAMMPS and S3D) running independently on the system for each BB architecture and network topology. The node allocations are not contiguous but guided by allocation logs from the Titan supercomputer, which are typically not contiguous in node locations. This provides a realistic comparison when we introduce adversarial jobs as well as co-schedule other applications as part of the workload.

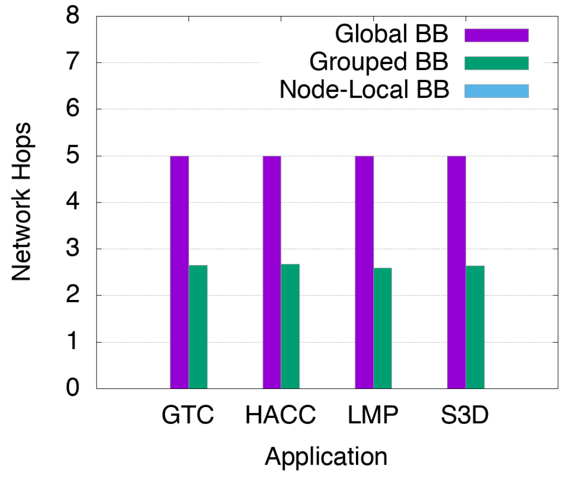
Figure 2.6 shows the average number of hops incurred by I/O traffic for each of the given applications across the different BB architectures. Since node-local traffic does not have to traverse the network, the number of hops for node-local BB model is 0. For the fat-tree network, all traffic to the global BBs is routed across the entire diameter of the network. For the grouped BBs traffic is routed to a combination of the BB servers in the local group as well as adjacent groups because of the striping policy.

Observation 1: For bursty traffic in a dragonfly network with global BBs we observe high non-minimal routing, with I/O traffic incurring significantly higher hops.

For the dragonfly network we see significant differences between applications for the global BB architectures. GTC exhibits highly bursty traffic, performing significant I/O in a relatively short duration of time. This causes a higher fraction of the packets to be routed non-minimally compared to HACC, which exhibits staggered rather than bursty behavior. The grouped architecture, on the other hand, does not concentrate traffic in any one part of the network and, therefore, does not suffer from excessive non-minimal routing. Dragonfly networks attempt minimal routing first, followed by non-minimal routing. Therefore, high non-minimal routing indicates congestion in the network.



(a) Dragonfly



(b) Fat-Tree

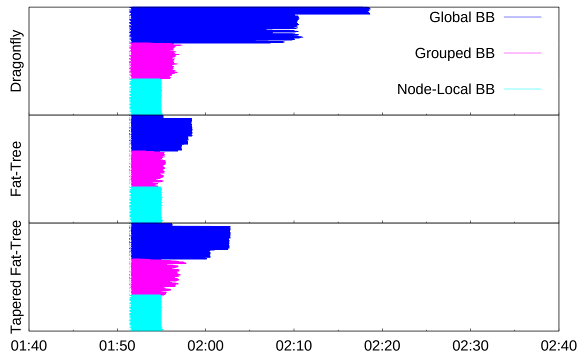
Figure 2.6: Avg. number of network hops incurred by application's I/O traffic

Observation 2: Grouped BB in a dragonfly topology perform their I/O phases significantly faster than the global configurations for bursty applications.

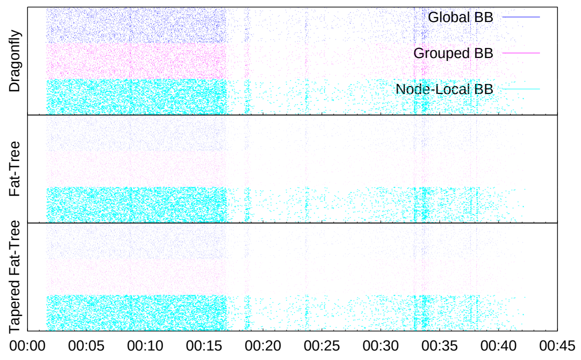
Figure 2.7 shows the I/O phase of each rank of the application on the y-axis (from highest rank on the top to lowest rank on the bottom) over time (x-axis) for a single I/O phase. As expected, the performance of the node-local BBs is independent of the choice of network topology across all applications. Ranks in the grouped architecture spend consistently less time in their I/O phase than with the global BB architecture for the three network topologies. The difference is most pronounced for the dragonfly topology with bursty applications. This is due to the lower global bandwidth on the dragonfly topology compared to the fat-tree with full bisection bandwidth. In case of HACC, not all of the ranks attempt write operations at the same time. This limits the network contention in the part of the network where the storage resources are concentrated.

Observation 3: Global, grouped and node-local BB configurations have comparable performance for fat-tree networks except for applications with staggered I/O like S3D.

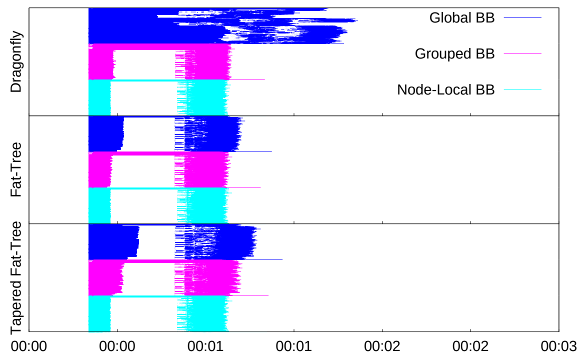
In the case of fat-tree topology, all applications except S3D spend comparable amounts of time in their I/O phases for the three BB architectures. For S3D, the grouped and global configurations outperform the node-local BB configuration for both the full fat-tree and the 2:1 tapered fat-tree networks. Since only a fixed number of ranks in S3D (400 in this case) perform I/O at any given time, there is more available bandwidth in the case of global and grouped BB configurations, while the application is bottle-necked by the single SSD



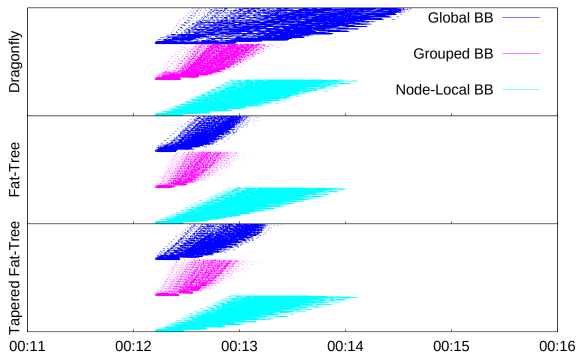
(a) GTC, x-axis from 1min40sec to 2min40sec



(b) HACC, x-axis from 0 to 45 seconds



(c) LAMMPS, x-axis from 0 to 3 seconds



(d) S3D, x-axis from 11 to 26 seconds

Figure 2.7: I/O phases of each application across network topologies and BB architectures, y-axis from highest (top) to lowest (bottom) rank

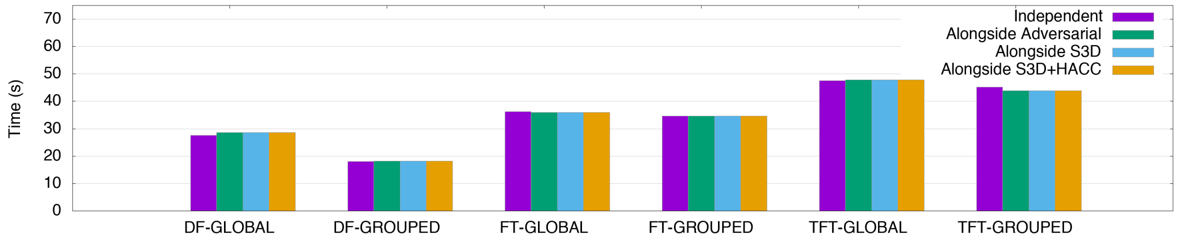
in the node-local BBs. The tapering imposes a performance penalty on both grouped and global BB configurations, but the network contention between the edge and aggregation layers results in the global configuration paying a significantly higher penalty compared to the grouped BB configuration.
Observation 4: Node-local BBs outperform the other configurations for bursty applications. In case of applications with a scattered I/O phase like HACC, the node-local configuration results in each rank spending more time on I/O.

We can also see from Figure 2.7 that node-local outperforms grouped and global BB architectures for bursty applications like GTC and LAMMPS. This is because (a) the SSD is local to the compute node and I/O does not incur slowdown due to network congestion, and (b) there is much less contention for the SSD device compared to grouped and global BBs. For HACC, results show that each rank in the node-local configuration spends significantly more time on I/O operations compared to grouped or global BBs but the resulting I/O phase lengths are almost identical irrespective of BB placement. This is because while all ranks in the application open their respective output files at the same time, they commence write operations only after a delay of anywhere between 1 to 40 seconds. Due to this scattering of write operations by ranks, network and storage contention is reduced drastically. Since the node-local BBs have lower bandwidth compared to BB servers in the grouped or global architectures, each rank spends more time on I/O. In this case, HACC has a schedule that perfectly matches bandwidth restrictions of the grouped and global models, as a mismatch would have caused performance to be adversely affected, which is not the case. We conjecture that an I/O runtime that manages scheduling of I/O operations can potentially benefit grouped and global BB architectures.

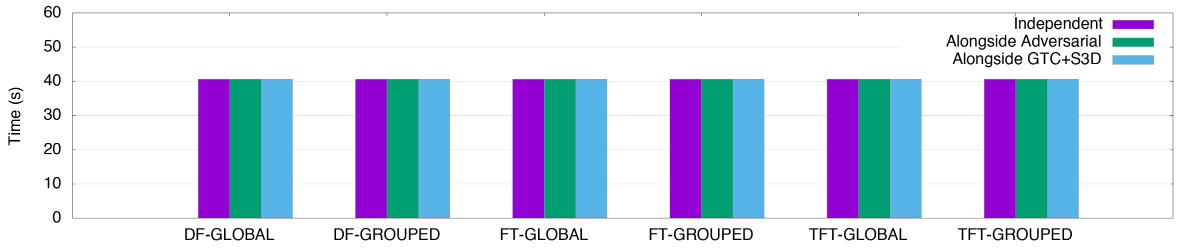
Our framework can also model different storage devices like Phase Change Memory (PCM) and 3D XPoint. To assess diverse BB devices, we experimented with a cost (overhead) of $45\mu s$ per operation for BBs (graphs omitted due to space). In this case, node-local BBs significantly outperform the other configurations for bursty applications and staggered applications like S3D. Furthermore, global and grouped BBs in a fat-tree configuration had similar performance across applications for storage devices with seek and operational overhead.

2.7.4 Effect of Co-Scheduled Jobs on Performance

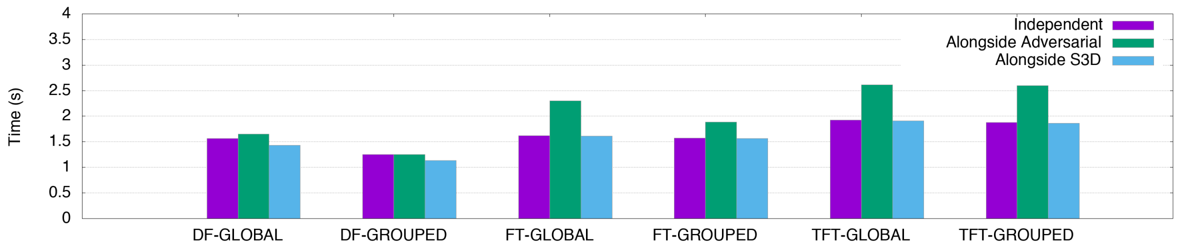
We also simulate the performance of the representative applications when scheduled alongside other applications plus an adversarial one. Here, the simulated storage device has



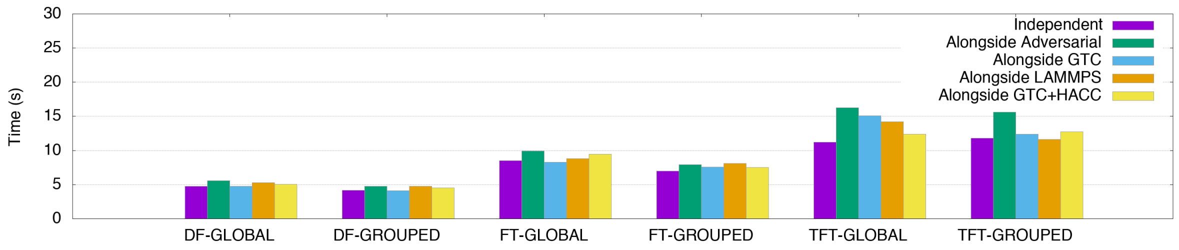
(a) GTC



(b) HACC



(c) LAMMPS



(d) S3D

Figure 2.8: I/O phases of each application with adversarial background workload across network topologies and BB architectures.

additional read and write overheads of $45\mu s$ per operation. This allows us to experiment with emerging storage technologies that have different characteristics than current generation SSDs (without measurable write overheads). We use traces of IOR, an I/O benchmark generating large volumes of data at specified intervals to simulate the adversarial traffic. The node allocation strategy causes co-scheduled jobs to be assigned to certain nodes adjacent to the nodes of the application we are studying. Since the co-scheduled jobs have no effect on the performance of the node-local BBs, we show results only for global and group-local BBs.

Figure 2.8 shows the effect of scheduling multiple jobs on application workloads. The different configurations are specified by network type and BB architecture. DF, FT, and TFT refer to the dragonfly, fat-tree, and tapered fat-tree networks, respectively. The figures show the total duration of the I/O phase (in seconds) on the y axis for each application, rather than on a per-rank basis.

Observation 5: Applications with I/O phases that do not overlap significantly have little to no effect on the application's I/O phase length even for bursty applications.

For GTC, the I/O phase of the application has little overlap with the traffic from co-scheduled jobs. This minimizes interference between jobs. Even though some ranks complete their I/O phases sooner and others later, the average time spent by each rank is not significantly affected. Even though individual ranks spend more time in I/O for certain ranks of the GTC application, the overall I/O phase remains similar.

Observation 6: The I/O phase lengths of non-bursty/scattered applications are not significantly affected by co-scheduled jobs, though there is an increase in the average time spent in I/O by each rank.

For HACC, the scattered I/O pattern results in ranks adjacent to the nodes running other jobs spending more time on their respective I/O phases, especially for fat-tree and tapered fat-tree topologies, which is not visible from the aggregate plots. The overall I/O phase of the application remains unaffected due to the highly scattered nature of I/O operations even as the average time spent in the I/O phase increases. Since the I/O behavior of HACC is not bursty, it is almost unaffected by the I/O behavior of other applications running on the system.

Observation 7: Bursty applications in fat-tree and tapered fat-tree networks are significantly affected by co-scheduled adversarial jobs due to contention for both storage and network resources.

LAMMPS, on the other hand, experiences significant interference from the adversarial job. The I/O phase of ranks adjacent to nodes allocated to the adversarial job increases significantly, causing the application to experience slowdown. While the interference is experienced across BB architectures and network topologies, the effect is most pronounced for the fat-tree networks. In case of the tapered fat-tree network, the limited network bandwidth from the edge to the aggregate layers results in similarly degraded performance for both the global and the grouped BB configurations. The bursty nature of LAMMPS coupled with its I/O phase coinciding with that of the IOR job results in contention for both storage and network resources. Although experiencing significant slowdown due to IOR, LAMMPS is not at all affected by the a co-scheduled S3D job as their I/O phases do not coincide.

Observation 8: The I/O performance of applications with custom patterns, like S3D, is dependent on the node allocation policy employed by HPC centers.

S3D experiences significantly degraded performance due to an IOR job being scheduled along side S3D, especially for the fat-tree networks. Just as for LAMMPS, network contention in the tapered fat-tree network causes similarly degraded performance for both the global and grouped BB configurations. We can also see from the figure that the tapered fat tree configuration imposes the highest penalty compared to the dragonfly network, where the performance degradation is the least. This indicates high network contention rather than contention at the storage resources. Furthermore, a co-scheduled GTC job has little to no effect on the I/O performance of S3D in dragonfly and fat-tree networks, but significantly affects performance for both global and grouped BB configurations in the tapered fat-tree network. Due to the nature of I/O in S3D, if the co-scheduled job is allocated nodes adjacent to the higher S3D ranks, the long tails of those ranks would result in increased I/O phases of the entire application.

Finally, we also perform experiments with SSD devices with 0 seek and operational overhead. Our preliminary results show that HACC is not significantly affected by the storage device model because of the scattered I/O pattern. Similarly, LAMMPS's I/O phase length decreases, reducing the overlap with the adversarial jobs, and is therefore not less affected by the adversarial job as in Fig. 2.8, which includes 25μs and 20μs seek and operational overheads.

2.8 Conclusion

We have developed a simulation framework for the provisioning of burst buffers in supercomputers to provide accurate, multi-tenant evaluations of realistic application and storage workloads. This allows us to compare multiple network and BB configurations as a means to select a best configuration for procurement of an HPC system based on the target workloads to run. Our experiments indicate difference depending on application characteristics, such as I/O burstiness, BB placement, overlap of I/O phases, background workloads, and network topology with an intricate interplay, all of which aid in ultimately deciding on a network topology and BB placement.

CHAPTER

3

PREDICTING THE IMPACT OF GPUDIRECT ON MULTI-NODE APPLICATION PERFORMANCE

3.1 Introduction

The use of general-purpose graphics processing units (GPGPUs) in modern high performance computing (HPC) systems is becoming increasingly popular, with six of the ten fastest machines on the Top500 list using GPUs to speed up computation in November 2020 [Top20]. The continued improvement in GPU communication technologies has made GPUs first-class computation devices with the ability to directly exchange messages with one another, both within and across compute nodes. Hardware for data movement to and from GPUs such as NVLink and enhancements in system software such as GPUDirect has reduced latencies, increased throughput, and eliminated redundant copies during data movement between different system components.

While GPUs on HPC systems have dramatically increased a node's computing capabilities, the interconnect bandwidth per flop/s has not increased at the same rate even on high-end systems such as Summit and Sierra. The objective of this work is to explore the impact of using a large HPC system with multi-GPU nodes and modern GPU communication paradigms such as NVLink and GPUDirect on the messaging performance of scientific applications. Moving data directly between GPUs using technologies such as GPUDirect requires modification to the application code. To assess the performance impact of making such changes before the developer makes extensive modifications, we use parallel discrete event simulations (PDES) to study such what-if scenarios.

Most current network simulators treat the compute node as a black box and only model network communication to and from the node. In order to simulate direct communication between GPUs, GPUs need to be treated as first class objects in the network simulation. We use the TraceR-CODES [AJB$^+$15] [CLL$^+$11] simulation framework that replays MPI execution traces using PDES to predict the communication performance of parallel codes. Explicitly simulating communication between the GPU and the network requires modifications to the traces used for simulation and also to the network simulation framework. We extend the Score-P OTF2 library [EWG$^+$11] to annotate MPI calls with locations of the MPI send/receive buffers, which allows us to identify whether the buffers are in GPU memory or host memory. This allows us to predict the performance of the code if the buffers were read from GPU memory instead of host memory for instance.

We also extend TraceR-CODES to explicitly model GPUs, and communication between GPUs or between the host and GPUs. We model both point-to-point and collective MPI communication using events between CPUs and GPUs. We add functionality to replay any MPI messages that send data from CPU buffers as though they were sending data from GPU memory instead, using GPUDirect communication. With this added what-if functionality, application developers can evaluate the performance of an application under various node, system, and interconnect configurations. This enables them to direct their coding efforts in deciding if they should enhance their applications with GPUDirect. Further, these novel capabilities can help system designers in making decisions about system procurement. They can use traces from applications that are important to the HPC center to simulate their performance across a wide range of network topologies, node configurations and system parameters before making procurement decisions.

We utilize these what-if capabilities to evaluate how application performance differs when using different communication paradigms, and identify the main factors contributing to GPUDirect benefits today. We also identify communication patterns largely unaffected by GPUDirect today, which can help in procurement, application tuning, and future enhancements of GPUDirect. Several scientific applications are yet to take advantage of GPUDirect. For those, our results provide guidance in determining the potential impact of GPUDirect before investing time into code refactoring.

The primary contributions of this work are as follows:

- We extend the Score-P OTF2 library to add annotations to MPI calls in OTF2 traces that identify the origin of source / destination buffers, thereby explicitly recognizing GPUDirect communication in application traces.
- We extend the TraceR-CODES simulation framework to model GPUs explicitly as first-class independent communicating devices along with modeling the associated GPUDirect and NVLink communication for point-to-point messages and collectives.
- We add the ability to explore the impact of these modern communication paradigms on applications that are yet to take advantage of them.
- Finally, we perform simulations using traces from proxy applications, and explore and analyze the predicted performance across a variety of node, system, and application configurations.

3.2 Background

We first provide a brief overview of modern GPU-based architectures, and the TraceR-CODES simulation framework used for performance predictions in this work.

3.2.1 GPU-based Node Architectures

Over the last decade, GPGPUs have been used to drive scientific computations and have led to tremendous improvements in performance and energy efficiency, particularly for numerical kernels [OHL+08, KES+09]. The development of frameworks and languages, such as NVIDIA's Compute Unified Device Architecture (CUDA) [Nvi07] and the Open Compute Language (OpenCL) [SGS10], have further contributed to establishing GPUs as first-class computational devices in HPC systems. Recently installed supercomputers such as Sierra and Summit rely predominantly on GPGPUs for their peak flop/s. The logical design of a node on the Sierra system is shown in Figure 3.1, featuring four high-end GPUs connected to two Power 9 CPUs via NVLink. The CPUs have access to a non-volatile memory (NVMe) device serving as a burst buffer. Summit has six GPUs per node in a similar design.

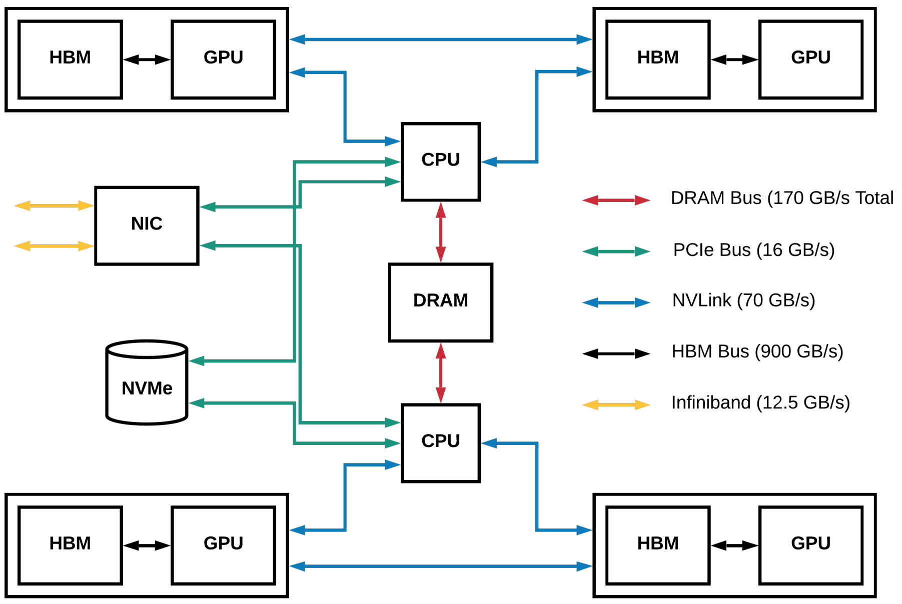


Figure 3.1: Organization of a compute node on the Sierra supercomputer at LLNL.

The trend of delegating more and more computation to GPUs has spurred the development of modern interconnects such as NVLink [FD17] and communication methods such as GPUDirect [SAL+11]. While previous generations of GPUs were connected over slower Peripheral Component Interconnect Express (PCI-E) interfaces (green lines in the figure), newer GPUs with NVLink (depicted using blue lines) significantly increase throughput between GPUs, and GPUs and CPUs on a node. Prior to GPUDirect, data subject to communication between GPUs on the same node (over the node-local interconnection network) or between GPUs on different nodes (via the network passing through the Network Interface Card (NIC)) required multiple copies, i.e., from GPU HBM memory via NVLink and over the memory bus (red) to the host DRAM on the sender, and from the host DRAM to the GPU HBM on the receiver. We refer to this mode as **HostCopy** throughout the thesis. The GPUDirect communication protocol avoids these extra copies by enabling data to be sent directly to the NIC from the GPU device's DRAM via NVLink and PCI-E. The GPUDirect transfers are still initiated by the CPU but the memory copies between the CPU DRAM and GPU HBM can be avoided.

3.2.2 Parallel Discrete Event Simulation

Parallel Discrete Event Simulation (PDES) uses a sequence of events to model a system with each event resulting in a change in the state of the system. Between these events which trigger state changes, the system is assumed to be in a constant state. While PDES can be used to model a wide variety of systems, it has been used extensively in existing literature to model HPC networks.

In the context of this work we use PDES to model our HPC interconnect with packet level precision. So each packet in our simulation is represented by an event which results in a change in the state of the system. As already mentioned earlier PDES has been used extensively in prior work to model HPC networks [LHSJ15, WCM$^+$16, MCRC16, KZM$^+$19].

3.2.3 The TraceR-CODES Simulation Framework

The TraceR-CODES (https://github.com/hpcgroup/TraceR) simulation framework (Figure 3.2) can be used to predict the performance of production applications by replaying their MPI execution traces on a variety of network topologies and system configurations. CODES performs packet-level simulations of communication in parallel workloads running in HPC environments. CODES provides models for several network topologies such as fat-tree, dragonfly, torus etc. ROSS is the discrete-event simulation engine used by TraceR-CODES. ROSS models each component in the system (MPI processes, switches, etc.) as a Logical Process (LP) that communicates with other LPs using time-stamped messages. We go into detail about some of the components of TraceR.

CPU LPs: Each CPU LP in TraceR acts as an MPI rank for an application. A list of timestamped MPI operations, usually read from a trace file, is associated with the CPU LP. These LPs communicate with each other through the network model using events during simulation. They also model compute loops by waiting for the appropriate time read from the trace.

MPI Model: The MPI model in TraceR is responsible for modeling the MPI layer of an application. It dictates the protocol to utilize for point-to-point MPI messages (explained in a later section), the algorithms for collective communication, and the expected behavior for synchronous and asynchronous sends and receives. While the CPU LPs talk to the MPI model, the MPI model itself communicates with the underlying network model to affect MPI communication.

OTF2 Traces: Simulations using TraceR-CODES require capturing MPI execution traces for

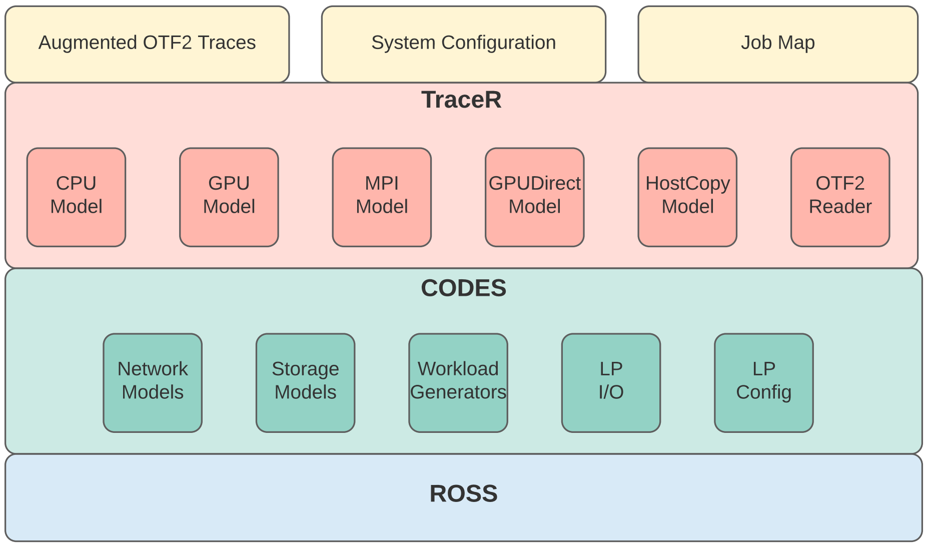


Figure 3.2: The TraceR-CODES simulation framework

each parallel application. In order to collect these traces, we use the Score-P library that generates traces in the OTF2 format [KRaM+12].

OTF2 Reader: The OTF2 Reader in TraceR takes the application trace file as input and associates operations in the trace file to each of the MPI ranks (CPU LPs). For each MPI operation it records the necessary information in order to replay the operation during simulation. The type of MPI operation, the destination/source for MPI Send/Receive messages, the root for collective communication, the size of the payload, request numbers for asynchronous communication, timestamps, etc. are read from the trace file into operation records for each operation and rank.

3.3 Simulating Communication on GPU-based Parallel Systems

In the current implementation of TraceR, GPUs and their communication are not modeled explicitly. For this study, to be able to model the impact of NVLink and GPUDirect technologies, we had to add the notion of independent GPU LPs to the simulation framework. Since TraceR uses OTF2 traces to simulate applications, we also need to extend Score-P and OTF2 traces to record additional information that can distinguish GPUDirect MPI calls from regular HostCopy MPI calls. Below, we describe the changes to Score-P and the OTF2 traces, the model of the GPU LP and the associated operations and communications.

3.3.1 Score-P and OTF2

We extend the Score-P MPI backend to query the CUDA runtime for the location of the buffer pointer passed to the MPI call and then annotate the OTF2 trace file with information about the location of this buffer. We annotate each MPI event with a flag denoting whether the buffer was in CPU memory or GPU memory. On the simulator end, we enhance the OTF2 reader in TraceR. During initialization of the simulation, the tasks for each server LP are read from the trace file, flagging each operation as either a regular MPI call or a GPUDirect call using the annotations.

Further, we enhance TraceR-CODES to provide the capability to replay regular MPI HostCopy calls as GPUDirect calls, which enables us to simulate GPUDirect communication using traces that were collected for a non-GPUDirect version of the code. This is useful when applications have not implemented GPUDirect, i.e., their traces only contain regular MPI message events but not GPUDirect events. This capability enables application developers to conduct a what-if study assessing the potential benefit they could get by implementing GPUDirect in their application. In order to compare the performance of GPUDirect with HostCopy, we also model the time required to copy the buffer from the GPU to the CPU. This involves modeling the cost of copying the data from GPU buffers in the GPU HBM to the CPU DRAM via NVLink. This enables us to simulate HostCopy.

3.3.2 The GPU Logical Process

We augment the server LP, which represents an MPI process in our simulation, with an additional GPU LP. There is a one-to-one mapping between a server LP and a GPU LP, which gives each MPI process exclusive control over its GPU. This means that in the current implementation, each MPI rank can only control one GPU. So, if there are n GPUs on a node, we simulate n MPI processes on it. As the simulation proceeds, server LPs are assigned events by the simulator from the execution traces during simulation initialization. The GPU LPs wait to receive events from other LPs to execute their corresponding tasks.

During the simulation initialization process, TraceR reads the respective tasks that need to be assigned to each process or server LP. The GPU LPs are not assigned tasks during initialization but receive events from either their associated server LP or other GPU LPs, both inter-node and intra-node. The time between these communication triggered events is used to model either an idle GPU or a GPU engaging in active computation. The parameters added to TraceR-CODES to support GPU LPs and modeling of their associated communication are shown in Table 3.1.

Table 3.1: New Parameters for TraceR-CODES supporting GPUs

Parameter	Description
gpu_copy_enabled	Enable HostCopy simulation
gpudirect_enabled	Enable GPUDirect simulation of regular MPI calls
gpu_copy_delay	Latency for cudaMemcpy
gpu_copy_per_byte	Copy cost per byte for cudaMemcpy
gpudirect_delay	Cost of GPUDirect memory pinning

3.3.3 Simulating HostCopy

The simulation of HostCopy communication uses the packet-level modeling already implemented in TraceR albeit with the added time to copy the message buffer from the host DRAM to the GPU, or vice versa. An MPI operation is recorded as a HostCopy operation when the OTF2 reader in TraceR determines that the location of the message buffer is the CPU. We add parameters to denote the latency and per-byte cost of cudaMemcpy calls (Table 3.1). In the simulation, this delay is added to the messaging time to facilitate comparison with GPUDirect performance. Apart from this delay, the simulation proceeds as if simulating regular MPI communication.

3.3.4 Simulating GPUDirect

MPI implementations use two different protocols for send/receive operations based on the size of the message: eager and rendezvous. The eager protocol allows a sender to send the message to the receiver without having to receive an acknowledgment from the receiver. The eager protocol is used for smaller message sizes (decided by the EAGER_LIMIT parameter in MPI). The rendezvous protocol is used for larger message sizes and requires the sender to receive a notification from the receiver that the corresponding receive has been posted and a buffer is available for the incoming message.

As we had obtained traces on Lassen under Open MPI, we also chose to model our GPUDirect simulation based on the implementation of Open MPI. In conformance to the Open MPI implementation [Ope21], we do not use the eager protocol for GPUDirect communication. Based on the MPI configuration on Lassen at LLNL, we model all GPUDirect communication to use the rendezvous protocol.

We extend TraceR to add various GPU LP-related events to enable simulating GPUDirect communication. When TraceR encounters an MPI operation in the OTF2 trace with the location of the buffer specified to be in GPU memory, or when we wish to redirect regular MPI calls as GPUDirect, TraceR uses the GPUDirect model for MPI messaging. These events support coordination between the GPU LPs and their associated CPU LPs:

1. GPU_SEND: This event is used by the sender server LP to inform its associated GPU LP that it should initiate a GPUDirect send.

2. GPU_RECV: The GPU_RECV event is used by the sender GPU LP to inform the receiver GPU LP of an incoming GPUDirect message. This event also carries the MPI payload.

3. GPU_SEND_DONE: This event is used by the sender GPU LP to inform the sender CPU LP that GPUDirect communication has been completed.

4. GPU_RECV_DONE: The receiver GPU LP uses this event to inform the receiver server LP that a GPUDirect message has been received.

A timeline of the simulation of the GPUDirect rendezvous protocol as implemented in TraceR-CODES is shown in Figure 3.3. The sender server LP sends a Request to Send (RTS) to the receiver server LP. The receiver waits until a corresponding receive has been posted, after which it informs the sender that it is cleared to send the message using a Clear to Send (CTS) message. The sender server LP then informs its associated GPU LP to initiate a GPUDirect send to the receiving GPU LP. Once the message has been transmitted to the receiver GPU LP, the sender and receiver GPU LPs inform their respective server LPs that the transmission has been completed using GPU_RECV_DONE and GPU_SEND_DONE events, respectively.

In addition to simulating point-to-point GPUDirect communication, we also implement the modeling of GPUDirect for some of the most popular MPI collectives encountered in HPC applications, namely MPI_Broadcast, MPI_Reduce, MPI_Allreduce, MPI_Allgather and MPI_Alltoall. The implementation of these collectives has been facilitated by using control messages between the CPU and GPU as well as GPUDirect communication between

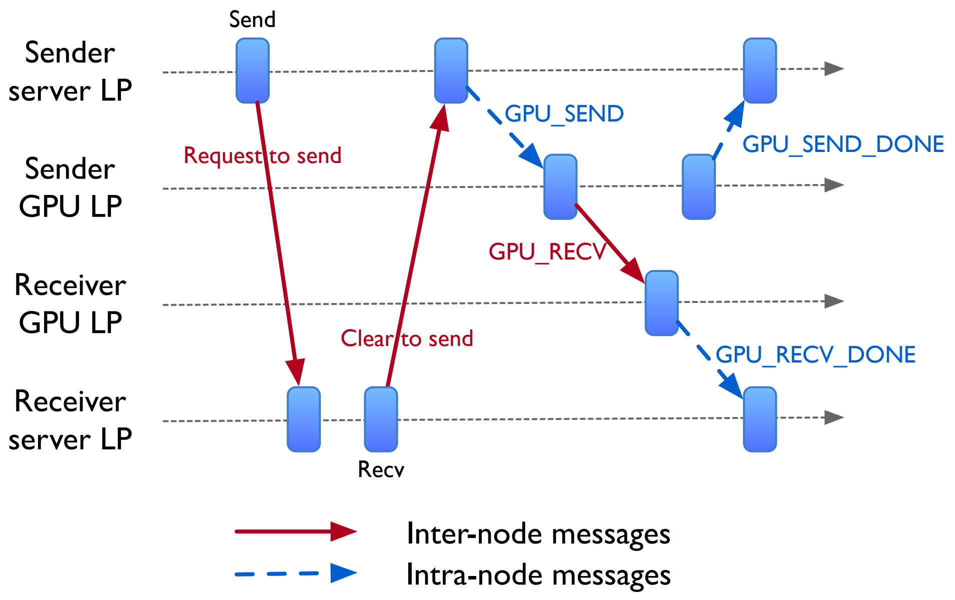


Figure 3.3: Timeline of GPUDirect messaging in TraceR-CODES.

the GPUs. We use tree-based collective implementations for modeling our GPUDirect collective communication. Our algorithms are based on the implementation of collectives in TraceR for standard MPI collectives. We do not model NCCL since the cost of collectives is dominated by off-node communication.

We add a parameter to represent the cost of pinning the GPU memory address with the GPU DMA engine in preparation of sending the message (gpudirect_delay in Table 3.1). GPU memory is pinned in BAR using the CUDA kernel driver [Inc21]. This introduces an overhead especially for small messages. To simulate NVLink communication, we use the intra-node bandwidth and node copy queues (NCQ) parameters in TraceR-CODES. The intra-node bandwidth parameter specifies the bandwidth of the intra-node bus to be used in the simulation, in this case NVLink. The NCQ parameter specifies the number of queues to be used for the intra-node communication.

3.3.5 Validation

We validate our simulation models by timing an MPI_Send/Recv between a pair of nodes on Lassen for both the HostCopy and GPUDirect scenarios for a range of message sizes. We time the MPI calls for 100 iterations and validate against the average time of the latter 90 iterations (ignoring the first 10 due to caching effects). We then simulate the traces of

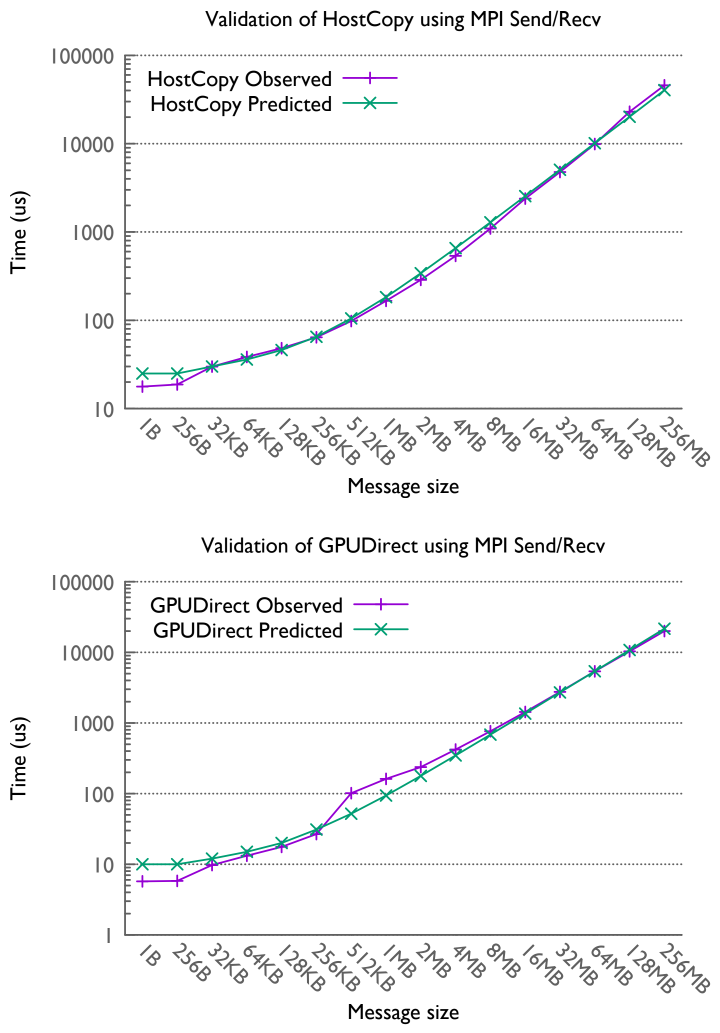


Figure 3.4: Plots showing the validation of HostCopy and GPUDirect simulations using TraceR-CODES.

the Send/Recv pairs for each message size with both our GPUDirect and HostCopy models. Figure 3.4 depicts the results of the validation experiments. Both plots use log scales on their y-axes denoting elapsed time and plot the size of an MPI message on the x-axis. We can see from the validation results that the time for Send/Recv pairs increases linearly with the size of the message. For both GPUDirect and HostCopy, we observe that our models are quite accurate with the average error for GPUDirect is less than 16% for messages that are of size 32KB or greater and for HostCopy is less than 13%. While we see a larger error for GPUDirect for small messages, they do not have a significant impact on our simulations since SW4lite and Minisweep exclusively use large messages (680KB to 11.5MB for SW4lite and 4MB to 8MB for Minisweep). Messages of size less than 32KB account for only about 4% and 0.6% of the messaging payload for AmgX and Lulesh, respectively.

3.4 Experimental Design

Let us describe the system and network used in the simulation studies, and the four applications used for generating traces.

3.4.1 Simulated GPU-based System

We set up a GPU cluster with 810 nodes and 45 leaf-level switches connected by a 3-level tapered fat-tree network [LKB+16]. This is similar to the Lassen supercomputer at LLNL, with a 1.5:1 tapering on the fat-tree. In a tapered fat-tree network, the bandwidth at higher levels of the tree is reduced while retaining the injection bandwidth at the node. For the 30-port radix switches, this means that 18 ports are used to connect nodes to a leaf-level switch and 12 ports in each leaf switch connect to higher-level switches. A dual plane EDR network is simulated, where each node has an aggregate bandwidth of 25 GB/s with adaptive routing. The packet size for the simulations is 4K. These parameters are selected to closely mirror the network parameters used in Lassen (and Sierra).

While we collect traces on the Lassen supercomputer as well as model the simulated system based on the Lassen system, we run our simulations on the Quartz system at LLNL [Lab21a]. Quartz is 3000 node machine with Intel Xeon CPUs and a Cornelis Omni-Path network.

Node Configurations: We keep the number of nodes in the system constant but vary the number of GPUs on each node – 1, 2, 4 and 6 GPUs/node. Each MPI processes gets exclusive use of its GPU. Therefore, we simulate 1, 2, 4 and 6 processes/node configurations for the different numbers of GPUs/node. This allows us to compare the performance on different GPU-based systems such as Sierra (4 GPUs/node), Summit (6 GPUs/node) and Piz Daint (1 GPU/node). We set the intra bandwidth parameter to the bandwidth of NVLink in Lassen and Sierra, which is 75 GB/s. We also set the GPUDirect delay, CUDA host copy delay and CUDA copy per byte parameters based on experiments on the Lassen supercomputer. For application runs that are not divisible by the GPUs/node, all but 1 node utilizes all of its GPUs; the last node uses only the remaining ranks. For instance, 32 ranks on 6 GPUs/node will have 5 nodes using all 6 GPUs and the last node using 2 GPUs.

Simulation Considerations: We base our simulations on the following design decisions:

1. Communication protocol: While HostCopy simulations use both the eager and rendezvous protocols based on message size, GPUDirect simulations use only the rendezvous protocol. This reflects the Open MPI implementation, which does not use the eager protocol under GPUDirect [Ope21]. Our experiments focus on bulk MPI messaging since the vast majority of HPC applications use this paradigm for communication.

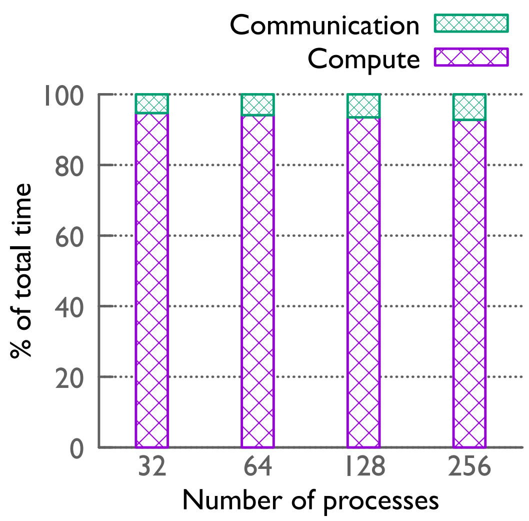
2. Exclusive GPU use: Each MPI process in our simulation has exclusive access to a GPU. While multiple processes can share a GPU, in practice, most HPC applications exclusively allocate a GPU to a process. To better utilize the GPU, applications can reduce the number of processes to match the number of GPUs, which tends to outperform scenarios where multiple processes have to contend for the same GPU. Each MPI process is allocated one core on the CPU associated with one GPU for exclusive use during trace collection and simulations in our work.

3. Process-to-node mapping: We use a linear mapping to allocate processes to nodes, where processes are assigned to nodes in MPI rank order. While certain codes may benefit from an application-specific mapping (discussed in the next section), we use a generalized mapping without requiring application-specific insight with respect to communication.

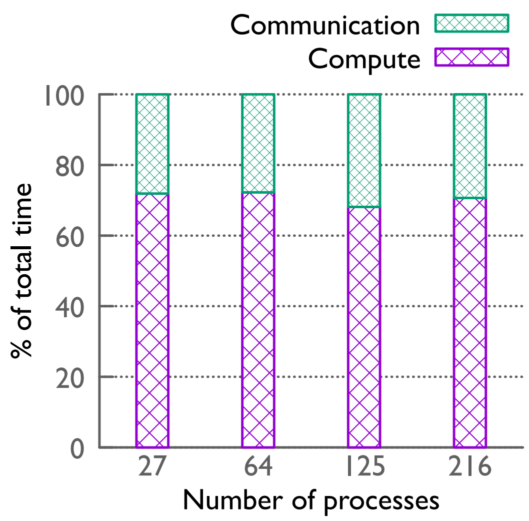
3.4.2 Applications

We collect traces from four proxy applications for the simulation experiments. All applications are run in weak scaling mode, and we only instrument the main computation loop in each application to generate traces. For each application, we collect traces on Lassen using 4 MPI processes on each node in order to provide each rank with an exclusive GPU. We describe the proxy applications and their input parameters below.

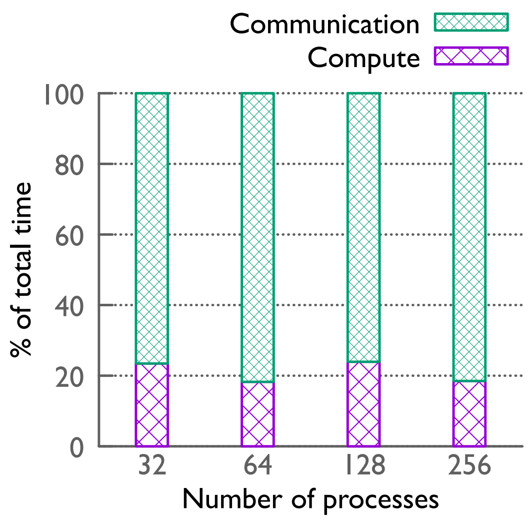
AmgX AmgX [NAC+15] is NVIDIA's version of a Distributed Algebraic Multigrid Solver Library. It is a GPU-accelerated solver for sparse linear systems. For our trace collection, we use the 7-point Poisson example application in AmgX. We collect traces for 32, 64, 128 and 256 processes while weakly scaling the application to maintain the same problem size per process. We use Vampir [KBD+08] to visualize the communication pattern from the traces we collect using Score-P (omitted due to space limitations). We see that each process



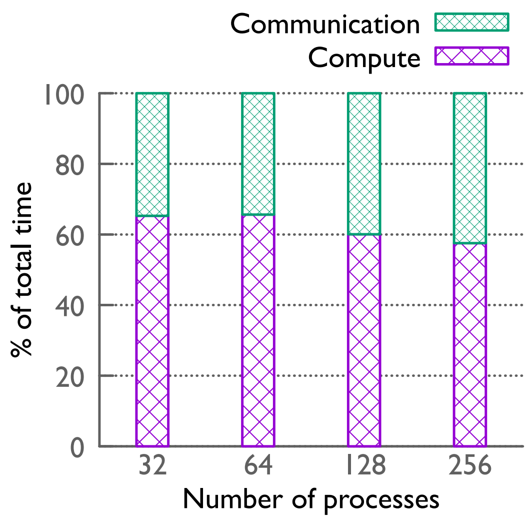
(a) AmgX



(b) LULESH



(c) Minisweep



(d) SW4lite

Figure 3.5: Time spent in computation versus communication (MPI routines) over different number of processes.

in AmgX communicates with a maximum of 4 other processes and all communicating processes exchange the same number of messages.

LULESH LULESH [KMKS13] is a shock hydrodynamics proxy application developed at LLNL under DoE's exascale co-design efforts. The number of MPI processes that LULESH can run on is constrained by n^3, where $n \in N$. Given this constraint we collect traces for 27, 64, 125 and 216 processes with weak scaling. We collect traces for a structured mesh size of 144^3 running for 100 iterations. We then repeat these 100 iterations 10 times to simulate an execution of 1000 iterations. The communication pattern for Lulesh shows that while there is symmetry in the communication pattern, each rank communicates with a different number of ranks between 4 and 26. The number of messages exchanged between the ranks is also different for different rank pairs.

Minisweep Minisweep [BDE+12] is a deterministic S^n radiation transport miniapp developed at Oak Ridge National Laboratory. Similar to AmgX, we perform weak scaling of Minisweep on Lassen and collect traces for execution sizes of 32, 64, 128 and 256 processes over 10 iterations. The communication pattern for Minisweep shows a symmetric communication pattern with each rank communicating with a maximum of 4 other ranks in the application. The number of messages also remains the same for all communicating pairs of ranks. Minisweep is a miniapp used to study the communication characteristics of a Sn radiation transport application and is therefore dominated by communication.

SW4lite SW4lite is a proxy application for SW4 [PS17], a code for solving seismic wave equations in Cartesian coordinates. We capture traces for SW4lite for 100 iterations and weakly scale the application for 32, 64, 128 and 256 processes. The communication pattern for SW4lite shows a symmetrical pattern with each rank communicating with 2-4 other ranks. The number of messages exchanged between each pair of communicating ranks is also the same.

3.4.3 Communication Behavior of Different Applications

We use Vampir to analyze the collected application traces to ascertain the amount of time each application spends in computation vs. the time it spends in MPI routines. This analysis provides a preliminary estimate of how much of the application runtime can be affected by either GPUDirect, or the different node configurations. We aggregate the time spent in all MPI communication functions to get the total MPI time of the application. Figure 3.5 shows the results from this analysis.

AmgX (Figure 3.5(a)) spends a relatively small amount of time in MPI routines ranging from 5.3% for 32 ranks to 7.2% for 256 ranks. This implies that even large improvements in MPI time will not significantly improve the overall performance of the application.

For Minisweep (Figure 3.5(c)), we observe that overall execution time is dominated by the MPI routines. MPI time ranges from 76.0% for 128 ranks to 81.7% for 64 ranks. Minisweep can greatly benefit from improvements in communication time as even small improvements in the messaging performance can significantly improve the overall performance of the application.

For applications such as LULESH (Figure 3.5(b)) and SW4lite (Figure 3.5(d)) we observe that a moderate amount of time is spent in MPI routines. LULESH spends between 28.1% (for 64 ranks) to 31.9% (for 125 ranks) of its execution time inside MPI routines. Similarly, SW4lite spends between 34.3% (for 64 ranks) and 42.5% (for 256 ranks) in MPI routines. The benefit accorded to the overall execution by reducing messaging time for these applications will largely depend on the quantum of improvement in messaging time. Slight variations in communication/computation ratio can be caused either by variations in the state of the system when the traces were collected or due to the communication characteristics of the application for different rank sizes.

This static analysis can be used to determine if it is worthwhile to invest in improving the MPI performance of the application either using GPUDirect or by changing the number of ranks on each node.

3.5 Results

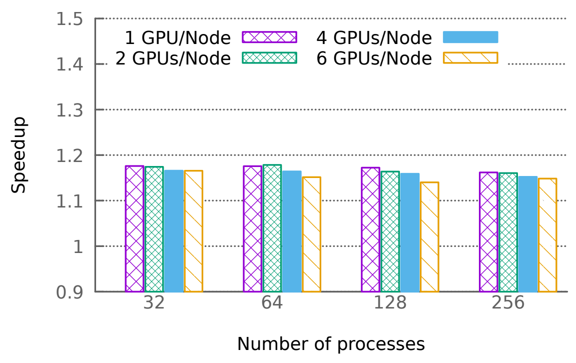
This section presents the results obtained from simulating executions via the traces for the node configurations described previously. For each application, we also simulate the effect of implementing GPUDirect. We assess the performance of GPUDirect and HostCopy by comparing the time each application spends in MPI routines (for GPUDirect) and MPI routines + CUDA memcpy calls (for HostCopy) for both the weak scaling and the strong scaling scenarios. Finally, we discuss the impact of the node configuration on each application by varying the number of ranks (GPUs) per node.

3.5.1 Weak Scaling Scenario

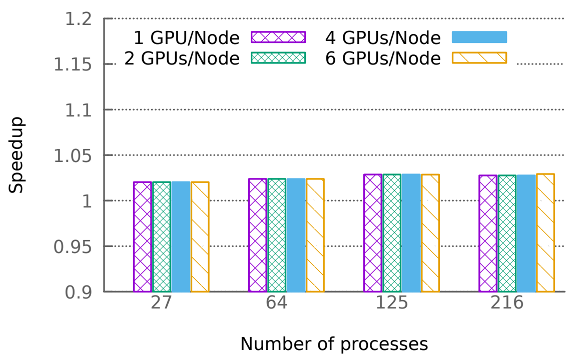
Figure 3.6 depicts the speedup in communication time achieved by GPUDirect over HostCopy for the four applications while weakly scaling.

Observation 1: *Some applications benefit more than others from GPUDirect communication. Lulesh has only up to a 3% improvement in communication performance while SW4lite shows up to a 75% improvement in communication performance with GPUDirect.*

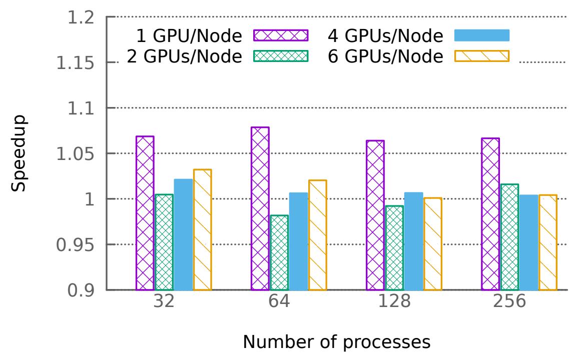
There is a substantial improvement in communication time for SW4lite (Figure 3.6(d)) when using GPUDirect in lieu of HostCopy, with a 15.8%-74.4% speedup of the former over the latter. For AmgX (Figure 3.6(a)), we observe that GPUDirect is faster by 16.2%-17.6%



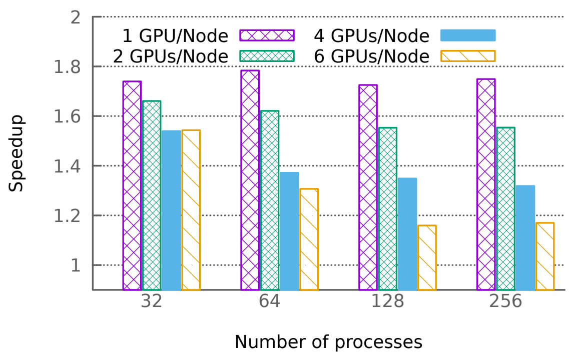
(a) AmgX



(b) Lulesh



(c) Minisweep



(d) SW4lite

Figure 3.6: Predictions of speedup achieved in communication time when using GPUDirect over HostCopy for different applications (in weak scaling mode).

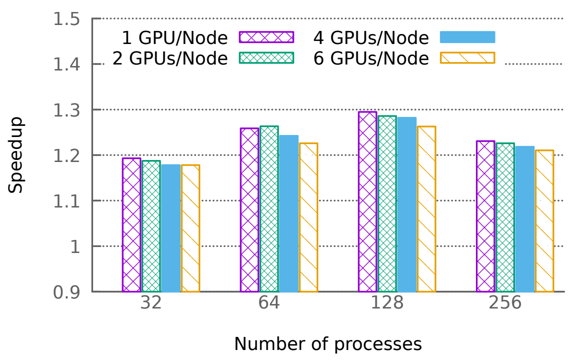
compared to the Host Copy. For Minisweep (Figure 3.6(c)), this difference ranges from 0.1% for 128 ranks to 7.9% for 64 ranks, and for LULESH (Figure 3.6(b)) the difference is small (less than 3%).

Observation 2: *The impact of reduced communication on overall application runtime under GPUDirect varies significantly depending on communication to computation ratio. Assessing both this ratio as well as the improvement due to GPUDirect helps in determining viable candidates for a GPUDirect implementation. Our experiments show that AmgX with only 5% communication is not a viable candidate while SW4lite with 40% communication is ideal.*

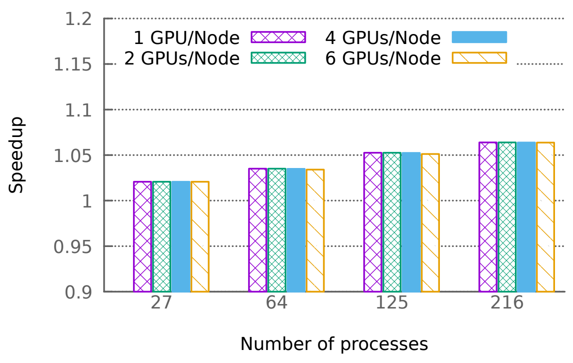
SW4lite spends up to about 42% of its execution time in MPI calls with up to a 74% improvement in communication performance when GPUDirect is used. Such applications can be expected to exhibit significantly improved execution times with GPUDirect. Minisweep (Figure 3.5(c)) spends up to about 81% of its execution time in communication. For such applications, the modest improvement under GPUDirect (up to 7.9%) can have a significant impact on the overall execution time of the application. In contrast, AmgX spends only about 5.3%-7.2% of its time in communication (Figure 3.5(a)). Even though this results in up to a 18% improvement in GPUDirect performance over HostCopy, the small time AmgX spends in communication limits the overall benefit in execution time under GPUDirect. Finally, LULESH spends a non-trivial amount of time in MPI (up to 32%), but small improvements in communication performance makes it unsuitable for retrofitting with GPUDirect.

Observation 3: *The improvement in GPUDirect performance for an application depends on the characteristics of the MPI messages. The size of the MPI messages as well as collective vs. point-to-point communication characteristics have an impact on GPUDirect performance.*

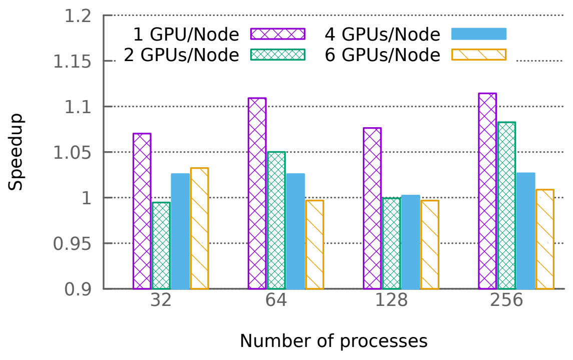
The benefit of using GPUDirect over HostCopy is most evident when MPI message sizes are large. In particular, Minisweep utilizes predominantly large MPI messages of either 4MB or 8MB. Similarly, SW4lite uses message sizes of about 64KB (50% of MPI messages) or about 12MB (the remaining 50% of messages). AmgX has a large distribution of message sizes: About 8% of messages are greater than 500KB, 17% have sizes of 100KB-500KB, and 71% range from 1KB-100KB. Experiments further indicate that AmgX benefits from GPUDirect collectives, dominated by *MPI_Allreduce*. For smaller message sizes, the performance improvement is modest. The size of MPI messages ranges between 24B to 6960B for LULESH accounting for more than 44% of the MPI messages, and an equal number of MPI messages are smaller than 512KB. Only about 11% of the MPI messages are around 1MB. This results in only a negligible improvement in GPUDirect performance.



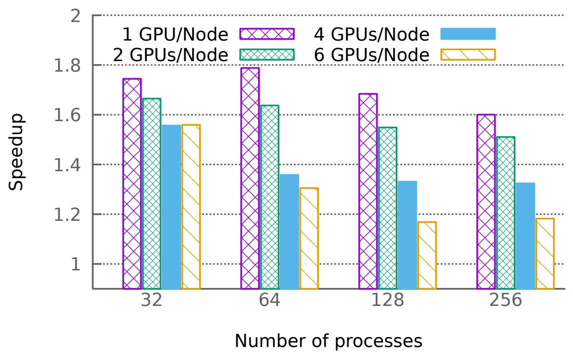
(a) AmgX



(b) Lulesh



(c) Minisweep



(d) SW4lite

Figure 3.7: Predictions of speedup achieved in communication time when using GPUDirect over HostCopy for different applications (in strong scaling mode).

While other factors also play a role, results show that message size has a significant impact on the application's ability to benefit from GPUDirect.

3.5.2 Strong Scaling Scenario

We conducted experiments using strong scaling traces collected for each application and rank size. Figure 3.7 depicts the results when strong scaling the applications from 32-256 ranks (27-216 ranks for LULESH). The figure shows the speedup in communication time achieved with GPUDirect (instead of the HostCopy) for each configuration. Results indicate improvements of GPUDirect similar to that in the weak scaling experiments across all applications. For AmgX (Figure 3.7 (a)), GPUDirect results in improvements of 18%-29% over HostCopy, and for SW4lite (Figure 3.7 (d)) improvements range from 18%-77%.

Marginal improvements are observed under GPUDirect for Minisweep (Figure 3.7 (c)) with only a 11% improvement over HostCopy in the best case, just as in weak scaling experiments. Similarly, experiments with LULESH (Figure 3.7 (b)) show that GPUDirect is only about 2%-6% faster than HostCopy during strong scaling.

3.5.3 Effect of Node Configuration

We investigated the effect of node configurations on the communication performance of applications. HPC centers procure machines with varying performance characteristics, the Sierra and Lassen supercomputers at LLNL have four GPUs per node while Summit at ORNL has 6 GPUs per node. In contrast, Piz Daint at CSCS has one GPU per node. Increasing the number of GPUs per node increases the peak performance of the machine while keeping the number of nodes constant, thus reducing the network and server costs. GPUs on different nodes communicate over the inter-node network, which is Infiniband for Lassen, Sierra and Summit. GPUs on the same node communicate over the intra-node networks, e.g., NVLink.

While Figures 3.6 and 3.7 show speedup numbers for communication, we also need to look at absolute time to understand the impact of the number of GPUs (figures omitted due to space constraints). The overall execution time can be split into computation and communication time. Because we are weakly scaling, the computation time remains about the same across different rank sizes. Increasing the number of GPUs (ranks) on a single node reduces the number of nodes required for an execution with the same number of ranks. Adding more GPUs opens up more opportunities for intra-node communication (over NVLink) since there are more ranks on the same node. In our simulations, we allocate each rank to its own GPU.

Observation 4: *Adding more GPUs (and subsequently more ranks) decreases performance of applications. Furthermore, larger application runs are impacted more than smaller ones. The communication performance of Lulesh is not impacted by adding more GPUs to the node while Minisweep experiences a communication degradation of up to 242% for 6 GPUs/node.*

The experiments show that while the high-speed intra-node network is used, the overall communication runtime of an application increases when more ranks are added to the same node. SW4lite and Minisweep see the largest performance penalty when more ranks are allocated to the same node. For SW4lite, the total communication time increases by 23.9%-39.8% for 32 ranks and up to 118%-227% for the 256 ranks when increasing the number of ranks from one per node to 6 per node whereas the overall execution time increases by 9.1% for 32 ranks to 50.4% for 256 ranks. This is because SW4lite spends only about 40% of its time on communication. Similarly, Minisweep suffers a performance penalty of 47.6%-52.6% for 32 ranks and up to 222%-242% for 256 ranks when going from one rank per node to six ranks per node. This also has a severe effect on the overall execution time since Minisweep spends a vast majority of its execution time on communication. The execution time increases by 36.9% for 32 ranks and up to 169.7% for 256 ranks. In case of AmgX the communication performance decrease is smaller, ranging from 2.4%-3.3% for 32 ranks and from 6.2%-7.4% for 256 ranks when comparing one rank per node to six ranks per node. Finally, the communication performance for LULESH remains roughly the same regardless of the number of ranks allocated to each node. For 27, 64 and 125 ranks, the communication performance remains the same, and for 216 ranks, an increase in communication time of only about a 1% is observed when modulating the number of ranks per node from one to six. The standard deviation in the communication time for different ranks is also quite low: 0.34-0.42 for AmgX, 0.09-0.11 for Lulesh, 0.01-0.02 for Minisweep, and 0.07-0.12 for SW4lite. The low standard deviation indicates that we do not introduce jitter when simulating collective communication. Furthermore, it also indicates that our collected traces do not experience high jitter; otherwise, the standard deviation would be significantly higher.

The higher communication cost observed while increasing the number of ranks on each node is due to rising contention for the shared NIC on the node. As the number of ranks per node increases, more ranks are engaging in MPI communication with ranks outside the node. This causes contention on the NIC shared within a node. While there is

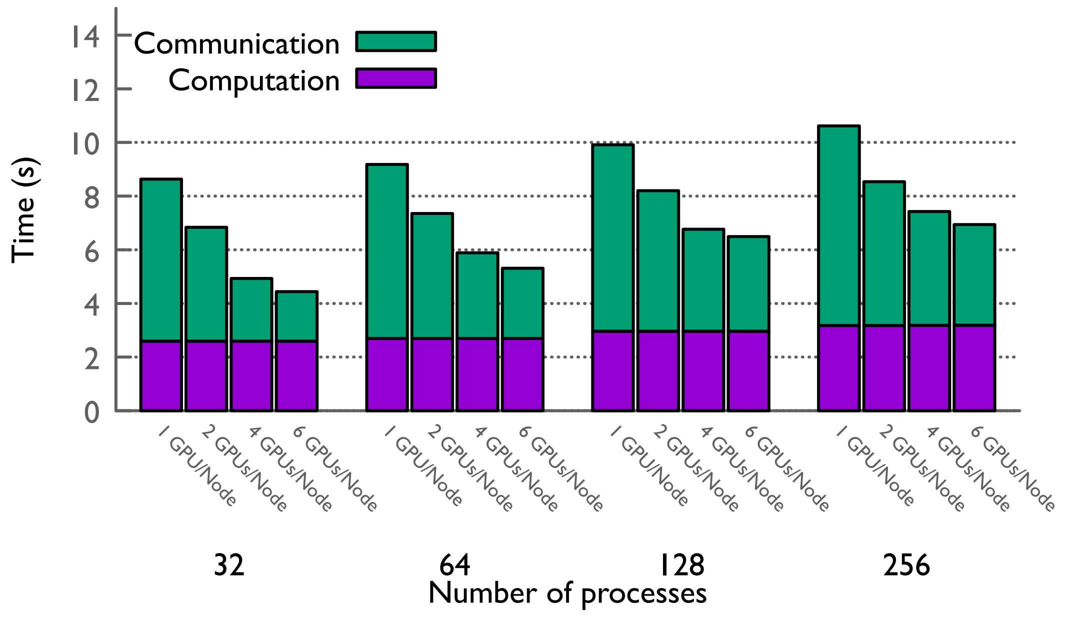


Figure 3.8: Minisweep communication/computation times for different node configurations while scaling the network bandwidth with number of GPUs/node.

sufficient bandwidth available on the intra-node interconnect, the increased requirement for inter-node communication causes a bottleneck at the node boundary.

Under strong scaling, the communication performance of applications degrades as more GPUs are added to the node, just as it did for weak scaling before. For AmgX, communication time increases modestly by 3.2% for 32 ranks and up to 10% for 64 ranks. As before, the increase in communication time for Lulesh is negligible at less than 1% across all configurations. In contrast, SW4lite and Minisweep have significantly reduced communication performance as more GPUs are added to the node with an increase in communication time of 39.5%-181% for SW4lite and and from 55.7%-225.8% for Minisweep.

Observation 5: *Adding more compute resources to a node requires careful consideration of the trade-offs, the expected application workload and optimized rank-to-node allocation schemes. Adding more GPUs to the node requires improvement in network interconnect technology if communication performance is to be maintained.*

HPC centers can increase the peak compute performance of the system by adding more GPUs to each node while keeping the server and networking costs in check. Consequently, operational expenditure can also be curtailed. This strategy is becoming common in HPC centers, i.e., they increase the number of GPUs on each node to four (Sierra at LLNL) or even six GPUs per node (Summit at ORNL). But interconnects have not kept pace with the increase in compute performance of modern CPUs and GPUs creating bottlenecks at the shared NIC. HPC centers should conduct studies of the trade-offs between cost and performance. We conducted simulation experiments with Minisweep by increasing the network bandwidth linearly with the number of GPUs on the node (shown in Figure 3.8). Our results show that the decrease in communication performance is avoided with interconnects having larger bandwidths. Here the intra-node NVLink interconnect starts to become the bottleneck. To this end, HPC centers need to evaluate the applications that dominate utilization on their system. Some applications (e.g., AmgX and LULESH) either show little or no performance penalty when more compute resources are added to the node. Others, however, like SW4lite and Minisweep, show significantly decreased performance. Furthermore, we observe across applications that choosing 1-2 GPUs per node has no significant effect on performance, but a further increase to four or six GPUs per node can severely degrade performance.

Finally, an application-conscious optimization of the allocation of ranks to nodes can increase the use of the intra-node interconnect if local communication dominates, which reduces the pressure on the shared NIC. Such an optimization requires prior analysis of the MPI messaging characteristics for each application. The proportion of intra-node communication can be increased by allocating more communicating ranks to the same node, particularly for small groups of frequently communicating ranks with less communication between such groups.

3.6 Related Work

Prior work has assessed the performance of different GPU interconnect topologies. Li et al. [LSC+19] evaluate the performance of interconnect topologies such as NVLink, PCIe, NV-SLI, NVSwitch and GPUDirect on 6 high-end servers and HPC platforms. They observe that intra-node multi-GPU communication is dependent on the choice of GPU combinations. Potluri et. al [PHV+13] study the performance of GPUDirect on PCIe-connected GPUs in a multi-node HPC environment. The authors show that using GPUDirect increases inter-GPU bandwidth significantly while decreasing communication latency.

Jiao et al. [JLBF10] characterize GPU applications based on performance and energy efficiency. They study applications with varying compute and memory intensity to conclude that performance and efficiency of GPUs in an HPC context depends on the computational patterns employed by the application. Pennycook et al. [PHJM11] utilize the NAS LU benchmark to compare existing HPC clusters and future large-scale systems. The authors use the LU performance model to extrapolate the performance of the benchmark to future large-scale distributed GPU clusters.

Choi et al. [CRKB20] model the end-to-end performance of GPU HPC applications using PDES while Arafa et al. [ABE$^+$21] focus on the prediction of computational performance of GPU applications. Finally, Chapuis et al. [CES16] use PDES to predict the GPU performance by using a model that is a combination of cycle-accurate GPU models and more coarse-grained analytical models. Their work focuses on the compute performance of individual GPUs and not on modern GPU communication paradigms or the resulting MPI performance. To the best of our knowledge, ours is the first work that contributes a PDES framework to comprehensively study the impact of multi-node GPU applications on HPC messaging performance with modern GPU communication paradigms.

3.7 Conclusion

This work enhances the TraceR simulation framework with novel capabilities to treat GPUs as a first-class computation device, enabling us for the first time to model multi-GPU nodes without actually having to port application code to GPUDirect. It complements TraceR with support for GPU-aware MPI communication, both for point-to-point messages and collectives in order to accurately simulate HPC systems with multi-GPU nodes. It also extends the trace collection utility, Score-P, with a feature to annotate MPI operations so as to indicate the use of GPUDirect without a need to refactor application code for GPUDirect. The framework gains the ability to support what-if analysis of modern communication paradigms such as GPUDirect. The performance of GPUDirect, in a what-if scenario, is then compared against the traditional method of copying data through host memory. These novel capabilities also allow us to study the impact of the number of GPUs per node on application and network performance. Using this framework, the communication performance of four important HPC applications is evaluated. Results indicate that applications experience asymmetric benefits from using GPUDirect, i.e., some will benefit significantly while others do not. Given our what-if analysis, this allows applications programmers to selectively decide which applications to refactor for GPUDirect. Furthermore, this work points out that HPC centers are facing an important tradeoff between maximized performance vs. minimized capital and operational expenditure, i.e., a choice between adding more GPUs to fewer nodes vs. deploying more nodes with fewer GPUs — a decision that also depends on the application mix, as our results show. Finally, the simulator can be combined with a performance model like a roofline or analytical model to further improve prediction performance.

CHAPTER

4 HETEROGENEOUS WORK SHARING AND CO-SCHEDULING ON CPUS, GPUS AND FPGAS

4.1 Introduction

Over the last decade significant changes have impacted HPC hardware. The introduction of accelerators, such as GPUs, FPGAs and DSPs, have added considerable computational capabilities to HPC systems, albeit at the cost of programming complexity. Furthermore, recent HPC systems such as Summit [Lab21c] and Sierra [Lab21b] as well as next generation systems such as Frontier [ORN] and Perlmutter [NER] are adopting fatter nodes with multiple CPUs and GPUs in favor of many more thinner nodes to achieve the desired peak performance. This trend is expected to accelerate further [VBG+19] with more and diverse accelerators becoming a mainstay in HPC systems. While current applications need to be updated to efficiently leverage this diverse set of computational hardware, there is a need to effectively manage the available hardware to achieve ideal utilization.

Current production applications are designed to target specific hardware. Applications target GPUs using frameworks such as CUDA [Nvi07] or ROCm, CPUs with OpenMP, and FPGAs with OpenCL (HLS) or VHDL. OpenCL provides a common interface for developing applications for a diverse set of computational hardware but applications running on a specific device leave other computational hardware on the node remaining idle. While there has been some prior work on sharing work amongst heterogeneous computing devices, such studies either require significant changes to the application or do not support a wide variety of devices.

Furthermore, scheduling multiple applications on a single node has raised significant interest in the HPC community. Co-scheduling applications allows HPC systems to use the available HPC resources more effectively. Current approaches either target specific devices for co-scheduling while reducing interference (e.g., by mitigating shared resources when multiple applications use the same CPU) or avoiding interference by having co-scheduled applications utilize disjoint accelerating devices (e.g., one using GPU and another using FPGAs) on a heterogeneous HPC node.

Our work aims to leverage both work sharing and co-scheduling in a common framework to more holistically use the heterogeneous nature of current and future HPC systems. We split each OpenCL computational kernel into a "bag of tasks" with each task (or slice) scheduled on different devices. This enables our work-sharing framework to achieve higher performance than running on any single device. Furthermore, we use this framework to seamlessly migrate, expand, or contract our application between devices. This capability creates a co-scheduling framework to enable multiple applications to run on a single node. The key contributions of this work are as follows:

1. We create a work-sharing framework that allows OpenCL kernels to be scheduled on multiple devices without requiring the application developer to make significant changes in the application.

2. We provide the capability to seamlessly migrate kernels from one accelerator to another, expand kernels to use more accelerators, or contract to use fewer accelerators.

3. We augment the work-sharing framework with co-scheduling capabilities by providing a framework with pluggable scheduling algorithms and the ability to optimize for job throughput, job priority, or a hybrid of multiple objectives.

4. Finally, we evaluate our work-sharing and co-scheduling framework under different scheduling algorithms. We show that applications benefit from a different combina-

tion of accelerators when using work sharing as opposed to running on any single device. We further show that workload characteristics such as inter-arrival times and the application mix can impact the suitability of different co-scheduling algorithms.

We evaluate our work sharing and co-scheduling framework using a single node on a mid-sized HPC cluster comprising of a CPU, a GPU and an FPGA. We use four applications to evaluate our work-sharing framework with different combinations of devices. We also create four workloads comprised of different sizes of the applications and evaluate our co-scheduling approach under different scheduling algorithms. We will make our framework and its code base available as open source to the community opening up novel research and community contributions based on our work.

We evaluate applications and workloads optimizing for job throughput while prioritizing jobs by arrival time but scheduling algorithms can be developed prioritizing other factors. Furthermore, we evaluate our framework with a single node, with the potential to expand to co-scheduling for multi-node applications using MPI in the future. Finally, our framework is best suited for applications where data output by a kernel is not used by subsequent calls to the kernel. This, too, can be tackled in future work by orchestrating data movement for the output of each slice upon slice completion.

4.2 Background

This section provides a brief overview of accelerators used in modern HPC systems — GPUs and FPGAs. It also provides a fundamental background on OpenCL and High Level Synthesis (HLS).

4.2.1 GPUs in HPC

General Purpose Graphics Processing Units (GPGPUs) have become increasingly popular and have been driving scientific computation with significant improvements in performance and power efficiency [OHL+08, KES+09]. The development of frameworks and languages such as NVIDIA's Compute Unified Device Architecture (CUDA [Nvi07]), the Open Compute Language (OpenCL) [SGS10] and, more recently, Intel's OneAPI [Int] have contributed toward making GPUs first class computation devices in modern HPC systems with recently commissioned systems such as Summit [Lab21c] and Sierra [Lab21b] relying primarily on GPUs for their peak floating point operations per second (Flop/s).

4.2.2 FPGAs in HPC

Field Programmable Gate Array (FPGA) devices as HPC accelerators have gained traction in recent times [DRP11, VB13]. FPGAs consist of configurable logic blocks (CLBs), digital signal processing (DSP) blocks, I/O blocks, a block RAM, a digital clock manager (DCM) module and a programmable interconnect to connect these blocks. Figure 4.1 shows the block diagram of an FPGA and its components. The CLBs consist of look up tables (LUTs), which are used to program the application logic into the FPGA hardware. FPGAs are often connected to the system as PCIe devices and while current generation FPGAs might lag behind other accelerators, they have been shown to be more energy efficient for certain applications [BTL10, NMS+21]. Furthermore, high level synthesis (HLS) has made the process of programming for FPGAs much more accessible leading to an increase in popularity.

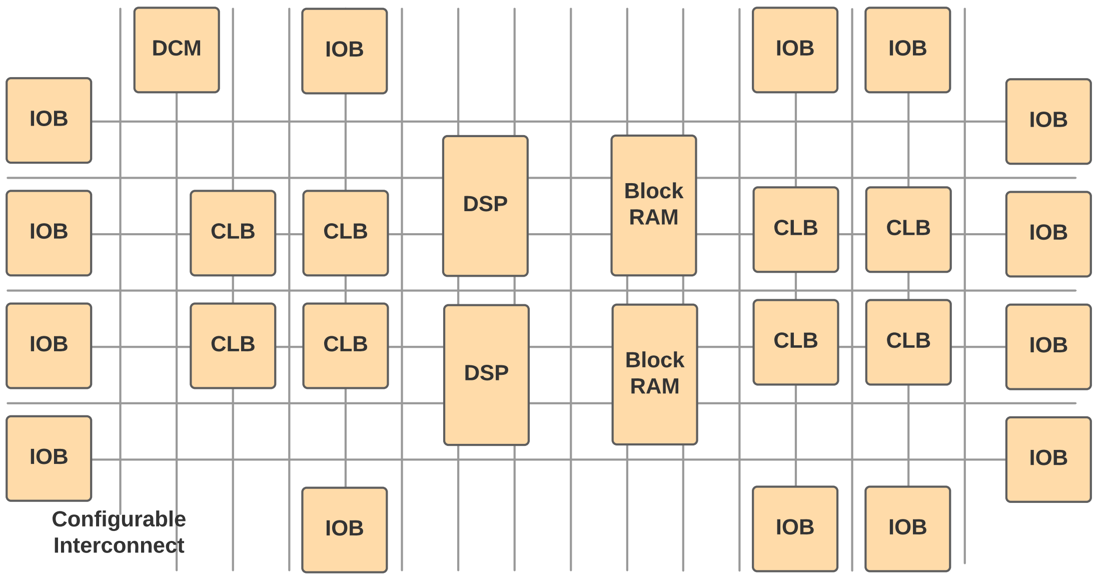


Figure 4.1: Block diagram of an FPGA with its components

4.2.3 OpenCL

Unlike CUDA, which is an NVIDIA proprietary framework for NVIDIA's GPUs, OpenCL provides an open standard for programming parallel applications for a wide range of accelerators, such as multi-core CPUs, GPUs, FPGAs, DSPs, Tensor cores, etc. It provides a standard interface for task-based and single instruction multiple data (SIMD) parallelism that the application developer can leverage based on the target architecture and application. Like CUDA, the application is divided into the host code and the device (or kernel) code. The host code is responsible for initializing the target device(s), allocating memory on the device, coordinating the movement of data between the host DRAM and the device memory, and enqueuing the kernel on the device. The kernel code describes the computation that is to take place on the accelerator.

Figure 4.2 shows the flow of an OpenCL application. The host code is compiled using a C/C++ compiler and linked to the OpenCL library to generate the host binary. The kernel code (shown in the figure as device.cl) can either be pre-compiled for a specific accelerator, or the host code can compile the device code at run-time for any target accelerator (with some exceptions to be discussed later). The host binary is then executed on the CPU, it initializes the target accelerator, co-ordinates data movement, compiles the device code and finally enqueue the kernel on the device before it is run on the FPGA.

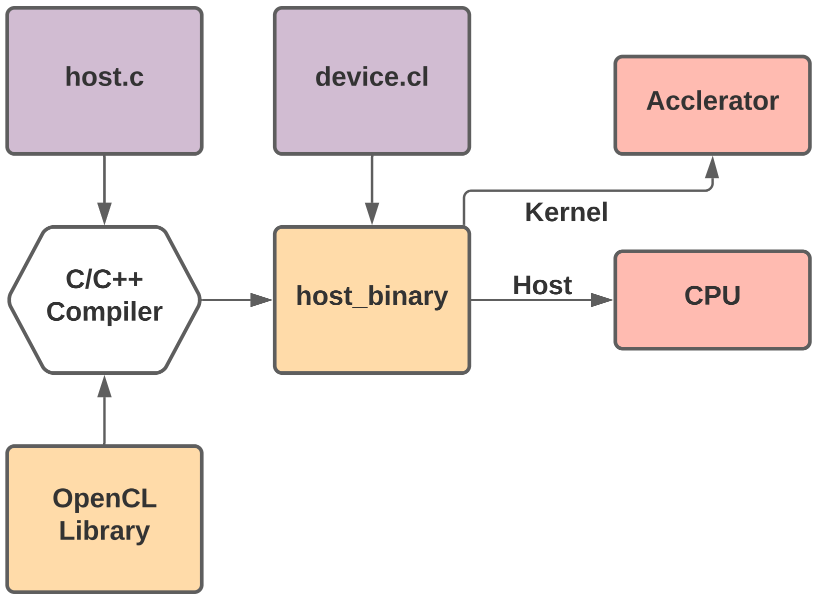


Figure 4.2: OpenCL application flow

4.2.4 High Level Synthesis

Traditionally FPGAs have been dependent on Register Transfer Level (RTL) code to describe the logic that needs to be programmed into the accelerator. High level synthesis (HLS) allows application developers to write the accelerator logic in a high-level language such as OpenCL and compile that to its equivalent RTL code that can be programmed into the FPGA. This allows for abstraction of the core application logic from the underlying hardware description. HLS tools not only enable compilation of high-level code to RTL but also provide tools to ensure correctness.

Figure 4.3 shows the development workflow for FPGAs when using HLS. While a full compile for the FPGA can take several hours, application developers can ensure correctness using emulation and make changes to the code if necessary. Furthermore, profiling tools can be used to ensure the energy and performance requirements are met without having to embark on a full scale compilation. Finally, once correctness is ensured and performance and energy constraints have been met, the application developers can run the application on the physical hardware.

4.3 Design and Implementation

4.3.1 Kernel Slicing

This section describes the design and implementation of our work-sharing and co-scheduling framework. It begins by describing our technique of slicing of OpenCL kernels that enables work-sharing and supports co-scheduling within our framework.

OpenCL kernels can either be programmed as a single task or an NDRange kernel. In a single-task kernel, SIMD parallelism is not leveraged and there is only one thread of execution. This is not suitable for accelerators such as GPUs or even multi-core CPUs, bit it lends well to FPGAs, which utilize pipelined parallelism. An NDRange kernel describes the computation as work items and utilizes SIMD parallelism. In this work, we exclusively study NDRange kernels, which can be programmed to be suitable for all three types of accelerators, CPUs, GPUs and FPGAs.

In an NDRange kernel, the computation is divided into work items, where each work item has its own logical thread of execution. The work items are grouped into a local work group with the ability for threads to synchronize only within the local work group. The overall computation is referred to by the global work group. Along with the local and global

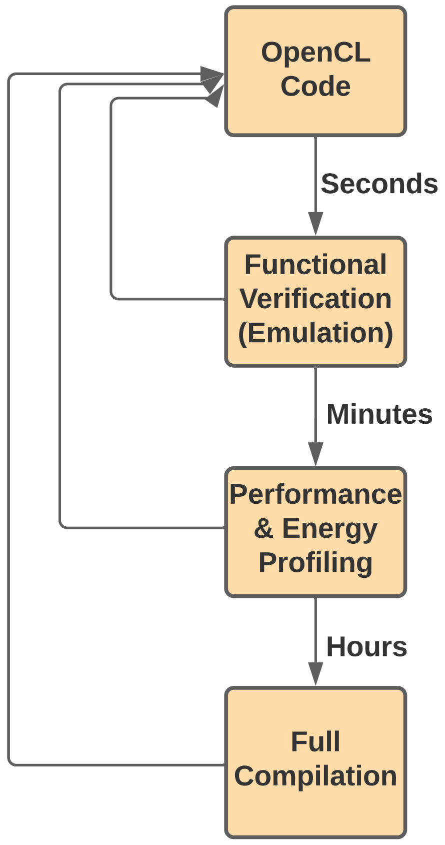

OpenCL Code
Seconds
Functional Verification (Emulation)
Minutes
Performance & Energy Profiling
Hours
Full Compilation

Figure 4.3: HLS development workflow

work groups, we add the notion of a *slice*, which is a multiple of the local work group size. Figure 4.4 shows an NDRange kernel for a two-dimensional (2D) computation with a single work item shown in green. Since synchronization is possible only within a local work group, each slice can be computed independently.

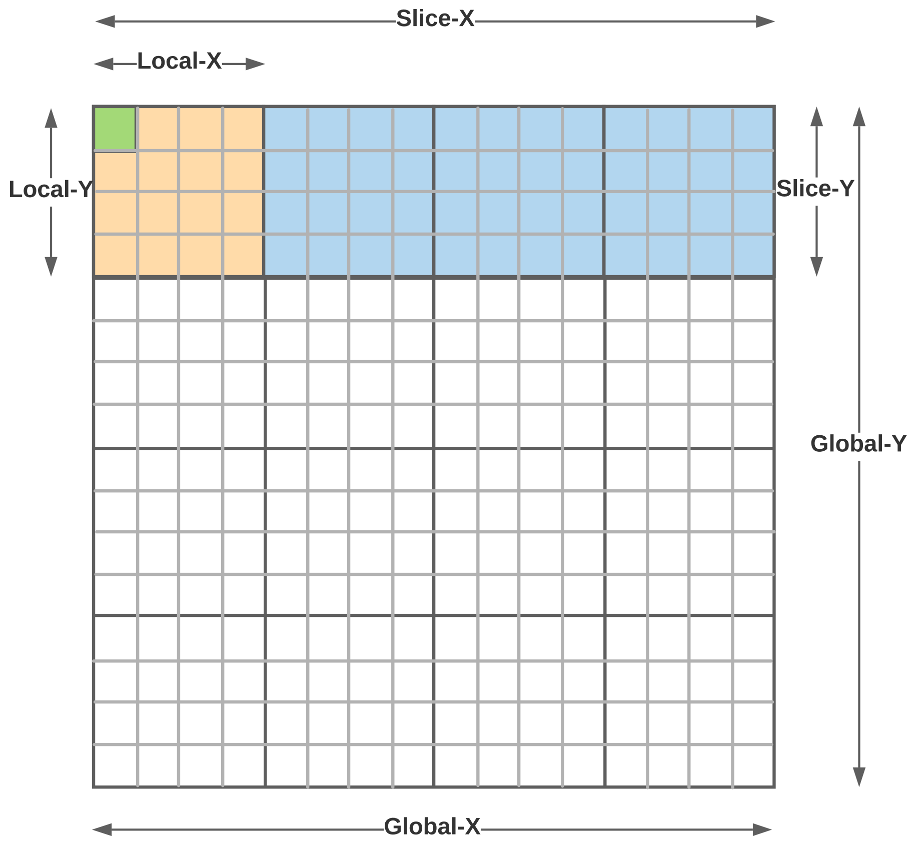


Figure 4.4: A 2-D NDRange kernel

We leverage this slicing mechanism to transform a single kernel into a series of kernels (equal to the number of slices) with each slice being computed one after the other. Figure 4.5 shows the overhead of slicing with GEMM with different number of slices. For a small input size (4096x4096), we observe that a large number of slices (256) degrades performance by about 5.6% but for larger input sizes the performance is indistinguishable from a simple single kernel execution.

Note: For the slicing mechanism to produce correct output, we changed the kernel to take an extra parameter that specifies the current slice. This enables to kernel to write

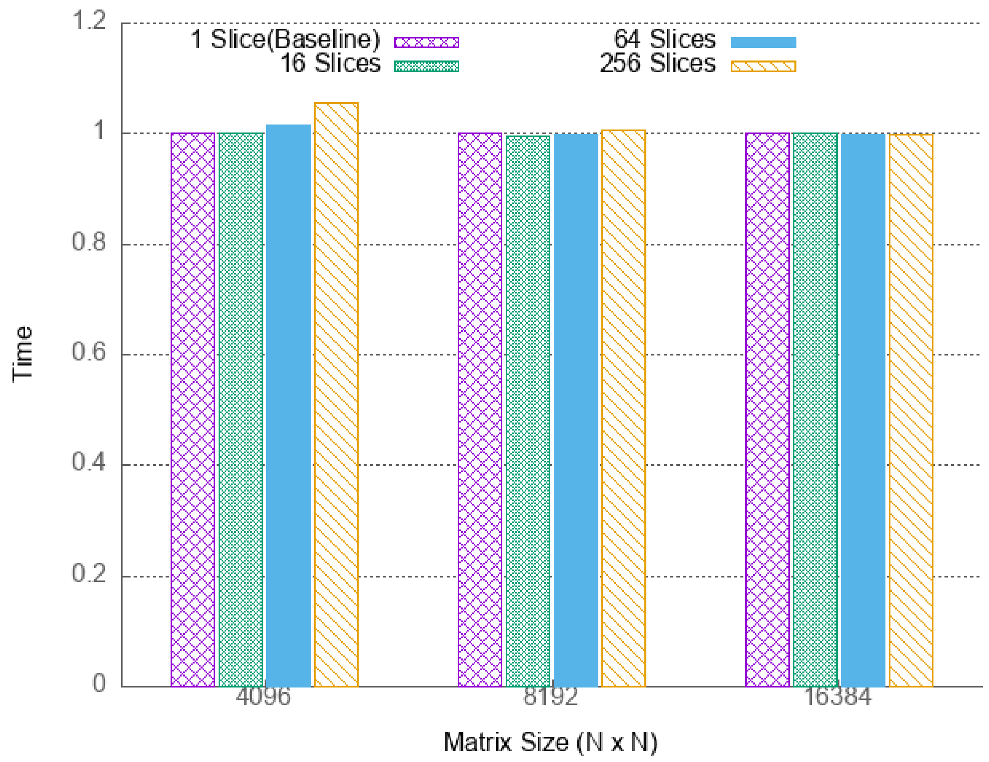


Figure 4.5: Overhead of slicing with GEMM

the result of the computation to the correct location in the output buffer(s). This can be avoided by using the clEnqueueNDRangeKernel function (used to enqueue a kernel on an accelerator), which accepts a parameter that can be used to specify an offset for the global work id. However, one of the versions of OpenCL on our system has not implemented this functionality, which forced us to resemble it by code refactoring [Khr].

4.3.2 Work Sharing

With the ability to organize a single OpenCL kernel into several slices that can be scheduled as independent kernel executions, we create a fine-grained work sharing framework. Figure 4.6 shows the combined design for our work sharing and co-scheduling framework. Memory for the input data structures as well as the output data structures are allocated on each of the accelerators participating in the work sharing for the job. Input data structures are copied to each of the accelerators (if necessary), then independent threads for each accelerator atomically picks slices from the bag to execute as a kernel. Once all the slices have completed, output buffers are copied from the accelerators. Finally, each of the accelerators' clean-up routine is executed to release the memory and device(s).

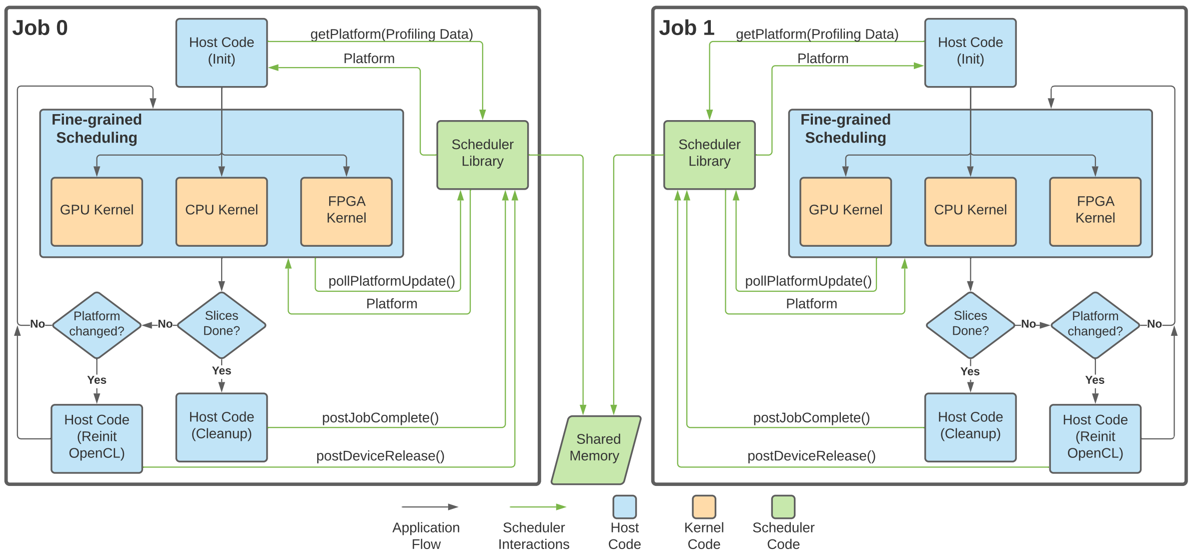


Figure 4.6: Co-Scheduler and Work Sharing Framework Design

This fine-grained scheduling approach ensures that the job is held up for a shorter time waiting for a single accelerator to complete execution for the last remaining slice. As seen in Figure 4.5, such fine-grained execution does not add significant overhead as only a small overhead is imposed by work sharing. Multiple copies of each data structure need to be allocated for each of the accelerators and threads, and each accelerator needs to be spawned. Furthermore, the data structures need to be copied to and from the device memory for each accelerator that participates in the work sharing adding additional overhead, just as for GPUs. Finally, for a slow device executing a particular kernel, the last slice allocated to this device might only complete execution significantly later than when the other accelerators have completed execution. This can be mitigated by increasing the number of slices (and reducing their respective size) for the kernel, which may, conversely, hamper performance on the faster devices.

To maximize performance, the above mentioned trade-off needs to be carefully assessed for each application. In this work, we show a common OpenCL kernel being executed on CPUs, GPUs and FPGAs. In general, the work scheduling framework can be used to combine multiple programming paradigms such as CUDA for GPUs or OpenMP for CPUs. Further, our system comprises of a CPU, a GPU and an FPGA, but the work sharing framework can be used for multi-GPU systems (e.g., Petascale systems such as Summit and Sierra) or even systems where GPUs with differing capabilities are present on the same node or within a cluster.

4.3.3 Co-Scheduling

The details of our co-scheduling framework are given in Figure 4.6, which shows the design of our framework with the scheduler and its interactions. Each job links to the scheduler library during compilation, and each instance of the scheduler library uses a shared block of memory to store all scheduling information. Scheduling decisions are taken whenever a new job arrives, a job switches devices, or a job completes execution. All accesses to the scheduler's critical data structure is protected using a semaphore to ensure that multiple instances of the scheduler read coherent data. While a single job can be assigned multiple accelerators, an accelerator can only be assigned to a single job.

The interactions between a job and the scheduler are as follows:

1. Before initialization of the application, the application requests an available platform using **getPlatformId**. It sends the scheduler profiled information about the application's relative performance on different combinations of devices. The scheduler either returns an available platform to the application (CPU, GPU, FPGA, CPU/GPU, etc.) or waits for a platform to become available.

2. Once the **getPlatformId** function returns indicating the assigned platform, the job starts execution as described in the previous subsection. After completing each slice, it polls the scheduler to check if a different platform has been assigned using **pollPlatformUpdate**.

3. If there is no change in the platform, the job continues with the next slice, otherwise the job waits for all of the currently executing slices to complete. Once all current slices are complete, the application executes a clean-up of the held devices before finally informing the scheduler using **postDeviceRelease**. The application can then initialize the newly assigned devices and continue execution from the next available slice.

4. Once all the slices are done, the job informs the scheduler using **postJobComplete** that it may terminate.

Switching accelerator platforms incurs a significant overhead on the application since partial output buffers need to be copied out from the current devices, the current devices need to be released, the newly assigned devices need to be initialized and input data needs to be copied to them. It is possible to mitigate some of this overhead. For example, if the scheduler assigns the GPU+FPGA to an application currently using the GPU, we do not need to release the GPU. But we run into an issue with the FPGA: If the FPGA library is loaded within an application, any use of the OpenCL library in that application would cause any other application using the FPGA to freeze due to an internal runtime lock beyond our control. Hence, we load the FPGA library *conditionally* if and only if the platform returned by the scheduler includes the FPGA using **dlsym**.

While this solves our initial issue, it creates another one. An application using the GPU cannot switch to the FPGA since loading the FPGA library using **dlsym** will not properly link it to the OpenCL library, which was loaded when execution started on the GPU. Fixing this issue requires us to unload OpenCL and reload both OpenCL and the FPGA library for the application to be able to use the FPGA. Therefore, we have to release all devices and reinitialize them whenever a change in platform occurs. Notice that a fix for this issue with the FPGA library could significantly decrease platform switching overhead, but this is beyond the scope of this work due to partially proprietary software stacks that we do not have sources of.

We implement and evaluate *four scheduling algorithms* for co-scheduling:

Baseline

This scheduling algorithm runs applications only on their most preferred device. While applications can be co-scheduled on the system, they will wait for their primary device to become available before being scheduled and dispatched. Priorities are assigned to the jobs based on their arrival times. A lower priority job can be scheduled before a higher priority job only if both are scheduled on different devices.

Greedy + Up Migration

This algorithm can start application execution on any device that is available. Once a more preferable device becomes available, jobs are switched to run on that new device. Priorities are still assigned based on arrival time, but in this algorithm lower priority jobs cannot run before a higher priority job since any job can run on any device. Furthermore, preference for switching to a newly released device is given to the higher priority jobs.

Elastic

This algorithm leverages our work sharing framework in conjunction with co-scheduling. Priorities to jobs are still assigned by arrival time, but this algorithm can schedule higher priority jobs on multiple devices using work sharing. Like the previous algorithm, jobs can start on any device. As other devices become available, jobs expand to share work amongst multiple devices.

Elastic — Device Limiting

Finally, we implement Elastic-DL by modifying the Elastic algorithm. Elastic-DL is similar to Elastic with one key difference. Here, we limit the maximum number of devices that a single job can be allocated to. In our experiments, we set the limit to two devices since we have a total of three accelerators on our test system.

4.4 Experimental Framework

This section describes the system, applications and workloads used to evaluate the work sharing and co-scheduling framework.

4.4.1 Applications

We use four applications for the evaluation of our work sharing framework, which together comprise workloads to evaluate the co-scheduling algorithms.

Mandelbrot

Calculation of the Mandelbrot set is an important application particularly in the field of encryption and security. Several methods have been proposed to speed up parallel computation of them using HPC [DSG14, GB20]. We extend the implementation of the Mandelbrot set provided with the Intel FPGA library to support work sharing. We use the application to generate a certain number of frames of the set with a certain number of colors and vary the size of each frame.

GEMM

General Matrix Multiplication (GEMM) is a critical kernel for HPC systems with a wide variety of applications requiring GEMM for computation. From AI workloads to physics simulations, all rely on GEMM. For GEMM, we modify the implementation provided with the OpenCL library to support our framework and evaluate its performance for a range of different square matrix sizes.

SpMV

Sparse Matrix-Vector Multiplication (SpMV) is another linear algebra kernel extensively used in scientific and engineering applications and processing of large data sets. It multiplies a sparse matrix with a dense vector to produce a vector. We implement an SpMV application using the compressed sparse row (CSR) format as a sparse matrix representation. We use a constant number of non-zero elements distributed evenly across the rows for different matrix sizes during evaluations.

XSBench

XSBench is mini-app representing the computation for a Monte Carlo neutron transport algorithm [TSIS14]. We modify an OpenCL implementation of the application provided by Argonne National Laboratory (ANL) to support work sharing and co-scheduling. Our evaluation uses the event-based simulation with 340k gridpoints and a nuclide grid search. We vary the number of look-ups to generate different application sizes.

4.4.2 Workloads

We have designed and implemented a workload generator to create randomized workloads using the above applications with different input sizes. Besides varying inputs, it takes the maximum inter-arrival time between jobs as a parameter. We created a total of 15 jobs shown in Table 4.1.

Table 4.1: Applications and Input Sizes

Application	S	M	L	XL
Mandelbrot (Frame Size NxN)	2048	4096	8192	16384
GEMM (Matrix Size NxN)	4096	8192	16384	22400
SpMV (Matrix Size 2^Nx2^N)	28	29	30	31
XSBench (Lookups)	16M	32M	64M	-

Based on these jobs, we create three randomly generated workloads consisting of all four applications with one instance of each job by varying the maximum inter-arrival time (IAT) parameter. We also create another workload with just two of the four applications, namely GEMM and Mandelbrot. The details of each of the workloads are shown in Table 4.2.

Table 4.2: Workload Details

Workload	Applications	Jobs	Avg. IAT
Workload 1	Mandelbrot GEMM SpMV XSBench	15	2.3s
Workload 2	Mandelbrot GEMM SpMV XSBench	15	10.5s
Workload 3	Mandelbrot GEMM SpMV XSBench	15	23.9s
Workload 4	Mandelbrot GEMM	12	2.2s

These applications and their parameterization provide a diverse mix of different workload characteristics. Each of these workloads are evaluated for the co-scheduling algorithms described in the previous section.

4.4.3 System Details

We run both our work sharing and co-scheduling experiments on a single node on a mid-tier HPC cluster. The system consists of an Intel multi-core CPU, an NVIDIA GPU and an Intel Altera FPGA.

The CPU is a 16 core Intel Xeon running at 2.10GHz capable of up to 32 threads (with Hyper-Threading). The GPU on the system is an NVIDIA RTX 2060 with 6GB of device memory and the FPGA is an Altera Arria 10 with 8GB of device memory. The system has 64GB of DRAM.

On the software side, the system is running CentOS with the 4.10.13 version of a back-patched Linux kernel. Each of the three devices utilizes a different versions of OpenCL: The CPU uses OpenCL 2.1 provided by Intel, the GPU exploits OpenCL 1.2 provided by NVIDIA, and the FPGA relies on OpenCL 1.0 also provided by Intel.

4.5 Results

This section present the results of our work sharing and co-scheduling framework for the applications and their workloads discussed in the previous section. First, we assess results for work sharing, followed by a discussion on how we use those results to achieve efficient co-scheduling of these applications.

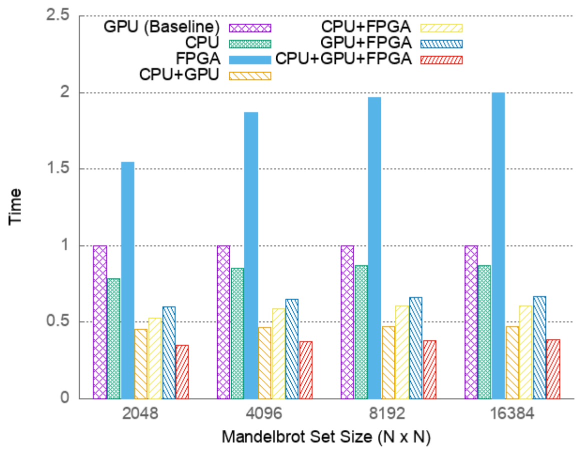
4.5.1 Work Sharing

Figure 4.7 depicts on the y-axis the performance in time (normalized to the GPU baseline, lower is better) of each application for different input sizes on the x-axis with work being shared amongst different combinations of devices indicated by different bars in the legend. For XSBench and SpMV, we omit certain FPGA combinations from the results. This is due to the poor performance of the FPGA for these applications. Due to the same reason, we also omit certain CPU combinations for GEMM.

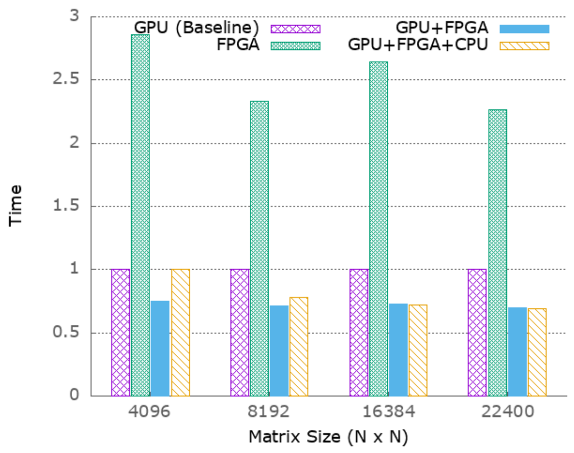
These results comprise the time spent in computation including copying to and from the device but neither include the input initialization nor the OpenCL initialization times.

Mandelbrot

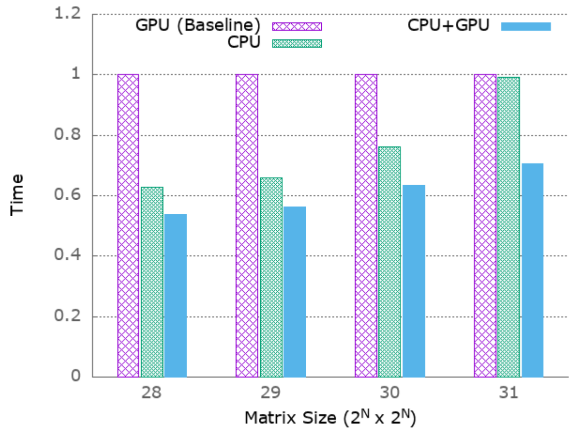
For Mandelbrot (Figure 4.7(a)), each of the devices have somewhat comparable performance. This results in efficient work sharing between any combination of devices. While the CPU is the best performing of the three devices, sharing work between any combination of devices yields better performance. For instance, combining GPU and FPGA (slower individual devices) yields a speed-up of 1.308x over running only on the CPU (fastest individual devices). Similarly, running on all three devices simultaneously results in a speed-up of 2.26x over running on the CPU alone.



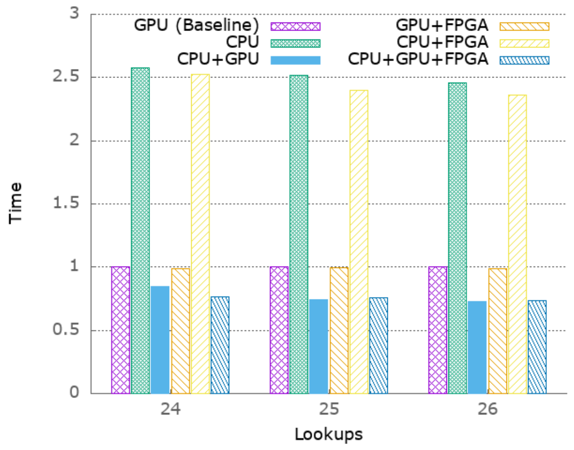
(a) Mandelbrot



(b) GEMM



(c) SpMV



(d) XSBench

Figure 4.7: Normalized performance over various input sizes for various devices of each application.

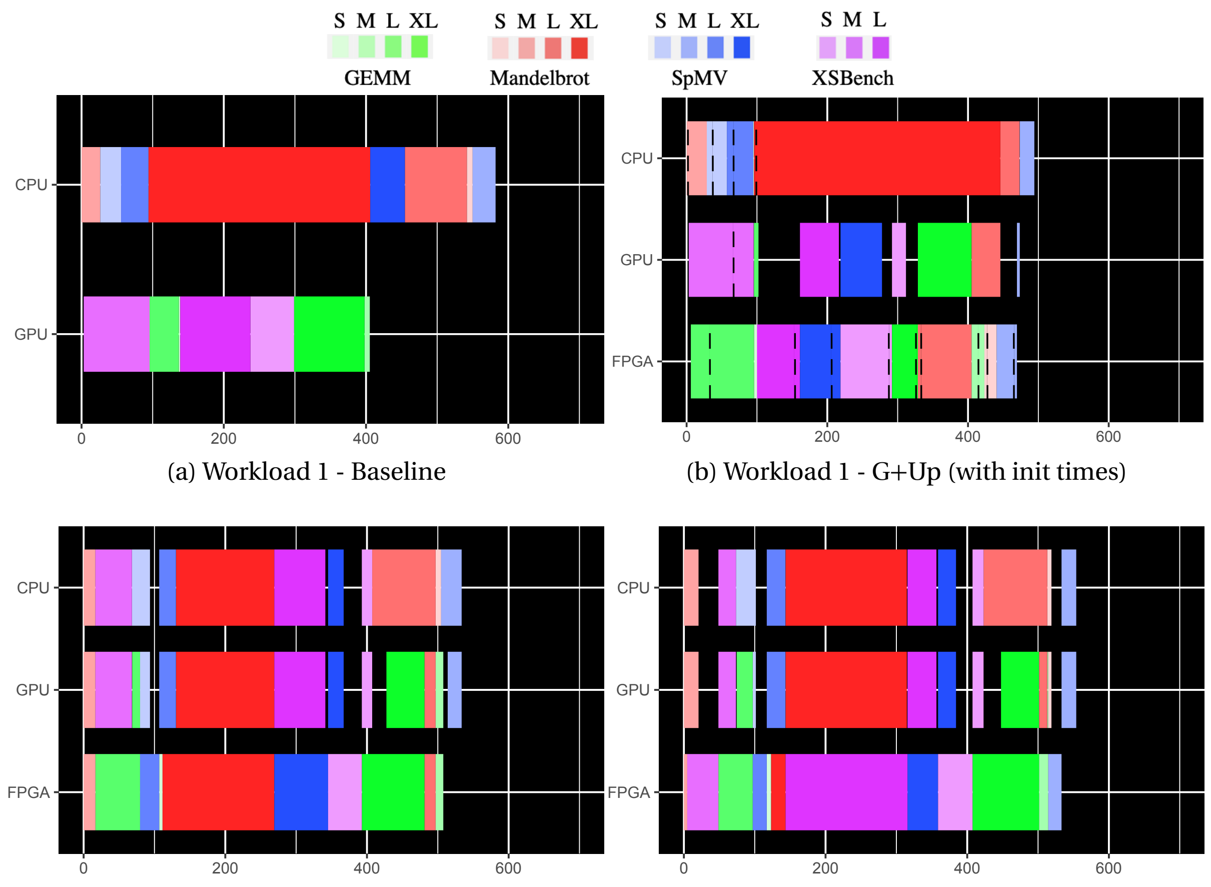
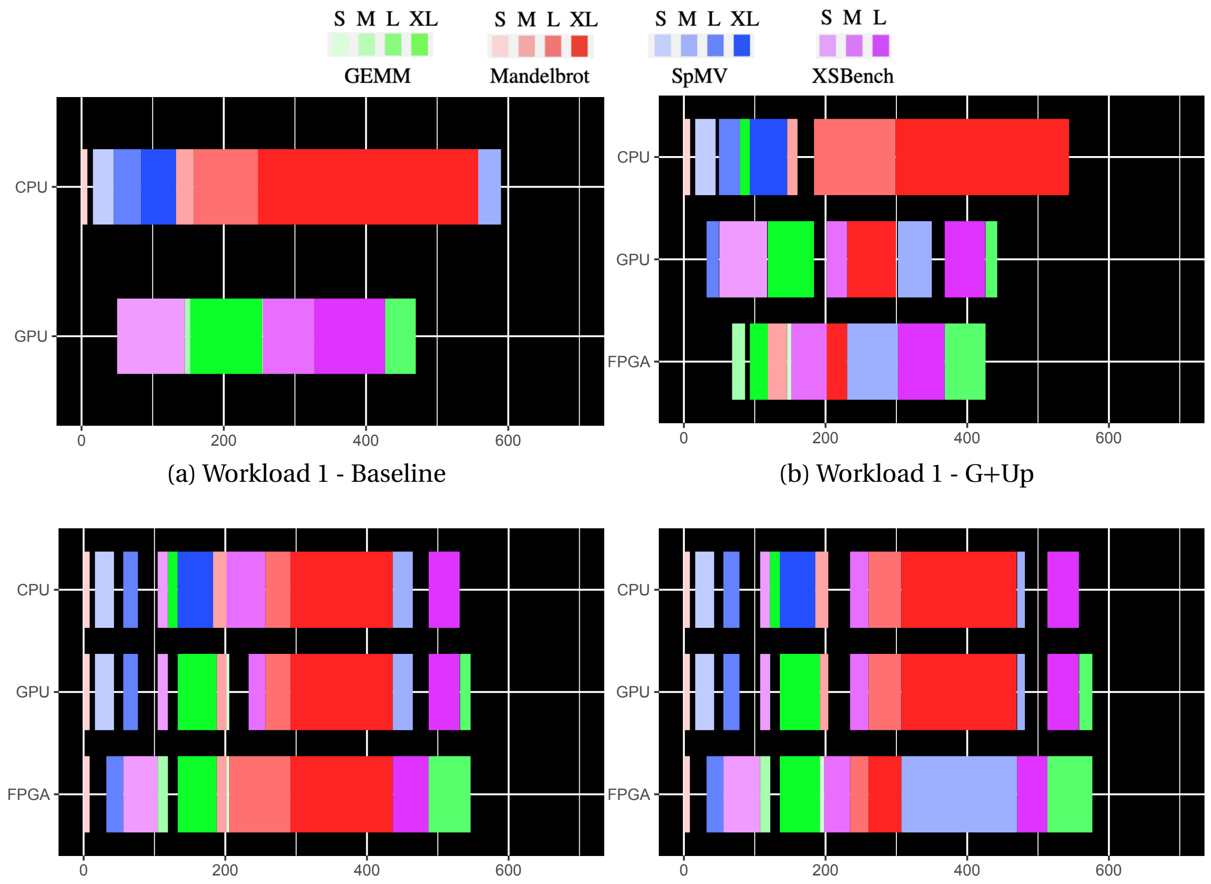


Figure 4.8: Timeline for the jobs on each accelerator for workload 1 for each of the scheduling algorithms



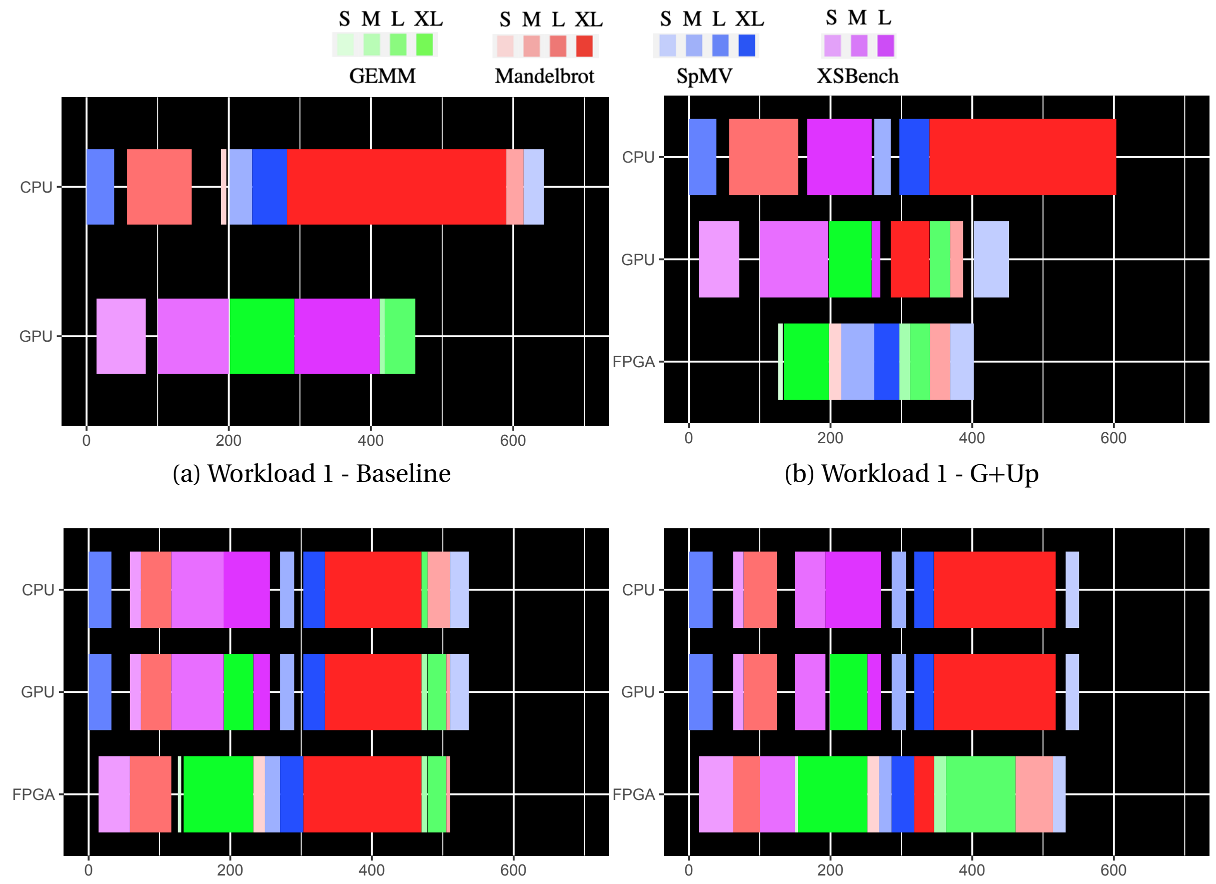
(a) Workload 1 - Baseline

(b) Workload 1 - G+Up

(c) Workload 1 - Elastic

(d) Workload 1 - Elastic-DL

Figure 4.9: Timeline for the jobs on each accelerator for workload 2 for each of the scheduling algorithms



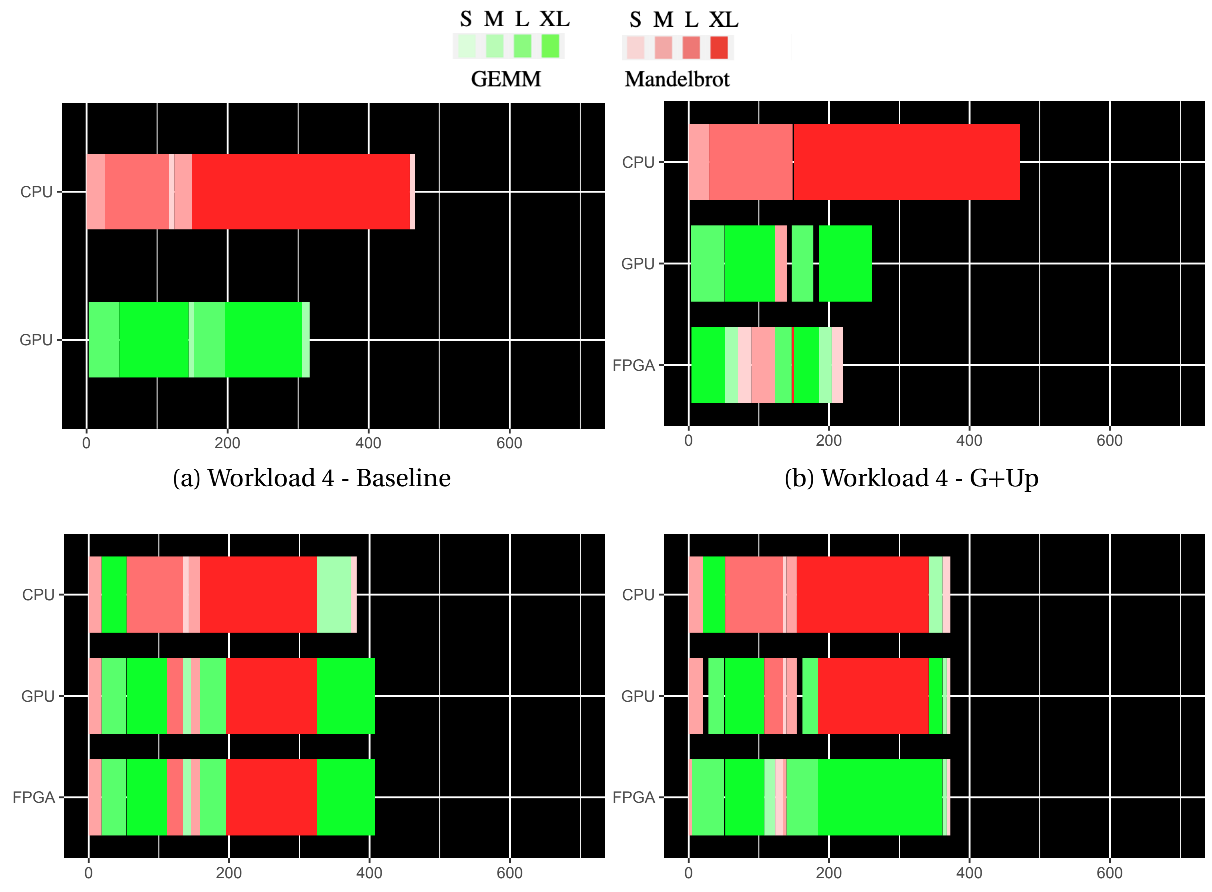
(a) Workload 1 - Baseline

(b) Workload 1 - G+Up

(c) Workload 1 - Elastic

(d) Workload 1 - Elastic-DL

Figure 4.10: Timeline for the jobs on each accelerator for workload 3 for each of the scheduling algorithms



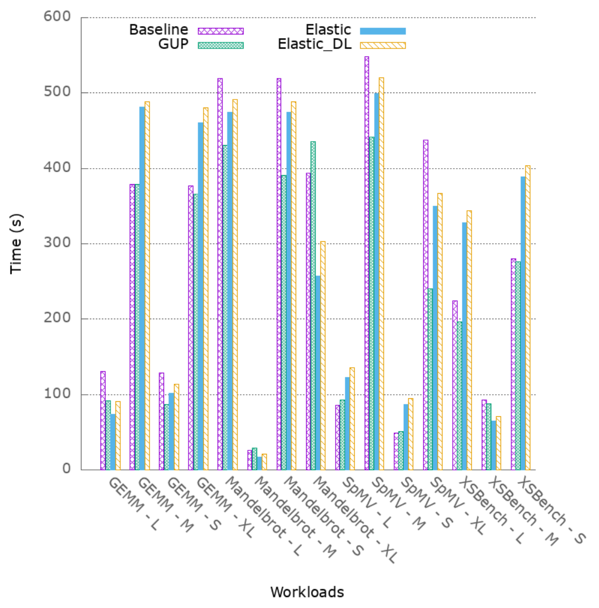
(a) Workload 4 - Baseline

(b) Workload 4 - G+Up

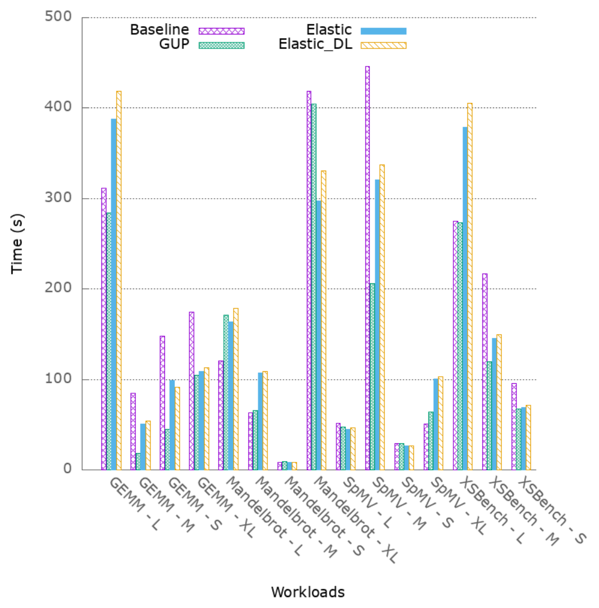
(c) Workload 4 - Elastic

(d) Workload 4 - Elastic-DL

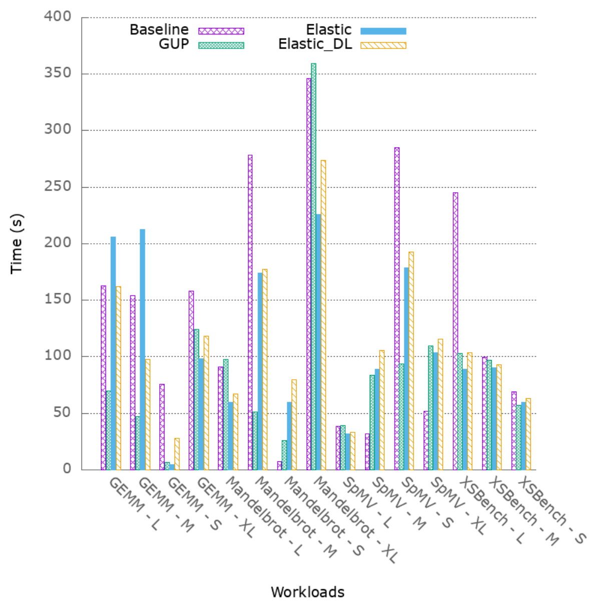
Figure 4.11: Timeline for the jobs on each accelerator in the two-application workload for each of the scheduling algorithms



(a) Workload 1



(b) Workload 2



(c) Workload 3

Figure 4.12: Turn around time for the jobs in the workloads for each scheduling algorithm.

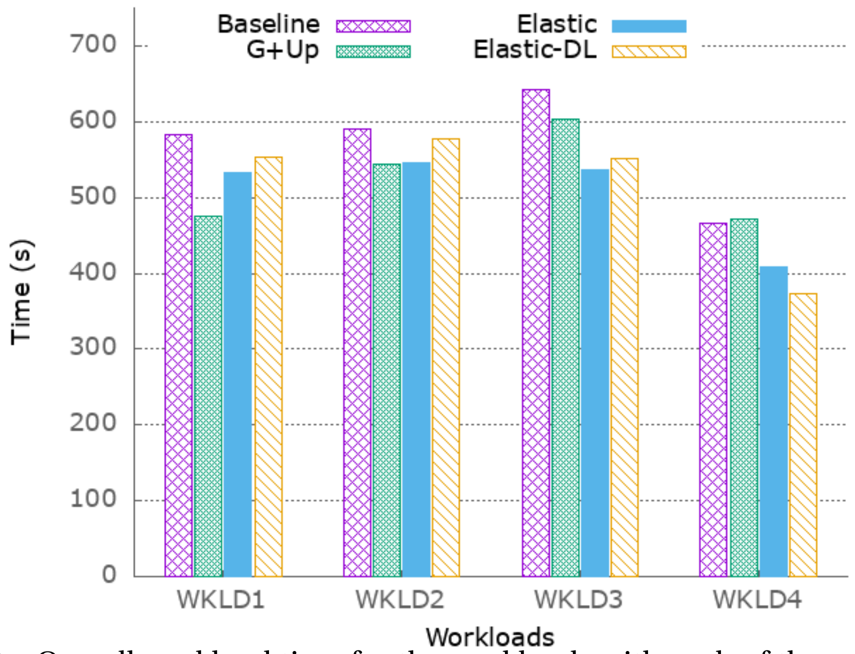


Figure 4.13: Overall workload time for the workloads with each of the scheduling algorithms

GEMM

In the case of GEMM (Figure 4.7(b)), only the GPU and FPGA show comparable performance. The CPU (omitted from figure) performs quite poorly for this particular kernel. The GPU performs on average 2.522x better than the FPGA for GEMM, but sharing work between the GPU and FPGA results in a speed-up of 1.386x over running exclusively on the GPU. Adding the CPU into the mix results in performance either being worse that running on the GPU alone (for 4096x4096), or worse than running on the GPU and FPGA (for 8192x8192), or only marginal improvements (for 16384x16384 and 22400x22400) with a speed-up of less than 1.01x over GPU and FPGA. This is because the added overhead of scheduling slices on the CPU is not amortized by adding the compute capability of the CPU. Another reason for this is the long tail created by the last slice allocated to the CPU. While the GPU and FPGA have completed their last slices, the CPU is still working on completing the last slice causing the long tail. The comparatively better performance of the FPGA can be attributed to the fact that the application uses the DSPs on-board the FPGA to improve GEMM performance.

SpMV

The work sharing results for SpMV are shown in Figure 4.7(c). Due to the low memory bandwidth on the FPGA, the performance of SpMV is more than an order of magnitude worse on the FPGA compared to the CPU and GPU and has therefore been omitted. The CPU performs best for this particular application with an average speed-up of 1.358x over the GPU. This is because SpMV has an irregular memory access pattern that benefits the CPU more than the GPU as the former favors regular memory access patterns. For SpMV, as seen before for other applications, sharing work between CPU and GPU results in better performance than using any single device with an average speed-up of 1.236x over the best performing device (CPU).

XSBench

Like SpMV, XSBench (shown in Figure 4.7(d)) also experiences poor performance on the FPGA, but unlike SpMV, the GPU performs better for this application with an average speed-up of 2.519x over the CPU. Sharing work between the CPU and GPU results in the most significant performance improvement with a speed-up of 1.3x over the GPU. Adding the FPGA along with the CPU and GPU results in either a slight increase in performance (16M) or a slight decrease in performance (32M). This makes FPGA an unsuitable candidate for work sharing for XSBench.

Discussion: Improvement in device technologies such as higher memory bandwidths on the FPGA, unified memory between accelerators, and adding more compute units to devices, can affect the relative performance of each of the accelerators for the applications studied. The work sharing framework would be able to leverage these enhancements to further improve overall application performance. Furthermore, the framework can be used on systems that utilize heterogeneous accelerators such as multiple but different GPUs. With the current framework, the output is copied out of the device when all slices are complete. A more fine-grained approach, where the output of each slice would be copied to the other devices while the kernel is still executing, could improve performance even more, particularly for applications that rely on multiple kernel executions such as stencil.

4.5.2 Co-Scheduling

Next, we will present results for co-scheduling the workloads discussed in the previous section. Figures 4.8, 4.9 and 4.10 shows the timeline for each job and the devices they are allocated for each of the randomly generated workloads and the scheduling algorithms. Figure 4.11 shows the timeline for the *dual application / 12 job* workload, where colors indicate the application while shades indicate input sizes of a given application per job. Figure 4.12 depicts the turn-around-time for each job per workload and scheduling algorithm. Finally, Figure 4.13 shows the time taken for the overall workload with each of the scheduling algorithms. While the previous section presented results for the computation time, the scheduling results are for overall job time (in seconds).

Workload 1

Figure 4.8 and Figure 4.12(a) depict results for Workload 1. For this workload with low inter-arrival times (2.2s on average), we see that Greedy+Up (G+Up) outperforms all other scheduling algorithms with a speed-up of 1.225x over the baseline. While both Elastic and Elastic-DL outperform the baseline (with a speed-up of 1.091x and 1.051x, respectively), they still show inferior performance when compared to G+Up. In terms of turn-around-time, we see that Elastic and Elastic-DL favor the longest and most elastic jobs to the detriment of most other jobs. We see idle time on certain devices, such as the GPU in Figure 4.8(b) in the time range of 100-150 seconds. This occurs because the scheduler has allocated the GPU to XSBench after it begins execution on the FPGA. Even though the scheduler assigns the GPU shortly after to XSBench, the job cannot switch to the newly assigned device until it either completes initialization or the current slice.

We show the timeline with initialization times for Figure 4.8(b). The dashes in the figure show when application initialization is complete, after which application starts its kernel execution. If application initialization has begun with a certain device allocation, it will hold the device until initialization has completed, even if another device has been assigned to the application. This is particularly visible for XSBench (large input), which starts on the FPGA as the GPU is still busy with GEMM (large), but after initialization and a single kernel run on the FPGA, XSBench is move to the now available GPU. An earlier switch to the GPU was not possible since any single kernel cannot be preempted but rather needs to first complete. And the FPGA had already been committed to the first XSBench kernel in this case. Notice that the delay in migration is dominated by initialization cost, which is high for XSBench (large), the single FPGA kernel run contributes only insignificantly to the migration delay. For other figures, dashed lines are omitted to improve legibility, but their timeline still includes application initialization time, including migration delays, albeit none of them as significant as in Figure 4.8(b).

Workload 2

Figure 4.9 and Figure 4.12(b) depict results for Workload 2 with an average inter-arrival time of 10.5s. We see that both G+Up and Elastic perform comparably with a speed-up of 1.085x and 1.08x, respectively, over the baseline. Elastic-DL experiences almost similar performance to baseline with a speed-up of 1.023x due to the overhead of switching between devices. We see a similar pattern with the turn-around-time as Workload 1, where Elastic and Elastic-DL give preference to the longest and most elastic jobs while showing an increase in the turn-around-time for most other jobs. The idle times on the GPU and FPGA at the beginning of the workload is due to there not being a job in the scheduler queue. Finally, we still see the impact of idle devices due to applications waiting for initialization and slice completion before switching devices.

Workload 3

Figure 4.10 and Figure 4.12(c) depict results for Workload 3, which has an inter-arrival time of 23.9s. We observe that both Elastic and Elastic-DL convincingly outperform G+Up migration. Elastic and Elastic-DL have speed-ups of 1.198x and 1.167x, respectively, while G+Up has a smaller speed-up of just 1.066x over the baseline. Because of the comparatively longer inter-arrival times, G+Up migration experiences significant idle times on the FPGA. Elastic and Elastic-DL, in contrast, leverage work-sharing resulting in higher utilization of the available accelerators on the system. We see improvement in turn-around-time for more jobs compared to Workloads 1 and 2 but it is still the longer and more elastic jobs that experience a marked decrease in turn-around-times. Furthermore, jobs that do suffer from a negative impact on their turn-around-times do so to a much lesser extent as compared to the previous workloads.

Workload 4

Finally, we run a workload consisting of two applications, GEMM and Mandelbrot, and a total of 12 jobs for each of our scheduling algorithms. Figure 4.11 shows the results for Workload 4 (average inter-arrival time of 2.25). We observe that Elastic-DL performs the best with a speed-up of 1.249x over the baseline while Elastic shows a speed-up of 1.142x over the baseline. G+Up, in contrast, shows a slight slowdown of 0.986x due to the overhead of device switching. This workload shows that applications with fine-grained slices as well as small application initialization times causes minimal idle times on devices, and therefore achieve superior performance with work-sharing even for low inter-arrival times.

Discussion: As for the work-sharing framework, the co-scheduling framework opens several areas for discussion:

1. While our scheduling algorithms prioritize arrival time of the jobs when allocating accelerators to jobs, other scheduling algorithms can be developed that schedule jobs based on relative performances of applications on different accelerators. For instance, a job that arrives later can be scheduled earlier if the available accelerator is better suited for it.

2. Better integration between the work-sharing and co-scheduling framework can further enhance performance. If the scheduler determines that the overhead of switching devices is more than running the job on the current device, it can let the application continue with the existing configuration. Furthermore, the scheduler can be made aware of the initialization time of the job. This would resolve a significant amount of the idle time that we see on devices in Figures 4.8, 4.9 and 4.10.

3. Third, instead of offline profiling, the scheduler can estimate relative performance of jobs on different accelerators in order to make scheduling decisions. This would eliminate the need to collect profiling data for each application as well as enable applications to run on arbitrary systems.

4. Finally, a two-phase scheduler (inter-node and intra-node) can be developed that uses MPI to enable multi-node co-scheduling of applications.

4.6 Related Work

Prior work has explored methods to efficiently use heterogeneous devices on HPC systems. Scogland et al. [SFRdS14] use OpenMP-like directives to schedule computational load across CPUs and GPUs. Aji et al. [APBF16] schedule task-parallel workloads on devices by mapping OpenCL queues to devices at run-time to achieve ideal performance. Spafford et al. [SMV10], Guzman et al. [GNT+19] and Pandit et al. [PG14] provide frameworks to orchestrate data and task decomposition for multi-device cooperative execution. Our work-sharing framework goes further by allowing dynamic migrations of kernels on devices and the ability to expand or contract based on the devices available on the system.

Several techniques have been explored in prior work to make maximum use of available resources by reducing idle time. Weidendorfer et al. [WB16] characterize applications for their suitability for co-scheduling. Frachtenberg et al. [FFPF03] use jobs as fillers to reduce fragmentation and achieve reduced idle time. Zacarias et al. [ZPN$^+$21] create a resource manager that uses machine learning to predict the cost of co-scheduling and a scheduler that reduces performance degradation. Further, Xiong et al. [XAHC18] propose Tangram, which oversubscribes nodes resulting in CPU sharing. They use prior knowledge to determine if co-scheduling will result in overall performance improvement. Dauwe et al. [DJF$^+$16] create a model to predict an application's execution time and energy usage when co-scheduled with other applications. They demonstrate that their model can significantly improve the scheduling performance.

In contrast to these works, to the best of our knowledge, ours is the first that holistically combines work sharing and elasticity of kernels with the ability to expand and contract as well as migrate kernels on devices combined with co-scheduling and enabled by pluggable scheduling algorithms to coordinate the execution of multiple kernels on the same node.

4.7 Conclusion

This work contributes novel methods to more effectively use the computational resources available in a modern or future HPC node. We create a framework for sharing work expressed as a single kernel across several different accelerators. We create capabilities to migrate a kernel from one device to another, expand the kernel to occupy more devices (potentially speeding up the application), or contract the kernel to occupy fewer devices (potentially slowing it down), all of which is done dynamically. Our framework further leverages these capabilities to enable sharing of a single node amongst multiple, concurrently running applications. We implement four scheduling algorithms to evaluate the efficacy of our co-scheduling framework.

Our framework is evaluated with four applications over newly contributed workloads comprising those applications with different input sizes. Our work-sharing results show that we can achieve a speedup of up to 2.26x when work is shared amongst all the available accelerators compared to our baseline. Furthermore, our co-scheduling experiments show that we can achieve a speedup of up to 1.25x with an Elastic-DL algorithm as opposed to the naive baseline co-scheduling approach. Finally we propose methods to further extend our framework to support multi-node applications, more accelerators and other scheduling algorithms.

CHAPTER

5

CONCLUSION

HPC systems are moving towards an increasingly heterogeneous architecture, which creates novel challenges for HPC centers and application developers in terms of how to best utilize the various computing, storage and networking resources available on a system. Tackling these challenges requires a combined effort for the emerging heterogeneity on each of the HPC subsystems. Furthermore, HPC centers need to incorporate specific heterogeneous resources balanced across the jobs that are expected to run on their systems while considering capital and operational expenditure constraints and device-specific performance characteristics.

In this work, we first assess the impact of heterogeneous storage resources on the I/O performance of various applications on different storage placements and network interconnects. We utilize PDES to demonstrate that the I/O performance of applications depends upon the placement of burst buffer resources in the network as well as the chosen network interconnect. Furthermore, we also investigate the impact of different burst buffer models and network interconnects on the capital and operational expenditures of the HPC centers.

Next, we study the impact of heterogeneity on the network performance of HPC applications. We use PDES to model GPUs as first-class compute devices with the ability to communicate directly with other GPUs, both within and across nodes, using GPUDirect and NVLink. Our simulation experiments show that the communication performance of applications under GPUDirect highly depends on the messaging characteristics of the applications. We also show that the density of accelerators on a node significantly impacts both the inter-node and intra-node communication performance of applications.

Finally, we explore the emerging compute heterogeneity in modern and next generation HPC systems. We create a work sharing and co-scheduling framework that allows applications to cooperatively execute a single compute kernel on multiple accelerators. Our framework allows a kernel to elastically schedule slices on multiple devices with the ability to migrate computation from one accelerator to another, expand to more accelerators, or contract to fewer accelerators. Our framework leverages this ability to schedule multiple applications simultaneously. We evaluate our framework for four applications and workloads. Our results show that effective work sharing is dependent on the applications and the available accelerators while co-scheduling performance is dependent on the scheduling algorithm and the job mix of the workload.

In summary, we have shown that our hypothesis holds since the effective utilization of heterogeneous resources requires software support to identify ideal placements, application tuning with respect to heterogeneous resources as well as workload scheduling decisions that leverages elastic and transparent scheduling of resources within and across applications.

REFERENCES

[ABD+09] Jung Ho Ahn, Nathan Binkert, Al Davis, Moray McLaren, and Robert S Schreiber. Hyperx: topology, routing, and packaging of efficient large-scale networks. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 41. ACM, 2009.

[ABE+21] Yehia Arafa, Abdel-Hameed Badawy, Ammar ElWazir, Atanu Barai, Ali Eker, Gopinath Chennupati, Nandakishore Santhi, and Stephan Eidenbenz. Hybrid, scalable, trace-driven performance modeling of gpgpus. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.

[AJB+15] Bilge Acun, Nikhil Jain, Abhinav Bhatele, Misbah Mubarak, Christopher D. Carothers, and Laxmikant V. Kale. Preliminary evaluation of a parallel trace replay tool for hpc network simulations. In *Proceedings of the 3rd Workshop on Parallel and Distributed Agent-Based Simulations*, PADABS '15, August 2015. LLNL-CONF-667225.

[APBF16] Ashwin M Aji, Antonio J Peña, Pavan Balaji, and Wu-chun Feng. Multicl: Enabling automatic scheduling for task-parallel workloads in opencl. *Parallel Computing*, 58:37–55, 2016.

[BBR+16] Wahid Bhimji, Debbie Bard, Melissa Romanus, David Paul, Andrey Ovsyannikov, Brian Friesen, Matt Bryson, Joaquin Correa, Glenn K Lockwood, Vakho Tsulaia, et al. Accelerating science with the nersc burst buffer early user program. *CUG2016 Proceedings*, 2016.

[BDE+12] Christopher Baker, Gregory Davidson, Thomas M Evans, Steven Hamilton, Joshua Jarrell, and Wayne Joubert. High performance radiation transport simulations: preparing for titan. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–10. IEEE, 2012.

[Ber21] Zuse Institute Berlin. Konrad. `http://www.zib.de/features/supercomputing-zib`, 2021.

[BH14] Maciej Besta and Torsten Hoefler. Slim fly: A cost effective low-diameter network topology. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 348–359. IEEE Press, 2014.

[BTL10] Brahim Betkaoui, David B Thomas, and Wayne Luk. Comparing performance and energy efficiency of fpgas and gpus for high productivity computing. In

2010 International Conference on Field-Programmable Technology, pages 94–101. IEEE, 2010.

[CBP02] Christopher D Carothers, David Bauer, and Shawn Pearce. Ross: A high-performance, low-memory, modular time warp system. *Journal of Parallel and Distributed Computing*, 62(11):1648–1669, 2002.

[Cen21] National Energy Research Scientific Computing Center. Cori. `http://www.nersc.gov/users/computational-systems/cori/`, 2021.

[CES16] Guillaume Chapuis, Stephan Eidenbenz, and Nandakishore Santhi. Gpu performance prediction through parallel discrete event simulation and common sense. In *Proceedings of the 9th EAI International Conference on Performance Evaluation Methodologies and Tools*, pages 204–211, 2016.

[CLL+11] Jason Cope, Ning Liu, Sam Lang, Phil Carns, Chris Carothers, and Robert Ross. Codes: Enabling co-design of multilayer exascale storage architectures. In *Proceedings of the Workshop on Emerging Supercomputing Technologies*, volume 2011, 2011.

[CLR+09] Philip Carns, Robert Latham, Robert Ross, Kamil Iskra, Samuel Lang, and Katherine Riley. 24/7 characterization of petascale i/o workloads. In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*, pages 1–10. IEEE, 2009.

[CPF99] Christopher D Carothers, Kalyan S Perumalla, and Richard M Fujimoto. Efficient optimistic parallel simulations using reverse computation. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 9(3):224–253, 1999.

[CRKB20] Jaemin Choi, David F Richards, Laxmikant V Kale, and Abhinav Bhatele. End-to-end performance modeling of distributed gpu applications. In *Proceedings of the 34th ACM International Conference on Supercomputing*, pages 1–12, 2020.

[CSB17] Lei Cao, Bradley W Settlemyer, and John Bent. To share or not to share: comparing burst buffer architectures. In *Proceedings of the 25th High Performance Computing Symposium*, page 4. Society for Computer Simulation International, 2017.

[DJF+16] Daniel Dauwe, Eric Jonardi, Ryan D Friese, Sudeep Pasricha, Anthony A Maciejewski, David A Bader, and Howard Jay Siegel. Hpc node performance and energy modeling with the co-location of applications. *The Journal of Supercomputing*, 72(12):4771–4809, 2016.

[DRP11] Rob Dimond, Sébastien Racaniere, and Oliver Pell. Accelerating large-scale hpc applications using fpgas. In *2011 IEEE 20th Symposium on Computer Arithmetic*, pages 191–192. IEEE, 2011.

[DSG14] Xiang Wei Duan, Wei Chang Shen, and Jun Guo. The mpi and openmp implementation of parallel algorithm for generating mandelbrot set. In *Applied Mechanics and Materials*, volume 571, pages 26–29. Trans Tech Publ, 2014.

[EWG+11] Dominic Eschweiler, Michael Wagner, Markus Geimer, Andreas Knüpfer, Wolfgang E Nagel, and Felix Wolf. Open trace format 2: The next generation of scalable trace formats and support libraries. In *PARCO*, volume 22, pages 481–490, 2011.

[FD17] Denis Foley and John Danskin. Ultra-performance pascal gpu and nvlink interconnect. *IEEE Micro*, 37(2):7–17, 2017.

[FFPF03] Eitan Frachtenberg, Dror G Feitelson, Fabrizio Petrini, and Juan Fernandez. Flexible coscheduling: Mitigating load imbalance and improving utilization of heterogeneous resources. In *Proceedings International Parallel and Distributed Processing Symposium*, pages 10–pp. IEEE, 2003.

[Fuj90] Richard M Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, 1990.

[GB20] Bhanuka Manesha Samarasekara Vitharana Gamage and Vishnu Monn Baskaran. Efficient generation of mandelbrot set using message passing interface. *arXiv preprint arXiv:2007.00745*, 2020.

[GNT+19] María Angélica Dávila Guzmán, Raúl Nozal, Rubén Gran Tejero, María Villarroya-Gaudó, Darío Suárez Gracia, and Jose Luis Bosque. Cooperative cpu, gpu, and fpga heterogeneous execution with enginecl. *The Journal of Supercomputing*, 75(3):1732–1746, 2019.

[HAL+16] Stephen Herbein, Dong H Ahn, Don Lipari, Thomas RW Scogland, Marc Stearman, Mark Grondona, Jim Garlick, Becky Springmeyer, and Michela Taufer. Scalable i/o-aware job scheduling for burst buffer enabled hpc clusters. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, pages 69–80. ACM, 2016.

[HOAV16] Kevin Harms, H Sarp Oral, Scott Atchley, and Sudharshan S Vazhkudai. Impact of burst buffer architectures on application portability. Technical report, Oak Ridge National Laboratory (ORNL), Oak Ridge, TN (United States). Oak Ridge Leadership Computing Facility (OLCF), 2016.

[HOP+86] James O Henriksen, Robert M O'Keefe, C Dennis Pegden, Robert G Sargent, Brian W Unger, and Douglas W Jones. Implementations of time (panel). In *Proceedings of the 18th conference on Winter simulation*, pages 409–416. ACM, 1986.

[HPF+16] Salman Habib, Adrian Pope, Hal Finkel, Nicholas Frontiere, Katrin Heitmann, David Daniel, Patricia Fasel, Vitali Morozov, George Zagaris, Tom Peterka, et al. Hacc: Simulating sky surveys on state-of-the-art supercomputing architectures. *New Astronomy*, 42:49–65, 2016.

[Inc21] NVIDIA Inc. Gpudirect rdma. `https://docs.nvidia.com/cuda/gpudirect-rdma/index.html`, 2021.

[Int] Intel. Intel oneapi. `https://software.intel.com/en-us/oneapi`.

[Jef85] David R Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(3):404–425, 1985.

[JLBF10] Yang Jiao, Heshan Lin, Pavan Balaji, and Wu-chun Feng. Power and performance characterization of computational kernels on the gpu. In *2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*, pages 221–228. IEEE, 2010.

[KBD+08] Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S Müller, and Wolfgang E Nagel. The vampir performance analysis tool-set. In *Tools for high performance computing*, pages 139–155. Springer, 2008.

[KDSA08] John Kim, Wiliam J Dally, Steve Scott, and Dennis Abts. Technology-driven, highly-scalable dragonfly topology. In *Computer Architecture, 2008. ISCA'08. 35th International Symposium on*, pages 77–88. IEEE, 2008.

[KES+09] V. V. Kindratenko, J. J. Enos, G. Shi, M. T. Showerman, G. W. Arnold, J. E. Stone, J. C. Phillips, and W. Hwu. Gpu clusters for high-performance computing. In *2009 IEEE International Conference on Cluster Computing and Workshops*, pages 1–8, 2009.

[Khr] Khronos. Opencl api. `https://khronos.org/registry/OpenCL/sdk/1.0/docs/man/xhtml/`.

[KMKS13] Ian Karlin, Jim McGraw, Jeff Keasler, and Bert Still. Tuning the lulesh mini-app for current and future hardware. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2013.

[KMM+12] Dries Kimpe, Kathryn Mohror, Adam Moody, Brian Van Essen, Maya Gokhale, Rob Ross, and Bronis R De Supinski. Integrated in-system storage architecture for high performance computing. In *Proceedings of the 2nd International Workshop on Runtime and Operating Systems for Supercomputers*, page 4. ACM, 2012.

[KRaM+12] Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen Malony, et al. Score-p: A joint performance measurement run-time infrastructure for periscope, scalasca, tau, and vampir. In *Tools for High Performance Computing 2011*, pages 79–91. Springer, 2012.

[KZM+19] Harsh Khetawat, Christopher Zimmer, Frank Mueller, Scott Atchley, Sudharshan S Vazhkudai, and Misbah Mubarak. Evaluating burst buffer placement in hpc systems. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–11. IEEE, 2019.

[Lab21a] Lawrence Livermore National Laboratory. Quartz. `https://hpc.llnl.gov/hardware/platforms/Quartz`, 2021.

[Lab21b] Lawrence Livermore National Laboratory. Sierra. `https://computing.llnl.gov/computers/sierra`, 2021.

[Lab21c] Oak Ridge National Laboratory. Summit. `https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/`, 2021.

[Lab21d] Oak Ridge National Laboratory. Titan. `https://www.olcf.ornl.gov/olcf-resources/compute-systems/titan/`, 2021.

[LCC+12] Ning Liu, Jason Cope, Philip Carns, Christopher Carothers, Robert Ross, Gary Grider, Adam Crume, and Carlos Maltzahn. On the role of burst buffers in leadership-class storage systems. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, pages 1–11. IEEE, 2012.

[Lei85] Charles E Leiserson. Fat-trees: universal networks for hardware-efficient supercomputing. *IEEE transactions on Computers*, 100(10):892–901, 1985.

[LHL+98] Zhihong Lin, Taik Soo Hahm, WW Lee, William M Tang, and Roscoe B White. Turbulent transport reduction by zonal flows: Massively parallel simulations. *Science*, 281(5384):1835–1837, 1998.

[LHSJ15] Ning Liu, Adnan Haider, Xian-He Sun, and Dong Jin. Fattreesim: Modeling large-scale fat-tree networks for hpc systems and data centers using parallel and discrete event simulation. In *Proceedings of the 3rd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, pages 199–210, 2015.

[LKB+16] Edgar A León, Ian Karlin, Abhinav Bhatele, Steven H Langer, Chris Chambreau, Louis H Howell, Trent D'Hooge, and Matthew L Leininger. Characterizing parallel scientific applications on commodity clusters: An empirical study of a tapered fat-tree. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 909–920. IEEE, 2016.

[LSC+19] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R Tallent, and Kevin J Barker. Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect. *IEEE Transactions on Parallel and Distributed Systems*, 31(1):94–110, 2019.

[LYC+18] David Lignell, Chun Sang Yoo, Jacqueline Chen, Ramanan Sankaran, Mark Fahey, Evatt Hawkes, Tianfeng Lu, and Chung Law. S3d: Petascale combustion science, performance, and optimization collaborators. *Proceedings of the Cray Scaling Workshop*, 06 2018.

[MCJ+17] Misbah Mubarak, Philip Carns, Jonathan Jenkins, Jianping Kelvin Li, Nikhil Jain, Shane Snyder, Robert Ross, Christopher D Carothers, Abhinav Bhatele, and Kwan-Liu Ma. Quantifying i/o and communication traffic interference on dragonfly networks equipped with burst buffers. In *Cluster Computing (CLUSTER), 2017 IEEE International Conference on*, pages 204–215. IEEE, 2017.

[MCRC16] Misbah Mubarak, Christopher D Carothers, Robert B Ross, and Philip Carns. Enabling parallel simulation of large-scale hpc network systems. *IEEE Transactions on Parallel and Distributed Systems*, 28(1):87–100, 2016.

[MCRC17] Misbah Mubarak, Christopher D Carothers, Robert B Ross, and Philip Carns. Enabling parallel simulation of large-scale hpc network systems. *IEEE Transactions on Parallel and Distributed Systems*, 28(1):87–100, 2017.

[NAC+15] Maxim Naumov, M Arsaev, Patrice Castonguay, J Cohen, Julien Demouth, Joe Eaton, S Layton, N Markovskiy, István Reguly, Nikolai Sakharnykh, et al. Amgx: A library for gpu accelerated algebraic multigrid and preconditioned iterative methods. *SIAM Journal on Scientific Computing*, 37(5):S602–S626, 2015.

[NER] NERSC. Perlmutter. `https://www.nersc.gov/systems/perlmutter/`.

[NMS+21] Tan Nguyen, Colin MacLean, Marco Siracusa, Douglas Doerfler, Nicholas J Wright, and Samuel Williams. Fpga-based hpc accelerators: An evaluation on performance and energy efficiency. *Concurrency and Computation: Practice and Experience*, page e6570, 2021.

[Nvi07] CUDA Nvidia. Compute unified device architecture programming guide. *Nvidia*, 2007.

[OHL+08] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[Ope21] OpenMPI. Openmpi: Running cuda-aware open mpi. `https://www.open-mpi.org/faq/?category=runcuda`, 2021.

[ORN] ORNL. Frontier. `https://www.olcf.ornl.gov/frontier/`.

[PC03] Michael D Peters and Christopher D Carothers. Parallel distributed simulation and modeling methods: an algorithm for fully-reversible optimistic parallel simulation. In *Proceedings of the 35th conference on Winter simulation: driving innovation*, pages 864–871. Winter Simulation Conference, 2003.

[PCT07] Steve Plimpton, Paul Crozier, and Aidan Thompson. Lammps-large-scale atomic/molecular massively parallel simulator. *Sandia National Laboratories*, 18:43–43, 2007.

[PG14] Prasanna Pandit and R Govindarajan. Fluidic kernels: Cooperative execution of opencl programs on multiple heterogeneous devices. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 273–283, 2014.

[PHJM11] Simon J Pennycook, Simon D Hammond, Stephen A Jarvis, and Gihan R Mudalige. Performance analysis of a hybrid mpi/cuda implementation of the naslu benchmark. *ACM SIGMETRICS Performance Evaluation Review*, 38(4):23–29, 2011.

[PHV$^+$13] Sreeram Potluri, Khaled Hamidouche, Akshay Venkatesh, Devendar Bureddy, and Dhabaleswar K Panda. Efficient inter-node mpi communication using gpudirect rdma for infiniband clusters with nvidia gpus. In *2013 42nd International Conference on Parallel Processing*, pages 80–89. IEEE, 2013.

[PS17] NA Petersson and B Sjogreen. Sw4, version 2.0 [software], computational infrastructure for geodynamics, 2017, doi: 10.5281/zenodo. 1045297, 2017.

[SAL$^+$11] Gilad Shainer, Ali Ayoub, Pak Lui, Tong Liu, Michael Kagan, Christian R Trott, Greg Scantlen, and Paul S Crozier. The development of mellanox/nvidia gpudirect over infiniband—a new model for gpu to gpu communications. *Computer Science-Research and Development*, 26(3-4):267–273, 2011.

[SFRdS14] Thomas RW Scogland, Wu-chun Feng, Barry Rountree, and Bronis R de Supinski. Coretsar: Adaptive worksharing for heterogeneous systems. In *International Supercomputing Conference*, pages 172–186. Springer, 2014.

[SGS10] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66–73, 2010.

[SMV10] Kyle Spafford, Jeremy Meredith, and Jeffrey Vetter. Maestro: data orchestration and tuning for opencl devices. In *European Conference on Parallel Processing*, pages 275–286. Springer, 2010.

[Top20] Top500. Top500 hpc systems. `https://www.top500.org/lists/top500/2020/11/`, 2020.

[TSIS14] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. Xsbench-the development and verification of a performance abstraction for monte carlo reactor analysis. *The Role of Reactor Physics toward a Sustainable Future (PHYSOR)*, 2014.

[VB13] Wim Vanderbauwhede and Khaled Benkrid. *High-performance computing using FPGAs*, volume 3. Springer, 2013.

[VBG$^+$19] Jeffrey S Vetter, Ron Brightwell, Maya Gokhale, Pat McCormick, Rob Ross, John Shalf, Katie Antypas, David Donofrio, Travis Humble, Catherine Schuman, et al. Extreme heterogeneity 2018-productive computational science in the era of extreme heterogeneity: Report for doe ascr workshop on extreme heterogeneity. 2019.

[VdSB$^+$18] Sudharshan S. Vazhkudai, Bronis R. de Supinski, Arthur S. Bland, Al Geist, James Sexton, Jim Kahle, Christopher J. Zimmer, Scott Atchley, Sarp Oral, Don E. Maxwell, Veronica G. Vergara Larrea, Adam Bertsch, Robin Goldstone, Wayne Joubert, Chris Chambreau, David Appelhans, Robert Blackmore, Ben Casses, George Chochia, Gene Davison, Matthew A. Ezell, Tom Gooding, Elsa Gonsiorowski, Leopold Grinberg, Bill Hanson, Bill Hartner, Ian Karlin, Matthew L. Leininger, Dustin Leverman, Chris Marroquin, Adam Moody, Martin Ohmacht, Ramesh Pankajakshan, Fernando Pizzano, James H. Rogers, Bryan Rosenburg, Drew Schmidt, Mallikarjun Shankar, Feiyi Wang, Py Watson, Bob Walkup, Lance D. Weems, and Junqi Yin. The design, deployment, and evaluation of the coral pre-exascale systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18, pages 52:1–52:12, Piscataway, NJ, USA, 2018. IEEE Press.

[WB16] Josef Weidendorfer and Jens Breitbart. Detailed characterization of hpc applications for co-scheduling. In *Proceedings of the 1st COSH Workshop on Co-Scheduling of HPC Applications*, page 19, 2016.

[WCM$^+$16] Noah Wolfe, Christopher D Carothers, Misbah Mubarak, Robert Ross, and Philip Carns. Modeling a million-node slim fly network using parallel discrete-event simulation. In *Proceedings of the 2016 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, pages 189–199, 2016.

[WOP$^+$15] Teng Wang, Sarp Oral, Michael Pritchard, Bin Wang, and Weikuan Yu. Trio: burst buffer based i/o orchestration. In *Cluster Computing (CLUSTER), 2015 IEEE International Conference on*, pages 194–203. IEEE, 2015.

[WOW$^+$14] Teng Wang, Sarp Oral, Yandong Wang, Brad Settlemyer, Scott Atchley, and Weikuan Yu. Burstmem: A high-performance burst buffer system for scientific applications. In *Big Data (Big Data), 2014 IEEE International Conference on*, pages 71–79. IEEE, 2014.

[XAHC18] Qingqing Xiong, Emre Ates, Martin C Herbordt, and Ayse K Coskun. Tangram: Colocating hpc applications with oversubscription. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2018.

[XCD+12] Bing Xie, Jeffrey Chase, David Dillow, Oleg Drokin, Scott Klasky, Sarp Oral, and Norbert Podhorszki. Characterizing output bottlenecks in a supercomputer. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–11. IEEE, 2012.

[YJM+16] Xu Yang, John Jenkins, Misbah Mubarak, Xin Wang, Robert B Ross, and Zhiling Lan. Study of intra-and interjob interference on torus networks. In *Parallel and Distributed Systems (ICPADS), 2016 IEEE 22nd International Conference on*, pages 239–246. IEEE, 2016.

[ZPN+21] Felippe Vieira Zacarias, Vinicius Petrucci, Rajiv Nishtala, Paul Carpenter, and Daniel Mossé. Intelligent colocation of hpc workloads. *Journal of Parallel and Distributed Computing*, 151:125–137, 2021.